universität ulm

Doctoral Thesis

Empirical Assessment of Advantages and Disadvantages of Model Transformation Languages

Author:
Stefan Höppner (née Götz)
born in Ehingen (Donau)

Supervisor:
Prof. Dr. Matthias Tichy

A thesis submitted in fulfilment of the requirements for the degree of Dr.rer.nat

at the

Institute of Software Engineering and Programming Languages
Faculty of Engineering, Computer Science and Psychology

2023

Empirical Assessment of Advantages and Disadvantages of Model Transformation Languages

Stefan HÖPPNER (née Götz)

Acting Dean:
Prof. Dr. Anke Huckauf

Reviewers:
Prof. Dr. Matthias Tichy
Prof. Dr. Regina Hebig
Prof. Dr. Daniel Strüber

Day of PhD Graduation: 19.07.2023

"*If the only tool you have is a hammer, it is tempting to treat everything as if it were a nail.*"

Abraham Maslow

Abstract

Context: Model-driven software engineering envisages the use of model transformations to evolve models. Model transformation languages, domain-specific languages developed for this task, are touted with many benefits over general-purpose languages. However, most of these claims have neither been substantiated nor investigated thoroughly. Moreover, they frequently lack the contextual information required to critically assess their merit or build meaningful empirical studies around them.

Objective: The main objective of this thesis is to aggregate all necessary data to set up proper evaluation and use this data to asses the most prevalent claims about model transformation languages empirically. We aim to provide evidence on whether those claims withstand rigorous empirical scrutiny. We further want to provide a foundation of data upon which more empirical evaluation can be built.

Method: To address our objectives, we employ several research methodologies. Initially, we use a structured literature review to determine the state of research and what claims about quality attributes of MTLs are propagated in literature. The SLR results serve as a basis for conducting semi-structured interviews to collect qualitative data on relevant factors and co-founding factors pertaining to the claims discussed. We quantify the identified influences between quality attributes, factors and co-founding variables using structural equation modelling and an online survey. Finally, we use repository mining and design science to collect and prepare artefacts. The artefacts are used in two separate case studies to empirically evaluate several MTL quality attributes based on the previously identified factors.

Results: Our results show that many quality attributes believed to be associated with MTLs are in dire need of empirical evidence. To aid in this task, we contribute a quantified structure model describing factors' influence and moderation effects on quality attributes of MTLs. The model aggregates the results of our literature review, interview study and online survey using structural equation modelling. The literature review produced a comprehensive list of quality attributes for which advantages and disadvantages of MTLs are claimed. The interviews resulted in factors contributing to the perception of quality attributes of MTLs and several co-founding factors defining context for the evaluation thereof. Data from the online survey is then used for quantification. Based on the in-depth discussions during our interviews, we further contribute 15 suggestions on actions requiring community-wide effort to improve confidence in the usefulness of model transformation languages. These are further refined based on the quantitative results of the online survey. We also make a direct contribution through the results of our case studies. The first case study provides empirical evidence of how well the ATL transformation language is suited to a category of model transformation problems. The second case study demonstrates the shortcomings of model transformations written in both legacy and modern Java styles. To execute these evaluations, we developed a novel approach for translating ATL transformations into Java code and a classification schema for model transformations written in Java.

Conclusion: Our results demonstrate that empirical evaluation of model transformation languages is feasible and necessary. Efforts to provide more empirical substance need to be undergone, and lacklustre language capabilities and tooling need to be improved. The results of this thesis can provide a basis for these further actions.

Acknowledgements

There is a lot of people without whom this thesis would not have been possible.

First, my supervisor and mentor Matthias Tichy, for allowing me to embark on this journey, his guidance at every step along the way and all his insightful comments, suggestions, and reviews. I could not have asked for a more invested and helpful supervisor.

Jens Kohlmeyer and Klaus Murmann for presenting me with the opportunity to start my PhD and giving me every possible room to do finish it.

My co-authors for supporting me along the way: Raffaela Groner, for being my go-to colleague all things MTL related. Timo Kehrer, for being my second mentor during our work together. Yves Haas, for doing a lot of the leg work to get the interview study going.

All my colleagues for the insightful discussions during our time together.

I would like to thank all my student assistants at SGI over the years, Burak Aktas, Julian Czymmeck, Kevin Jedlhauser, Max-Immanuel Appel, Melissa Loos, Micha Götz, Niklas Haas, Philipp Bitzer, Sergej Schmidt, Tobias Guggenmos, and Yves Haas. If it weren't for you guys having my back for all things SGI related, I'd probably still be working on installing Ubuntu 18.04 in the PC pool and writing my first paper.

Thank you to my friends, who have been there since school days and have allowed me to have a life outside of school and uni.

I owe much of what I have achieved to my parents, who let me choose my own path but were there for me whenever I needed support. I strive to one day make it all up to you and be the person you always knew I could be.

Finally, Anke. You have been my guiding light throughout this whole process. You encouraged me, believed in me, kept me on course and reigned me in. This is for all the support you gave me.

List of Publications

Appended publications

This thesis is based on the following publications:

[A] S. Götz, M. Tichy, R. Groner. Claimed advantages and disadvantages of (dedicated) model transformation languages: a systematic literature review
Software and Systems Modeling (SoSyM), volume 20, pages 469–503, 2021

[B] S. Höppner, Y. Haas, M. Tichy, K. Juhnke. Advantages and disadvantages of (dedicated) model transformation languages: A Qualitative Interview Study
Empirical Software Engineering (EMSE), volume 27, article number 159, 2022

[C] S. Höppner, M. Tichy. Traceability and Reuse Mechanisms, the most important Properties of Model Transformation Languages
Empirical Software Engineering (EMSE), under review

[D] S. Götz, M. Tichy. Investigating the Origins of Complexity and Expressiveness in ATL Transformations
Journal of Object Technology (JoT), volume 19, article number 2, July 2020

[E] S. Höppner, M. Tichy, T. Kehrer. Contrasting Dedicated Model Transformation Languages Versus General Purpose Languages: A Historical Perspective on ATL Versus Java Based on Complexity and Size
Software and Systems Modeling (SoSyM), volume 21, pages 805–837, 2022

Other publications

The following publications complete the list of my research contributions during my PhD studies. They are not appended to this thesis, due to overlapping contents or not being related to the thesis.

Journal (Peer reviewed)

[a] R. Groner, K. Juhnke, S. Götz, M. Tichy, S. Becker, V. Vijayshree, S. Frank "A Survey on the Relevance of the Performance of Model Transformations"
Journal of Object Technology (JoT), volume 20(2), 2:1-27, 2021

Conference (Peer reviewed)

[a] S. Kögel, M. Tichy, R. Groner, M. Stegmaier, S. Götz, S. Rechenberger "Developing an Optimizing Compiler for the Game Boy as a Software Engineering Project"
Software Engineering Education and Training Track of the International Conference on Software Engineering (ICSE SEET), 2018

[b] S. Götz, M. Tichy, Timo Kehrer "Dedicated Model Transformation Languages vs. General-Purpose Languages: A Historical Perspective on ATL vs. Java"
9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2021

[c] S. Götz, M. Tichy, R. Groner "Claimed advantages and disadvantages of (dedicated) model transformation languages: a systematic literature review (extended abstract)"
Software Engineering (SE), 2021

[d] S. Höppner, T. Kehrer, M. Tichy "Contrasting Dedicated Model Transformation Languages vs. General Purpose Languages: A Historical Perspective on ATL vs. Java based on Complexity and Size (extended abstract)"
Software Engineering (SE), 2022

[e] S. Höppner, M. Tichy "The Relevance of Model Transformation Language Features on Qualitative Properties of MTLs: A Study Protocol"
Empirical Software Engineering and Measures (ESEM), 2022

[f] S. Stiess, S. Höppner, F. Ege, M. Tichy "Event-based Simulation for Transient System with Capture Replay to Predict Self-adaptive Systems (Work in Progress Paper)"
14th ACM/SPEC International Conference on Performance Engineering (ICPE), 2023

[g] R. Groner, P. Bellmann, S. Höppner, P. Thiam, F. Schwenker, M. Tichy "Predicting the Performance of ATL Model Transformations"
14th ACM/SPEC International Conference on Performance Engineering (ICPE), 2023

Workshop (Peer reviewed)

[a] S. Stiess, S. Höppner, F. Ege, M. Tichy, S. Becker "Coordination and Explanation of Reconfigurations in Self-adaptive high-performance Systems"
2nd International Workshop on Model-Driven Engineering of Digital Twins (ModDiT), 2022

[b] S. Höppner, S. Stiess, F. Ege, M. Tichy "State Space Exploration for Planning Reconfigurations in Cloud-native Systems"
13th Symposium on Software Performance (SSP), 2022

[c] S. Greiner, S. Höppner, Frederic Jouault, Theo Le Calvar, Mickael Clavreul "Incremental MTL vs. GPLs: Class into Relational Database Schema"
Transformation Tool Contest (TTC), 2023

Research Contribution

I am the main contributor for all publications appended to this thesis. I had varying degrees of help by the co-authors of each publication. Each publication was composed by me with revisions based on reviews from the co-authors.

My contribution for Paper A was the study design, conduction of search and snowballing as well as the analysis of data. I was supported by the co-authors during the selection process.

In Paper B, I designed the study together with one of the co-authors. Apart from one instance, all interviews were conducted by me. Transcription was split between me and one co-author. Analysis of the transcripts was done by me with some supporting work by all co-authors.

Paper C was designed, conducted and analysed by me. I got help in selection of the research methodology by the co-author.

Similarly, for Paper D, I contributed the research design and conducted the analysis.

For Paper E, I was supported by both co-authors during the development of the translation schema. All other work was conducted by me.

Contents

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
ASE	**A**verage **S**imulated **E**ffect
AST	**A**bstract **S**yntax **T**ree
ATL	**A**tlas **T**ransformation **L**anguage
CIM	**C**omputation **I**ndependent **M**odel
DFG	**D**eutsche **F**orschungs**g**emeinschaft
DSL	**D**omain-**S**pecific **L**anguage
EMF	**E**clipse **M**odelling **F**ramework
GLSP	**G**raphical **L**anguage **S**erver **P**rotocol
GPL	**G**eneral **P**urpose **L**anguage
LOC	**L**ines **O**f **C**ode
LSP	**L**anguage **S**erver **P**rotocol
MBE	**M**odel **B**ased **E**ngineering
MDA	**M**odel **D**riven **A**rchitecture
MDD	**M**odel **D**riven **D**evelopment
MDE	**M**odel **D**riven **E**ngineering
MDSE	**M**odel **D**riven **S**oftware **E**ngineering
MOF	**M**eta-**O**bject **F**acility
MT	**M**odel **T**ransformation
MTL	**M**odel **T**ransformation **L**anguage
OCL	**O**bject **C**onstraint **L**anguage
OEAD	**O**verall **E**xplained **A**bsolute **D**eviation
OMG	**O**bject **M**anagement **G**roup
PIM	**P**latform **I**ndependent **M**odel
PSM	**P**latform **S**pecific **M**odel
QVT	**Q**uery-**V**iew-**T**ransform
SEM	**S**tructural **E**quation **M**odelling
SLR	**S**tructured **L**iterature **R**eview
TTC	**T**ransformation **T**ool **C**ontest
USM	**U**niversal **S**tructure **M**odelling
WMC	**W**eighted **M**ethod **C**ount

Chapter 1

Introduction

Sendall et al. (2003) describe model transformations as the *"heart and soul of model-driven software development"*. It stands to reason that this particular task has been the focus of much research. Notably, a large number of dedicated domain-specific languages, called model transformation languages (MTLs), have been developed to aid practitioners in developing model transformations. MTLs are touted with many advantages in areas like productivity, expressiveness and comprehensibility compared to general purpose programming languages (GPLs) for developing model transformations (Mernik et al. 2005; Sendall et al. 2003; Tratt 2005). However, little focus is put into providing evidence for these claims. Researchers tend to prioritize the development of new abstractions or the expansion of existing concepts to explore advanced and nice application domains, at the expense of properly assessing their practical utility. As a result, increasingly many features and new languages are being developed while evaluation is neglected.

One can argue that this is a fair use of resources because it is aligns with the purpose of research. However, as the model-driven paradigm gains maturity, industry picks up more of its concepts and new trends like AI emerge, confidence in one of MDSE's core elements is imperative.

This is best highlighted in a study by Burgueño et al. (2019). They found that social aspects like acceptance of the MDSE are a big issue for the community. As a result, proper empirical evaluation to provide evidence of the advantages of MTLs is becoming increasingly important. Furthermore, evaluation can help to detect current shortcomings that can be addressed to improve the usefulness of transformation languages. It can also assist in precisely defining those areas where using model transformation languages is most potent. Lastly, it helps to identify those features of model transformation languages that are most useful for transformation developers, allowing language development to focus on the correct areas to make the languages more streamlined for practical use. All this can help improve research on MTLs and help increase their acceptance by researchers and industry.

In the course of our work, we identified several hurdles that hamper extensive empirical evaluations of MTLs. First, little explicit knowledge exists about the quality attributes associated with model transformation languages (Götz et al. 2021a). Most of it exists solely in the minds of researchers and users. There is also no unified designation of the quality attributes, and different descriptions often exist for the same aspects. Moreover, there is a lack of knowledge about cause-and-effect relations between the quality attributes of MTLs and language features or other factors that facilitate or hamper them.

Second, it is unclear which quality attributes are most important and should thus be investigated first.

And third, setting up proper empirical evaluation is challenging (France 2008; Höppner et al. 2022a; Rainer et al. 2021). This is partly caused by the lack of cause-and-effect relations leading to uncertainty about variables to consider. And due to the high amount of effort and uncertainty of results. Thus, empirical evaluation is a high-risk use of limited resources.

The goal of this thesis is to curb all three of these hurdles. We aim to build up detailed data on what quality attributes are associated with MTLs, where advantages and disadvantages in those are presumed, and what factors facilitate them. We further intend to provide information on the most important properties to evaluate. Lastly, we want to demonstrate the feasibility of extensive empirical studies by assessing some of the suggested properties in studies ourselves.

This thesis is structured in two parts. Part one, comprised of Chapter 1, aggregates the results of all accompanying publications. Part two, comprised of Chapters 2 to 6, contains the five publications associated with this thesis.

The remainder of this chapter is structured as follows: Section 1.1 presents an overview of the problem domains model transformation languages and model-driven software development. In Section 1.2 the research objectives and scope of this thesis are presented and discussed. Related work is outlined in Section 1.3. The research methodologies employed for this thesis and the contributions of each individual paper are presented in Sections 1.4 and 1.5, respectively and the implications of our results are discussed in Section 1.6. In Section 1.7 the threats to validity of this thesis are discussed. Lastly, Section 1.8 a conclusion is drawn and future work is outlined.

1.1 Background

This thesis focuses on advantages and disadvantages of model transformation languages which are domain-specific languages used in model-driven (software) engineering. Therefore, this section provides an overview of the model-driven engineering approach, domain-specific languages and model transformation languages.

1.1.1 Model-Driven Engineering

Stachowiak (1973) defines models as "*a representation of entities and relationships in the real world with a certain correspondence for a certain purpose*". Model-driven engineering (MDE) and model-driven software engineering (MDSE) put such models at the centre of development (Brambilla et al. 2017).

There exist several other model-driven approaches, each with a slightly different focus. For this thesis, we will focus on MDE as it encompasses all the other approaches. We use the definition given by Brambilla et al. (2017).

MDE is centred around the concept of automatically generating artefacts from models. Artefacts can be models or different artefacts used in the running system. In MDE models are used both to describe and reason about the problem domain and to develop solutions (Brown et al. 2005). This constitutes an advantage over regular development because the models express domain-related concepts more closely (Selic 2003).

The realisation of a system is spread over three levels (Brambilla et al. 2017). The **modelling level**, where models are defined. The **realisation level**, where the solutions are implemented through artefacts in use within the running system. And the **automation level**, where transformations from models to artefacts are defined. An overview of the relationship between the three levels can be found in Figure 1.1.

In the context of the modelling level, meta-models play a crucial role. They define the structure of models that adhere to them. Each model is written in a modelling language expressed through a meta-model (Bézivin 2004). Meta-models thus define an application domain for which models can be created. The structure of meta-models themselves is defined through meta-models of their own. The Object Management Group (OMG) developed a modelling standard called *Meta-object Facility* (MOF) (OMG 2002) for this purpose. Several modelling frameworks such as the *Eclipse Modelling Framework* (EMF) (Steinberg et al. 2008) and the *.NET Modelling Framework* (Hinkel 2016) have been developed based on MOF.

MDE describes a top-down approach for the automatic generation of executable solutions derived from abstract models (Schmidt 2006; Selic 2003). The (automatic) transformations from one model into other artefacts are called *model transformations* (MTs). Since they connect the modelling level with the realisation level, they constitute the heart of MDE. A fact that is often referred to in literature (Metzger 2005; Sendall et al. 2003). Model transformations can be developed through the use of general-purpose programming languages (GPLs) or through the use of dedicated languages called model transformation languages (MTLs).

1.1.2 Domain-specific Languages

Domain-specific languages (DSLs) are languages focused on one particular aspect (Fowler 2011). They are designed with a notation that is tailored for a specific domain by focusing on relevant features of the domain (Van Deursen et al. 2002). In doing so DSLs provide domain specific language constructs, that let developers feel like working directly with domain concepts. This increases speed and ease of development and even reduces the barrier of entry for non experts to understand what is

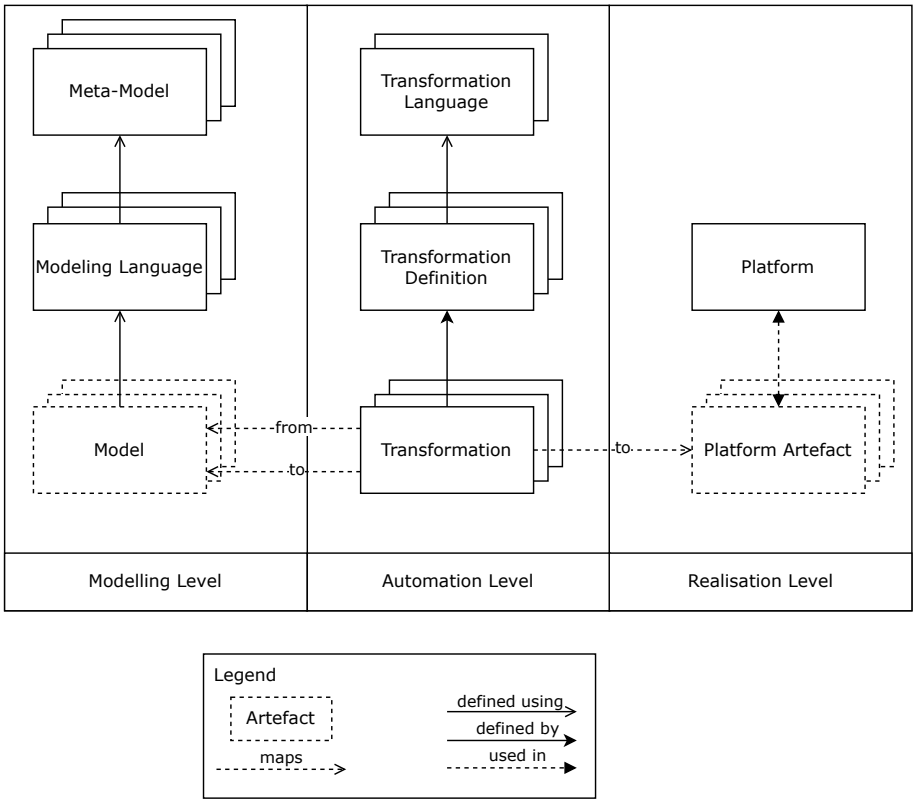


FIGURE 1.1: Overview of MDE based on Brambilla et al. (2017)

written (Fowler 2011; Sprinkle et al. 2009). A well-defined DSL can provide an alternative to using general-purpose tools for problem-solving in a specific domain.

Examples of this include languages such as *shell scripts* in Unix operating systems (Kernighan et al. 1984), HTML (Raggett et al. 1999) for designing web pages or AADL an architecture design language (SAEMobilus 2004).

Fowler (2011) describes DSLs as being either *internal* or *external*. External DSLs are languages that are parsed and often executed separately from the general purpose language in which they are used. Examples of external DSLs are SQL (Codd 1970) and CSS (W3C 2021). Internal DSLs are a form of API within a GPL referred to as fluent interfaces (Fowler 2011). A fluent interface allows an eloquent definition of DSL expressions that can be read much like a standard sentence in a natural language. The integrated query language LINQ (Meijer et al. 2006) in .NET and the declarative Java API to create mock objects JMock (Freeman et al. 2004) fall in this category of DSLs.

1.1.3 Model Transformation Languages[1]

Model transformation languages are DSLs designed to support developers in writing model transformations. For this purpose, they provide explicit language constructs for tasks involved in model transformations such as model matching. There are various features, such as directionality or rule organization (Czarnecki et al. 2006), by which model transformation languages can be distinguished. In this section, we will only be explaining those features relevant to the thesis. Please refer to Czarnecki et al. (2006), Kahani et al. (2019), and Mens et al. (2006) for complete classifications. Table 1.1 provides an overview of the presented features.

1.1.3.1 External and Internal Transformation Languages

As explained in Section 1.1.2, DSLs can be distinguished as internal or external languages. MTLs can be distinguished in the same fashion. They can be embedded into another language, the so-called host language or they can be fully independent languages that come with a compiler or virtual

[1]This section is based on the descriptions from Chapter 3 (Höppner et al. 2022a).

Table 1.1: MTL feature overview

Feature	**Characteristic**	**Representative Language**
Embeddedness	Internal	FunnyQT (Clojure), RubyTL (Ruby), NMF Synchronizations (C#)
	External	ATL, Henshin, QVT
Rules	Explicit Syntax Construct	ATL, Henshin, QVT
	Repurposed Syntax Construct	NMF Synchronizations (Classes), FunnyQT (Macros)
Location Determination	Automatic Traversal	ATL, QVT
	Pattern Matching	Henshin
Directionality	Unidirectional	ATL, QVT-O
	Bidirectional	QVT-R, NMF Synchronisations
Incrementality	Incrementality	NMF Synchronizations
	No incrementality	QVT-O
Tracing	Automatic	ATL, QVT
	Manual	NMF Synchronizations
Dedicated Model Navigation Syntax	Dedicated navigation syntax	ATL (OCL), QVT (OCL), Henshin (implicit in rules)
	No dedicated navigation syntax	NMF Synchronizations, FunnyQT, RubyTL

```java
public void methodExample(Member m) {
  System.out.println(m.getFirstName());
}
public void methodExample2(Member m) {
  Male target = new Male();
  target.setFullName(m.getFirstName() + " Smith");
  REGISTRY.register(target);
}
```

List. 1.1: Example Java methods

machine.

Model transformation languages embedded in a host language are called *internal* MTLs. Prominent representatives among model transformation languages are *FunnyQT* (Horn 2013) a language embedded in Clojure, *NMF Synchronizations* and the *.NET transformation language* (Hinkel et al. 2019a) embedded in C#, and *RubyTL* (Jesús Sánchez Cuadrado et al. 2006) embedded in Ruby.

Fully independent model transformation languages are called *external* MTLs. Examples of external model transformation languages are the *Atlas transformation language* (ATL) (Jouault et al. 2006), one of the most widely known MTLs, the graphical transformation language Henshin (Arendt et al. 2010), as well as a complete model transformation framework called VIATRA (Balogh et al. 2006).

1.1.3.2 Transformation Rules

Czarnecki et al. (2006) describe rules as *"a broad term that describes the smallest units of [a] transformation [definition]"*. Depending on the language, rules can take different forms. Transformation rules in ATL are the rules that make up transformation modules. In other languages, they can take the form of a function or method that implements a transformation from an input element to an output element.

The fundamental difference between model transformation languages and general-purpose languages that originates in this definition lies in dedicated constructs that represent rules. In GPLs, there is no clear-cut difference between a transformation rule and any other function, method or procedure. This distinction can only be made based on the contents thereof. An example of this can be seen in Listing 1.1, which contains exemplary Java methods. Without detailed inspection of the two methods, it is not apparent which method does some form of transformation and which does not. Using descriptive names for methods can help alleviate the problem, but it does not prevent transformation functionality from being written in methods that are not intended for it.

In MTLs, transformation rules are dedicated constructs within the languages that allow a definition of a *mapping* between input and output (elements). The example rules written in the model transformation language ATL in Listing 1.2 make this apparent. They define mappings between model elements of type `Member` and model elements of type `Male` as well as between `Member` and `Female` using *rules*, a dedicated language construct for defining transformation mappings. The transformation is a modified version of the well known Families2Persons transformation case (Anjorin et al. 2017).

1.1.3.3 Rule Application Control: Location Determination

Location determination describes the strategy that is applied for determining the elements within a model onto which a transformation rule should be applied (Czarnecki et al. 2006). Most model transformation languages such as ATL, Henshin, VIATRA or QVT (OMG 2016), rely on some form of *automatic traversal* strategy to determine where to apply rules.

We differentiate two forms of location determination based on the kind of matching that occurs during traversal: *basic automatic traversal* and *pattern matching*. In *basic automatic traversal*, the rule applied to an element is determined based on the model element and constraints on it. This type of location determination is employed in languages such as ATL or QVT. In *pattern matching*, a model- or graph-*pattern*, defined in a rule, is matched within the model. This allows developers to define sub-graphs consisting of several model elements and references between them, which are then manipulated by a rule. Pattern matching is used, e.g., in Henshin.

```
rule Member2Male {
  from
    s : Member (not s.isFemale())
  to
    t : Male (
      fullName <- s.firstName + ' Smith'
    )
}

rule Member2Female {
  from
    s : Member (s.isFemale())
  to
    t : Female (
      fullName = s.firstName + ' Smith'
      partner = s.companion
    )
}
```

List. 1.2: Example ATL rules

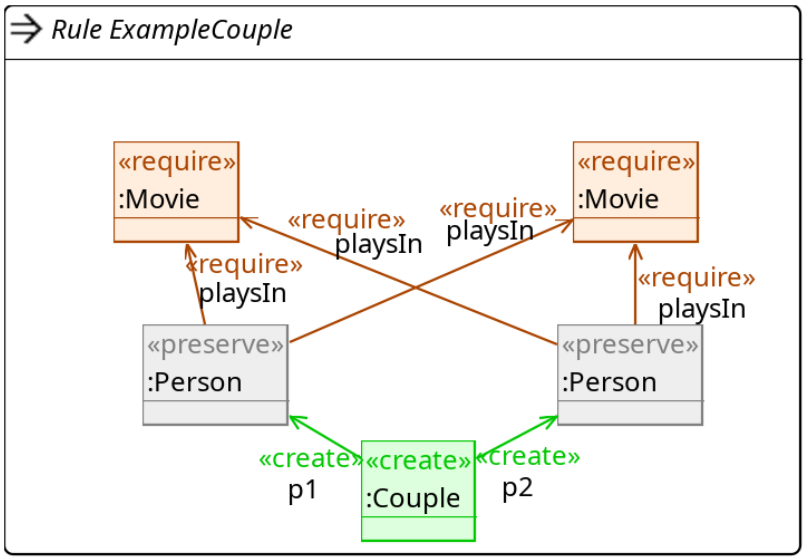


Figure 1.2: Example Henshin transformation

The *basic automatic traversal* of ATL applied to the example from Listing 1.2 will result in the transformation engine automatically executing the `Member2Male` on all model elements of type `Member` where the function `isFemale()` returns `false` and the `Member2Female` on all other model elements of type `Member`.

The *pattern matching* of Henshin can be demonstrated using Figure 1.2, a modified version of the transformation examples by Krause et al. (2014). It describes a transformation that creates a couple connection between two actors that play in two films together. When the transformation is executed the transformation engine will try and find instances of the defined graph pattern and apply the changes to the found matches.

The examples highlight the main difference between *automatic traversal* and *pattern matching*. The engine will search for a sub-graph within the model instead of applying a rule to single elements within the model.

```
top relation Member2Male {
  n, fullName : String;
  domain Families s:Member {
    firstName = n };
  domain Persons t:Male {
    fullName = fullName};
  where {
    fullName = n + ' Smith'; };
}
```

List. 1.3: Example QVT-R relation

1.1.3.4 Directionality

The directionality of a model transformation describes whether it can be executed in one direction, called a unidirectional transformation, or in multiple directions, called a multidirectional transformation (Czarnecki et al. 2006).

For this thesis, the distinction between unidirectional and bidirectional transformations is relevant. Languages that allow dedicated support for executing a transformation both ways based on one transformation definition are called bidirectional. Those that require users to define transformation rules for both directions are called unidirectional. General-purpose languages can not provide bidirectional support and always require both directions to be implemented explicitly.

The ATL transformation from Listing 1.2 defines a unidirectional transformation. Input and output are defined and the transformation can only be executed in that direction.

The QVT-R relation defined in Listing 1.3 is an example of a bidirectional transformation definition (for simplicity reasons, the transformation omits the condition that males are only created from members that are not female). Instead of a declaration of input and output, it defines how two elements from different domains relate to one another. As a result, given a `Member` element its corresponding `Male` elements can be inferred, and vice versa.

1.1.3.5 Incrementality

Incrementality of a transformation describes whether existing models can be updated based on changes in the source models without rerunning the complete transformation (Czarnecki et al. 2006). This feature is sometimes also called model synchronisation.

Providing incrementality for transformations requires (active) monitoring of input and/or output models to detect changes therein. Additionally information on which rules affect what parts of the models is also needed. When a change is detected, the corresponding rules can then be executed. It can also require additional management tasks to keep models valid and consistent.

1.1.3.6 Tracing

According to Czarnecki et al. (2006) tracing *"is concerned with the mechanisms for recording different aspects of transformation execution, such as creating and maintaining trace links between source and target model elements"*.

Several model transformation languages, such as ATL and QVT, have automated mechanisms for trace management. This means that traces are automatically created during runtime. Some trace information can be accessed through special syntax constructs. At the same time, some of it is automatically resolved to provide seamless access to the target elements based on their sources.

An example of tracing in action can be seen in `line 16` of Listing 1.2. Here the `partner` attribute of a `Female` element that is being created, is assigned to `s.companion`. The `s.companion` reference points towards a element of type `Member` within the input model. When creating a `Female` or `Male` element from a `Member` element, the ATL engine will resolve this reference into the corresponding element, that was created from the referred `Member` element via either the `Member2Male` or `Member2Female` rule. ATL achieves this by automatically tracing which target model elements are created from which source model elements.

1.1.3.7 Dedicated Model Navigation Syntax

Languages or syntax constructs for navigating models are not part of any feature classification for model transformation languages. However, it is a relevant distinction for this thesis.

Languages such as OCL (OMG 2014), which is used in transformation languages like ATL, provide dedicated syntax for querying and navigating models. They provide syntactical constructs that aid users in navigation tasks. Different model transformation languages provide different syntax for this purpose. The aim is to provide specific syntax, so users do not have to manually implement queries using loops or other general purpose constructs. OCL provides a functional approach for accumulating and querying data based on collections, while Henshin uses graph patterns to express the relationship of sought-after model elements.

1.2 Goals and Scope

As described in the introduction of this chapter, the objective of this Ph.D. thesis is threefold. We want to create a reliable source of data about advantages and disadvantages of MTLs and where they originate from, provide clear suggestions on the most important aspects to investigate and add to the current body of empirical research by investigating some of the suggested aspects ourselves. Each of these goals is addressed through research questions described in this section.

G1: Build up detailed data on what quality attributes are associated with MTLs.

- **RQ1:** What are quality attributes claimed to be associated with MTLs, and is the association positive or negative?
- **RQ2:** What empirical studies or other means are used to validate claimed advantages and disadvantages?

G2: Provide a model of factors that facilitate or hamper the perception of quality attributes and information on the importance of the factors.

- **RQ3:** What are the factors that influence the association of quality attributes with MTLs?
- **RQ4:** How do the identified factors influence MTL quality attributes?
- **RQ5:** How important are the influences of factors on the perceived MTL quality attributes?
- **RQ6:** What is the relevancy of the identified quality attributes and factors for language developers, researchers & transformation developers?

G3: Demonstrate the feasibility of extensive empirical studies and provide first real results through assessing some of the suggested properties

- **RQ7:** How do domain specific syntax constructs influence the expressiveness of ATL for developing model transformations?
- **RQ8:** How does the complexity of transformations written in ATL compare to those written in Java?

G1 & **G2** aim to form a basis to build empirical studies on and to further research on model transformation languages in general. In detail, **RQ1** addresses the need for explicit and precise information in the form of a list of quality attributes with a detailed description thereof. It also intents to establish a characterisation of what specific advantages and disadvantages are associated with quality attributes of MTLs. **RQ2** aims to establish the current state of validation of all claims on model transformation languages that are identified in **RQ1**. The results serve the purpose of providing further justification for this work and information on used methodologies and investigated properties. **RQ3** aims to extend the data basis by eliciting the underlying factors that lead people to associate advantages and disadvantages with MTLs. **RQ4** looks to complete a qualitative picture through details on how factors facilitate these influences. **RQ5** aims to extend the qualitative model with quantitative data on the influences between the identified factors and the perceived quality of MTLs. These numbers serve as the basis to evaluate what the most important quality attributes and factors are for **RQ6** with the aim to prioritise evaluation and further language development.

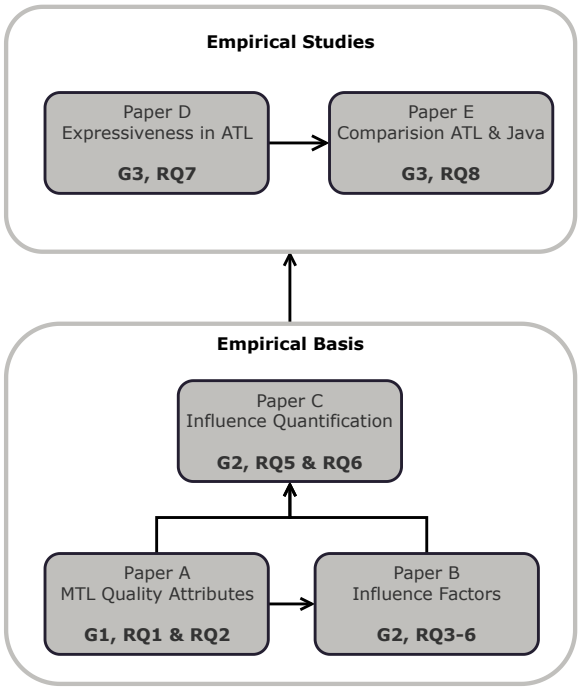


FIGURE 1.3: Ph.D. scope

Lastly, **G3** aims to add empirical evidence to some of the most prevalent aspects and demonstrate the feasibility of conducting such studies. For this purpose we focus on one specific model transformation language and one general purpose language and investigate one of the quality attributes identified in **RQ1**. The results of **RQ3-5** provide information about the factors that need to be taken into account when answering the research questions for this goal. **RQ7** aims to establish first quantitative results on the effects of the domain specific syntax constructs present in the model transformation language ATL. The results serve as a basis for comparing ATL with Java transformations in **RQ8**. **RQ8** aims to investigate how much manual effort is required to implement the domain specific functionality and how this effects the code structure. We are especially interested in how much more code has to be written and how much this changes the focus of the code from developing a transformation to writing code to make the transformation work. **RQ7** and **RQ8** are intended to be answered through several additional sub-research questions that define concrete and measurable variables to investigate.

The goals of this thesis are addressed through the five papers included in this thesis. Figure 1.3 depicts which goals and research questions are addressed by each individual paper. Papers A, B and C jointly form the basis for the empirical studies conducted in Papers D and E by addressing goals **G1** & **G2**. For this, Paper A answers research questions **RQ1** & **RQ2**. The results are used in Paper B to answer **RQ3** & **RQ4**. Additionally, due to the nature of the study conducted in Paper B, we can also infer first answers for **RQ5** & **RQ6**. Paper C uses all these results to address goal **G2** with answers to **RQ5** & **RQ6**.

The results from goals **G1** & **G2** are then used to design the studies in Paper D and E. Paper D answers **RQ7** through a case study focused on ATL transformations. In Paper E we extend this investigation with a comparison between ATL and Java transformations to answer **RQ8**.

1.3 Related Work

A broad body of literature can be related to the papers of this thesis individually and the goals in general. In this section, we give an overview of works related to the overall scope of this thesis. There are four main bodies of related work. First, works on classifying model transformation languages. Second, studies on domain specific languages. Third, empirical studies on model driven engineering. Fourth, empirical studies focused on model transformation languages. A detailed discussion of work related to the studies included in this thesis is given in each paper individually.

1.3.1 Classifications of Model Transformation Languages

There exist no prior works that contribute to a systematisation of what quality attributes are associated with model transformation languages. However there are several works that provide classification of model transformations (Czarnecki et al. 2006; Kahani et al. 2019; Mens et al. 2006) which we use as a basis to close the existing gap. Their classifications systematise functionality and language makeup. Our classification on the other hand focuses on non-functional quality properties that are associated with model transformation languages.

1.3.2 Studies on Domain Specific Languages

In the area of domain specific languages several classifying works exist. Tomaž Kosar et al. (2016) detail a mapping study, highlighting trends in DSL research. Similar to the results of Paper A they also identified a lack of empirical studies that compare DSLs with general purpose languages or investigate properties of DSLs in depth. A discussion of DSL terminology and risks and benefits thereof is presented by Van Deursen et al. (2000). Many of the points raised by Van Deursen et al., such as claims about the conciseness of DSLs or better performance, are also reflected in our results on model transformation languages from Paper A. Expanding on the discussion of using general purpose languages as alternatives to DSLs Tomaz Kosar et al. (2010) present an empirical study comparing C# Forms and XAML for the purpose of defining user interfaces. The study represents one of the few attempts in comparing DSLs and general purpose languages. Albeit being focused on user interface design it demonstrates the importance of such comparative studies as their results provide quantitative data that can be used to argue the use of a DSL over a general-purpose language for specific tasks.

1.3.3 Studies on Model Driven Software Engineering

More closely related to model transformation languages, several empirical studies on MDSE have been conducted and reported on. Staron (2006) conducted interviews with experts from two companies planning to adopt MDSE to find the state-of-practice in adopting the approach in industry. Their main finding was that the existing tooling and processes were not up to the task of basing development solely around models. A tooling gap specifically for model transformation languages was also identified in our study reported in Paper B.

Similarly, using questionnaires and interviews (Hutchinson et al. 2011a,b, 2014; Whittle et al. 2013), Whittle, Hutchinson, Rouncefiled et al. elicited positive and negative consequences of applying MDSE in industry and factors that drive or hamper its adoption. Their findings include organisational benefits in communication and flexibility to changing requirements. Productivity gains far beyond what they expected were also observed. Relevant factors for the success of MDSE were mainly centred around organisational tasks such as applying MDSE to the right use-cases and commitment to enact changes required in the process. In Paper B, we also identified the use-case as one of the most important aspects when comparing model transformation languages with GPLs. Whittle, Hutchinson, Rouncefiled et al. also point toward several open challenges for research that are similar to our findings. This includes opening communities to educate people on MDSE and its applications and focusing on improving the most relevant processes within MDSE rather than just developing more tools.

Mohagheghi et al. carried out multiple empirical studies on MDSE, focusing on factors for and consequences of adoption thereof. They used surveys and interviews at several companies (Mohagheghi et al. 2013a,b) as well as a literature review (Mohagheghi et al. 2008) for this purpose. They found MDSE to not be suited for small projects as it required tool customisation and integrating them into existing tool chains. Tasks that are labour intensive and do not pay off for small projects. Similar to our results in Paper A, but for MDSE in general, they too identified a lack of existing empirical studies.

1.3.4 Studies on Model Transformation Languages

There exist several empirical studies on the topic of model transformation languages dealing with different aspects thereof. Some focus on the social aspects involved in writing model transformations (Groner et al. 2020; Tehrani et al. 2016). Others compare model transformation languages with each other (Jakumeit et al. 2014).

Tehrani et al. (2016) use an interview study with five individuals to understand the context in which model transformations are developed when using MTLs and identify differences compared to GPL transformation development. An interesting finding of theirs is that all transformation projects discussed in their study were greenfield projects. This is in line with our findings in Paper B that embedding transformations written in MTLs is difficult. Additionally the development of transformations seems to lack systematic processes which we also identified as a gap in the state of the art.

Groner et al. (2020) report on a mixed method study to investigate whether transformation developers are concerned with the performance of their transformations and what strategies they employ to identify performance issues. Their results show that 42% of developers are only sometimes satisfied with their transformations performance which stands in contrast with much of literature that claims performance benefits of MTLs as identified in Paper A.

Jakumeit et al. (2014) report on an in-depth qualitative comparison of different model transformation languages based on the Transformation Tool Contest (TTC). They compare all solution submissions for a transformation case submitted for the contest in 2011 and highlight the process how the case was solved using the different languages. The study is a rare case of an in-depth report on how to use model transformation languages and thus provides a valuable resource for people when deciding on what language to use for development. Unfortunately the report is over 10 years old and thus suffers from the same over-ageing that we identified for many of the claims surrounding model transformation languages in Paper A.

Several studies follow a similar methodology as we use in the empirical studies included in this thesis. Di Rocco et al. (2015) analyse the impact of input and output meta-models on a number of metrics calculated for ATL transformations. For this purpose, they use repository mining to collect 91 ATL transformations and the meta-models associated with those transformations. We employ this approach in Papers D and E with a different aim. While they focus on identifying the impact of the meta-models on things like the size of ATL transformations, our studies are more introspective. We analyse and compare transformation scripts written in ATL and Java, drawing direct conclusions on the advantages and disadvantages of using ATL compared to Java for transformation development.

Similarly Angelika Kusel et al. (2013) analysed the ATL Zoo[2] with the goal to gain insights about the frequency of use of reuse mechanisms. Their results are highly important as we identified in Paper C, that the use of reuse mechanisms is a major factor for the perception of quality attributes of MTLs. They also highlight the current challenges of reuse mechanisms in model transformation languages, specifically ATL, because it requires complex abstraction and specialization mechanisms that are hard for transformation developers to utilise.

Tolosa et al. (2011) calculate several metrics for 9 different ATL transformations. However their main purpose was to demonstrate the feasibility of their metrics calculation transformation. Since they have not published any follow up studies that utilise their metrics the study is another example of the lack of thorough empirical rigour in the area of MTL evaluation.

Only one empirical study, a controlled experiment by Hebig et al. (2018), focuses on comparing model transformation languages with general purpose languages. They let 78 subjects solve three different model transformation tasks (comprehending, changing and creating) using ATL, QVT-O and Xtend. Unfortunately, they could not find statistically significant data on their main hypotheses about subjects performing better in the tasks using MTLs than GPLs. They did however find that MTLs provide better support to users for identifying context for changes and that subjects were much better in creating conditioning via types than via values. We were able to support several of their qualitative observations and identify which parts of ATL facilitate them in our study from Paper E using a different methodology.

Lastly, there exist several experiment templates for evaluating model transformation languages against each other and GPLs that have yet to be executed (Kramer et al. 2016; Strüber et al. 2016).

1.4 Research Methodology

This thesis aims to curb all three hurdles hampering the empirical evaluation of model transformation languages. To this end, we employ several empirical studies ourselves. The applied methodologies are tuned to the respective research questions answered in the studies. Papers A, B and C aim to collect empirical data on the advantages and disadvantages of MTLs and the influences that

[2] `https://www.eclipse.org/atl/atlTransformations/`

Table 1.2: Research Methodology per Paper

Paper	Strategy	Data
Paper A	Structured Literature Survey	58 Publications
Paper B	Interview Study	56 Participants
Paper C	Questionnaire & Universal Structure Modelling	113 Participants
Paper D	Repository Mining	33 Transformations
Paper E	Repository Mining & Design Science	3x12 Transformations

facilitate them. Papers A and B use qualitative methods while Paper C is a quantitative study. The results of these studies were then used to decide on the focus of the two case studies in Papers D and E. Table 1.2 summarises the empirical strategies employed and the data used in all five studies.

At the beginning of this work, in Paper A, we collected data on the current state of literature using a structured literature review (SLR) (Kitchenham et al. 2007). We surveyed literature to determine what quality aspects were associated with MTLs and whether the association is positive, i.e., an advantage, or negative, i.e., a disadvantage. Moreover, it allowed us to identify how much of the claimed advantages and disadvantages are backed up by evidence and what type of evidence was used. Structured literature reviews are well suited for this purpose, as they allow for mapping the landscape of current research and identifying gaps in said research (Boot et al. 2016).

In Paper B, these insights were contextualised and expanded upon in an interview study. We collected and analysed reasons and background information that led participants to believe claims about MTLs to be true. Interviews are an appropriate approach for this goal because they allow for ascertaining qualitative data such as opinions and estimates and are one of the most widely used research methods in the technical field for this purpose (Hove et al. 2005; Meyer et al. 1990). Due to the number of quality attributes identified in Paper A, Paper B only focuses on a subset thereof.

Paper C seamlessly follows and adds quantification to the findings of Paper B. We aimed at quantifying how strong the influence of language capabilities and user background are on their perception of quality attributes of MTLs. In this study, the focus was again narrowed to keep the size of the study in check. For the purpose of the study, we use a survey that follows the seven-step process outlined by Kasunic (2005) and utilises universal structure modelling (USM) (Buckler et al. 2008) for analysis. Survey research was chosen because "*it provides a means to distil the subjective (and often fuzzy) opinions of the respondents*" (Torchiano et al. 2017). Using USM to analyse the responses to the questionnaire has several reasons. The hypothesised relationships between quality attributes, user background and language capabilities define a complex structure model. Such models are generally investigated by means of structure equation modelling (Weiber et al. 2021). USM is a methodology for structure equation modelling that is able to handle uncertainty about the completeness of the structure model better than alternative approaches (Buckler et al. 2008). Moreover, it provides more potent capabilities to analyse moderation effects and non linear correlations between the analysed factors (Weiber et al. 2021).

Papers D and E use repository mining to investigate some of the relationships revealed in Papers A-C empirically. Repository mining is an approach where data collected from a platform hosting large numbers of structured or semi-structured text is analysed and cross-linked with the intent of finding interesting and actionable information about the data (Hassan 2008). In contrast to case studies (Runeson et al. 2012), repository mining does not investigate the dataset within its real-world context. Instead, it is concerned with using a larger amount of data to draw quantitative conclusions. We employ repository mining because more data is readily available, and the quantitative results are more robust in their generalisability.

Paper D is focused on the expressiveness of ATL for writing model transformations with a special focus on the effects of automatic trace handling, model traversal and dedicated model navigation syntax. We collected 33 transformations and analysed them qualitatively and quantitatively for what code is required for which aspect of the transformation development. This data is used to assess the code distribution over the different aspects quantitatively. The measure used for quantitative analysis is called syntactic complexity, a measure based on the collection of ATL measures by Lano et al. (2018).

In Paper E repository mining is complemented with design science. Design science is concerned with iterative creation and evaluation of new artefacts (Hevner et al. 2004). We use design science to create a translation schema used to generate additional artefacts based on those extracted through

repository mining. This complementary method was chosen, because no suitable Java artefacts were readily available through repository mining alone.

The focus of Paper E is to investigate how transformations written in a general-purpose language, i.e., Java compare to those written in ATL in legacy and modern Java styles. Because no transformations written in Java were readily available, we used design science to create them based on existing ATL transformations. All transformations were analysed and compared in terms of code distribution over the different transformation aspects. For this purpose we used the syntactic complexity measure for ATL and word count for Java. The word count measure counts the number of words that are separated either by whitespaces or other delimiters used in the languages, such as a dot (`.`) and different kinds of parentheses (`()[]{}`). It has already been successfully used by Anjorin et al. (2019) to compare several (bidirectional) transformation languages. Moreover, the Java transformations in the different programming styles were compared on in terms of lines of code, McCabe complexity (McCabe 1976) and word count.

A more detailed exposition on the methodologies used in each paper can be found in Chapters 2 to 6.

1.5 Contribution

In this section, we briefly outline all five papers included this thesis and describe their contribution to the state of research. The complete papers can be found in Chapters 2 to 6.

1.5.1 Paper A: Claims and Evidence in Literature

As mentioned in Chapter 1, model transformation languages are developed to improve the transformation development process. A substantial amount of effort is put into developing new languages and new concepts to incorporate into them. The advancements made through new developments are then casually related to new or improved advantages of MTLs compared to prior iterations and general-purpose languages. However, as we show in Paper A, there is a lack of evidence for claims raised in literature. Moreover, there is no uniform definition of the quality attributes generally associated with model transformation languages and no central repository of knowledge about the state of evidence. While these gaps can be considered convenience problems, they pose a problem for those trying to convince people outside of the MDE community of the usefulness of MTLs.

The contribution of Paper A is a comprehensive overview of the quality attributes associated with model transformation languages and state of evidence thereof throughout literature. Paper A aims to answer **RQ1** & **RQ2**:

> **RQ1:** What are quality attributes claimed to be associated with MTLs, and is the association positive or negative?
>
> **RQ2:** What empirical studies or other means are used to validate claimed advantages and disadvantages?

The paper lays the foundation for this thesis by providing a set of 15 quality attributes associated with MTLs in literature. It also provides further motivation for our work by highlighting how grave the state of empirical research, or any evidence, for MTLs is. Lastly, it also highlights how a casual way of handling claims leads to misrepresentation of the state of evidence, which crumbles under a small amount of scrutiny. This makes the community look inaccurate at best and dishonest at worst.

Paper A does not focus on a specific type of model transformation language or feature set. Instead it aims to provide a complete overview for the entirety of MTLs.

Using 4 literature databases and the search terms *MDE*, *model transformation* and *model transformation languages* (and synonyms thereof), as well as snowballing, we collected over 4000 publications. The publications were filtered for mentions of advantages or disadvantages of model transformation languages by two separate researchers. Different assessments were discussed. In the end 58 papers were selected for further analysis. The selected papers were analysed for quality attributes mentioned as well as advantages and disadvantages associated with the attributes. Finally, we looked for evidence the papers provided that support the raised connections. To do so, we employed initial coding and focused coding as described by Charmaz (2014), first extracting common phrases and then using them to form categories.

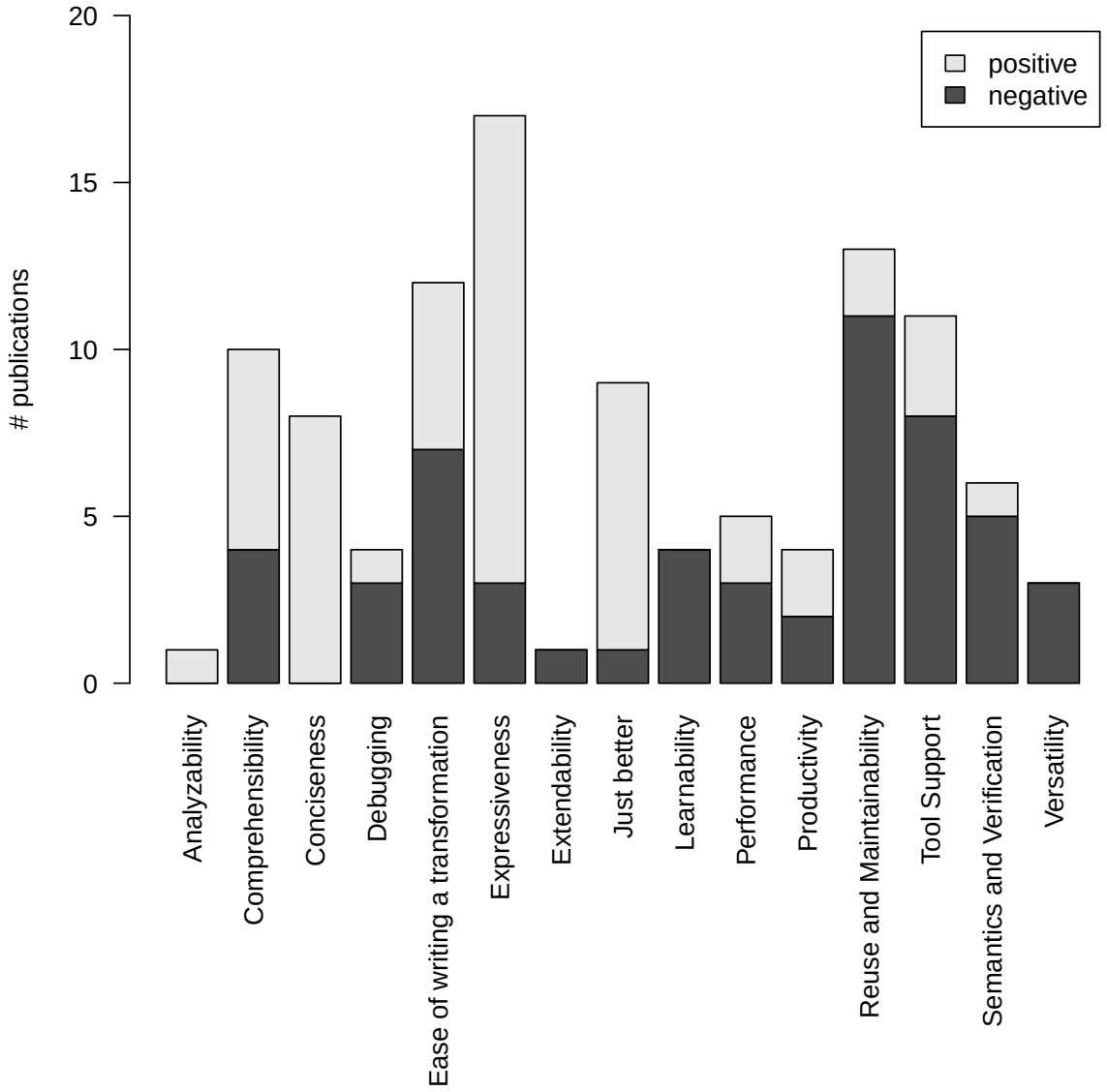


FIGURE 1.4: Quality attributes associated with MTLs from Paper A (Götz et al. 2021a)

The findings of Paper A answer **RQ1** as follows. Literature associates MTLs with 15 different quality attributes shown in Figure 1.4. Most quality attributes are associated positively and negatively with MTLs highlighting the intricate balance between useful abstractions and drawbacks thereof. Moreover, we believe that a portion of this variance can be attributed to the diverse range of languages, each with their own unique trade-offs between accentuating capabilities and accepting drawbacks. Outliers to this are *Analysability*, *Conciseness*, *Extendability* and *Versatility*.

These results are contrasted by a bleak state of evidence for claims made about the quality attributes as analysed for **RQ2**. We found four different means used to support claims. First, empirical studies. Regrettably they constitute the smallest amount of presented evidence. Second, references to other scientific literature. Third, examples that are used to demonstrate a claim. And fourth, no evidence.

More than 70% of all claims found in our literature review lack any form of substantiation. Much of the evidence that is given stems from examples that demonstrate the claimed advantage or disadvantage in a single case. We were only able to identify four studies that use empirical methods. Moreover, only one of these studies focused on model transformation languages as the central artefact of evaluation.

Another concerning observation is, that citations are often used to reference other works that

make similar claims rather than literature that provides proper substantiation. We attribute this to the fact that there is no evidence to support most of the claims. However, regardless of the reasons, the practice distorts the perception of evidence. It suggests that statements are supported by evidence even though the only available source of information is the opinions of other users and researchers.

Overall, Paper A shows that there is an enormous amount of claims, both positive and negative, made about model transformation languages. The state of evidence for claimed advantages and disadvantages is, however, lacking. Moreover, the practice of citing other literature making similar claims leads to a twisted image of reality in which there is more certainty about the advantages and disadvantages than what corresponds with reality. There is also no clear information on what functional properties of MTLs give them advantages or disadvantages for different quality attributes. Paper A thus creates motivation and a starting data set for further, in-depth studies into transformation languages and their associated quality attributes.

An overview of Paper A's contribution is shown in Figure 1.5.

RQ1	**RQ2**
• Collection of 137 claims on the advantages and disadvantages of MTLs • List of 15 quality attributes associated with model transformation languages • Categorisation of what quality attributes are raised in the 137 claims • Discussion of similarities and dissimilarities among claims for each quality attribute	• Overview of types of evidence used in literature to support claims on MTLs • Classification of the type of evidence used to support each of the 137 claims • Quantitative analysis of each type of evidence • Qualitative analysis of citations, used as evidence, in literature • Suggestions on how to improve the state of evidence

Figure 1.5: Contribution of Paper A

1.5.2 Paper B: Factors for Advantages and Disadvantages

Based on the results from Paper A, Paper B explores the relationship between model transformation languages and their quality attributes in more depth. Specifically we aim to find out what factors influence the association of quality attributes with MTLs (**RQ3**) and the nature of the influence (**RQ4**).

> **RQ3:** What are the factors that influence the association of quality attributes with MTLs?
>
> **RQ4:** How do the identified factors influence MTL quality attributes?

The results of this study are important for further empirical studies on model transformation languages because they provide explicit knowledge on the variables that need to be considered in such studies.

We employ a semi-structured interview study, because the association between MTLs and their quality attributes stems from the impression of users and developers. The qualitative nature of the study also allows for first insights for **RQ5** & **RQ6**.

> **RQ5:** How important are the influences of factors on the perceived MTL quality attributes?
>
> **RQ6:** What is the relevancy of the identified quality attributes and factors for language developers, researchers & transformation developers?

The study focuses on the quality attributes from the results of Paper A. We narrowed the set of investigated quality attributes down to 6 attributes, because of the large effort involved in conducting interviews. The 6 investigated attributes are: *Comprehensibility*, *Ease of Writing*,

Table 1.3: Overview of influence factors

Top-level Factor	**Sub-Factor**
GPL Capabilities	
MTL Capabilities	Domain Focus Bidirectionality Incrementality Mappings Traceability Model Traversal Pattern Matching Model Navigation Model Management Reuse Mechanisms Learnability
Tooling	Analysis Tooling Code Repositories Debugging Tooling Ecosystem IDE Tooling Interoperability Tooling Awareness Tool Creation Effort Tool Learnability Tool Usability Tool Maturity Validation Tooling
Choice of MTL	
Skills	Language Skills User Experience/Knowledge
Use Case	(Meta-) Models I/O Semantic gap Size

practical Expressiveness, *Productivity*, *Reuse and Maintainability* and *Tool Support*. They were chosen because of the large number of claims about them in literature and because we ascribe them a relevant role for the adoption of MTLs.

We held interviews with 55 participants and collected one written response. The interviews were transcribed and the transcripts were analysed using *content structuring content analysis* (Kuckartz 2014).

As answer to **RQ3**, we identified six top-level factors, namely *GPL Capabilities*, *MTL Capabilities*, *Tooling*, *Choice of Language*, *Skills* and *Use Case*. Each top-level factor comprises several sub-factors. An overview of the top-level factors and associated sub-factors can be found in Table 1.3.

The influences identified to answer **RQ4** are split into two groups, direct influences and context influences. We found that *GPL Capabilities*, *MTL Capabilities* and *Tooling* have a direct influence on perceived quality attributes of MTLs and *Choice of Language*, *Skills* and *Use Case* define context that moderates the type and strength of influence of the other factors. A graphical depiction of these relationships can be found in Figure 1.6.

Central to the abstract model of influences are the advantages and disadvantages of model transformation languages. The effectiveness of MTLs depends on their capabilities, which can vary based on the language in question and whether the abstractions provided are useful for the specific case, i.e., bidirectional support is only useful for bidirectional transformation cases. Additionally, the developer's skill plays a crucial role too. If they are unable to effectively utilise the capabilities of a language due to inexperience or lack of knowledge, none of the advantages might apply. Moreover,

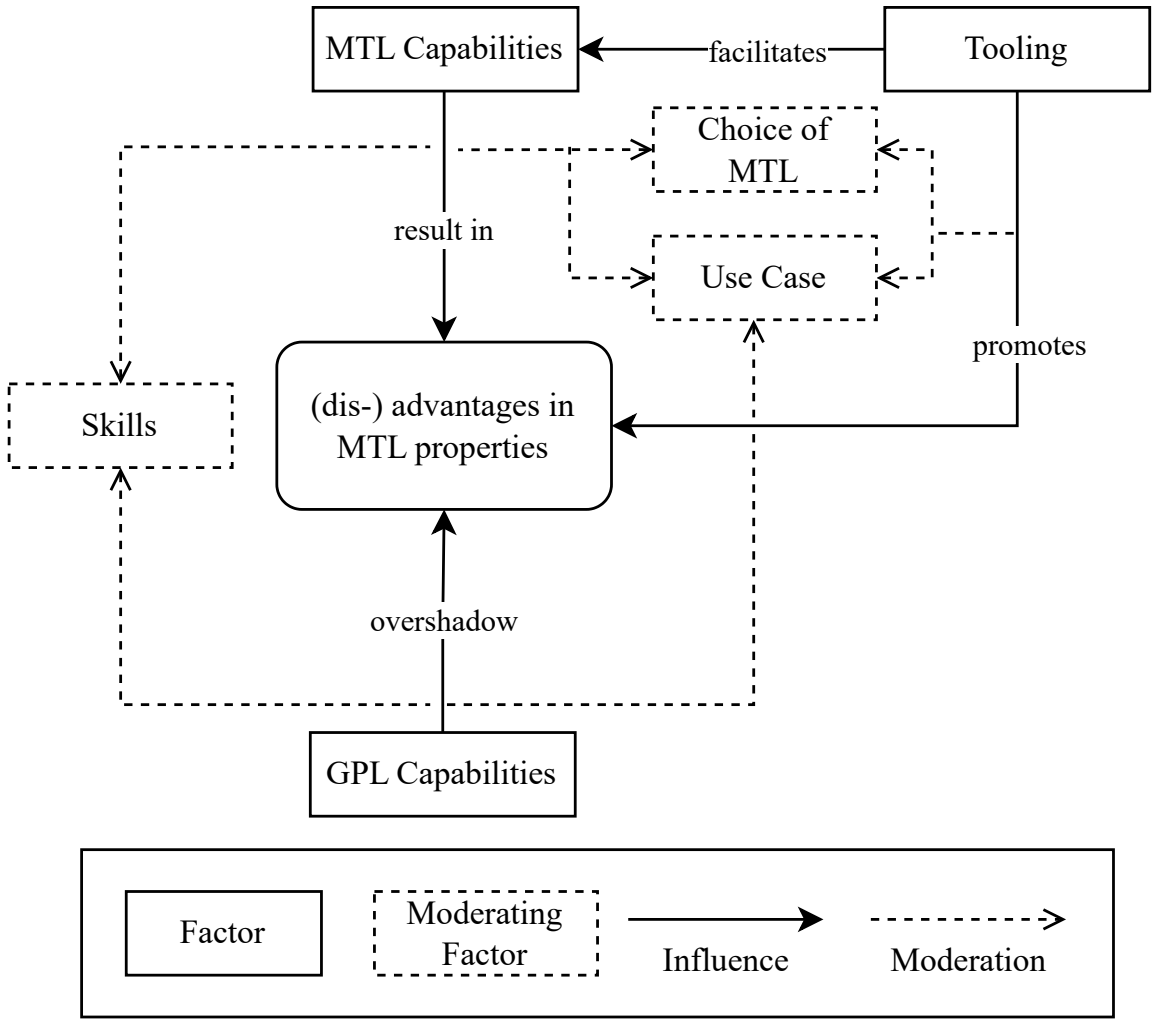


Figure 1.6: Graphical overview over factor influences and moderations adapted from Paper B (Höppner et al. 2022a)

general-purpose constructs in GPLs can overshadow any perceived advantage or disadvantage of an MTL. These constructs may be just as easy to use. On the other hand, lack of support in a GPL can make MTLs more attractive, as labour intensive manual implementations might be required. Finally, the tooling available for MTLs can either enhance or impede their capabilities. This again depends on the language and use case as tooling may or may not exist. A detailed description and discussion of the influences of all factors and sub-factors is also given.

We were also able to gain qualitative data for answers to research questions **RQ5** & **RQ6** based on the responses from the interview participants. While different participants talked about different language features in MTLs and GPLs, one common subject was raised. Most participants mentioned that the use case is extremely important in determining whether MTLs have advantages or disadvantages. There was also a lot of agreement that tooling needs to be expanded significantly. This universal agreement suggests to us that in addition to language development, focus should be on investigating precisely in which cases and how MTLs should best be applied. A big gap, we identified, is the embedding of MTLs in legacy systems. Participants frequently mentioned that MTLs are difficult to integrate into existing systems and processes and are therefore mainly suitable for greenfield projects.

We provide 15 suggestions for future work for language developers and researchers. For researchers, suggestions include focused empirical evaluation of MTL properties and features based on the influences we identified. For language developers, suggestions include the aforementioned legacy integration and the development of transformation-specific reuse mechanisms. In addition, we recommend more work in knowledge transfer. This includes the suggestions to improve the teaching

of the MDSE paradigm and better tutorials for MTLs. Furthermore, more industry collaborations for language improvement and evaluation should be aspired.

Paper B highlights the complexity and size of the topic of this thesis. Its intricate results also form the basis for much of our empirical studies. Three suggested future works, *influence quantification*, *empirical investigation of MTL quality attributes* and *empirical factor evaluation*, are tackled in Papers C, D and E. Paper C provides quantification of the influence weights of the factors, and Papers D and E are empirical studies to investigate MTL quality attributes based on the knowledge about which factors influence what attributes.

Figure 1.7 provides an overview of Paper B's contribution.

RQ3	**RQ4**
• Collection of 30 factors influencing the perception of MTL quality attributes • Categorisation of factors into 6 overarching top-level factors	• Differentiation of factors into direct and moderating influencing factors • Structure model describing which factors influence which quality attributes and which factors moderate these influences • Discussion of the implications resulting from the influence of each factor

RQ5 & RQ6
• Qualitative assessment which of the identified factors have the highest practical impact • Qualitative assessment which of the identified factors have the highest relevance for empirical studies • Suggestions on 15 actionable results, focusing on different identified factors, for researcher and language developers

Figure 1.7: Contribution of Paper B

1.5.3 Paper C: Quantification of Influence Weights of Factors

The structure model from Paper B that depicts assumed influences between factors and quality attributes is missing quantification of the influence strengths. These are important for further empirical studies to be able to decide what variables to measure and control. It also provides indication of what factors to focus on in other work as stronger influence suggests more relevance for both developers and users. Paper C aims to provide the quantification of the structure model and by answering **RQ5** & **RQ6**.

> **RQ5:** How important are the influences of factors on the perceived MTL quality attributes?
>
> **RQ6:** What is the relevancy of the identified quality attributes and factors for language developers, researchers & transformation developers?

A natural progression of the structure model from Paper B is to use further structural equation modelling methods to add quantitative results to it. The study reported in Paper C aims to do this. In this study, too, the focus was narrowed to keep the size of the study in check. We use a survey for this purpose that follows the seven-step process outlined by Kasunic (2005) and utilises universal structure modelling (USM) (Buckler et al. 2008) for analysis. The study focuses solely on the impact of MTL Capabilities and moderating factors. The survey was pilot tested and sent to about 2500 potential participants. The responses from 113 participants were recorded and analysed using USM to refine the structure model and answer **RQ5** & **RQ6**. The methodology used for this paper is published as a registered report at ESEM'22 (Höppner et al. 2022b). The complete paper

is under review for a special issue of EMSE as a progression from the registered reports track of ESEM'22.

To answer **RQ5**, we collected the *average simulated effect* and *overall explained absolute deviation* of all influences. Overall we found that *Traceability* and *Reuse Mechanisms* are the two most important direct influence factors. *Traceability* has the strongest influence on perceived *Comprehensibility*. *Reuse Mechanisms* exert the strongest influence on perceived *Maintainability*, *Productivity* and *Reuseability*.

Contrary to the hypothesis formulated in Paper B, moderation effects are nearly as nuanced as the direct influences of MTL Capabilities. The size of meta-models, for example, moderates the influence on *Comprehensibility* and *Ease of Writing*. But the strength of this moderation differs immensely between different MTL Capabilities. Its moderation effect on the influence of *Model Management* onto *Comprehensibility* is about 150 times as strong as the moderation effect on the influence of *Bidirectionality* onto *Comprehensibility*. Overall, we found the transformation size to be the most important moderating factor. It has a consistently high moderating effect on a multitude of influences.

The insights gained from the results of Paper C help to provide clear suggestions on further actions for language developers, researchers & developers to take, thus answering **RQ6**. For further language development, we suggest to focus on the development of transformation specific reuse mechanisms. This is because of the surprisingly high importance of reuse mechanisms for several MTL quality attributes. We also believe that such features provide a unique selling point for MTLs compared to general purpose languages, because they can not be easily adopted in them. Admittedly, this is an argument for domain-specific languages that is often used and, as highlighted in Paper A, does not necessarily hold. However, the results of Paper C show that reuse mechanisms play an important role, which in our view presents an opportunity that should be pursued.

For further empirical evaluation, we suggest investigating the cost of reimplementing MTL abstractions in general purpose languages. Most prominently the cost of manually handling traces. This also includes an assessment of how much tracing is required in real-world use cases to allow for a proper cost-benefit analysis later on. We further point out that during the selection of transformation cases to evaluate, their size has to be a selection criterium. Our results have shown that it is the moderating factor with the highest impact overall.

Paper C provides the first quantitative results for the importance of different language capabilities of model transformation languages. The results open up a variety of empirical research and language development opportunities. Some of these opportunities, namely the evaluation of *Traceability* costs in GPLs, are tackled in the remaining two publications included in this thesis.

The contribution of Paper C is summarised in Figure 1.8.

RQ5	**RQ6**
• Quantitative data on effect strength and significance of each factors influence on quality attributes • Identification of Traceability and Reuse Mechanisms as the two most important factors overall • Quantitative data on the moderation strength of each moderating factor on each investigated influence	• Empirically backed suggestions for the focus of further empirical studies • Empirically backed suggestions for the focus of model transformation language development

Figure 1.8: Contribution of Paper C

1.5.4 Paper D: The Suitability of ATL for Expressing Model Transformations

One of the most common claims about model transformation languages is that they bolster increased expressiveness (Götz et al. 2021a). We investigate this claim in Paper D through an empirical study.

Thus, working towards achieving **G3** by answering **RQ7**:

> **RQ7:** How do domain specific syntax constructs influence the expressiveness of ATL for developing model transformations?

The study focuses on the model transformation language ATL because, as shown by its prevalence in the results reported on in Paper A, B and C, it is one of the most widely known and used transformation languages.

Based on the results from Paper B, the study can be clearly defined as focusing on *automatic trace handling*, *model traversal* and *dedicated model navigation syntax*. As described in Section 1.2 we define additional sub-research questions to define concrete and measurable variables to investigate. The research questions are derived from three claims that have been made multiple times in literature:

***H1*:** *Model transformation languages hide complex semantics behind simple syntax (Gray et al. 2003; Jouault et al. 2008; Krikava et al. 2014; Sendall et al. 2003).*

***H2*:** *Automatic handling and resolution of trace information by the transformation engines is a huge advantage of model transformation languages (Hinkel et al. 2019b; Jouault et al. 2008; Lawley et al. 2007).*

***H3*:** *Model transformation languages allow for implicit rule ordering which can lessen the load on developers (Jouault et al. 2008; Lawley et al. 2007).*

The resulting research questions investigated in Paper D are thus:

> **RQ7.1:** How is the complexity of ATL transformations distributed over multiple transformations and transformation components?
>
> **RQ7.2:** When looking at the complexity distributions of individual transformation components, are there any salient characteristics?
>
> **RQ7.3:** How does the usage of refining mode impact the complexities of ATL modules?
>
> **RQ7.4:** How large is the percentage of bindings that require trace-based binding resolution?
>
> **RQ7.5:** What portion of ATL transformations use implicit rule ordering?

We surveyed GitHub and the ATL Zoo[3] for transformations written in ATL. In total, 33 transformations were selected and their syntactic complexity was calculated. The syntactic complexity measure is based on the collection of ATL measures by Lano et al. (2018). It determines the amount of tokens used in ATL code. We then investigated how much complexity is associated with each part of the transformation. Additionally, we looked into how much tracing was used in the transformations to see how useful automatic trace handling is.

Figure 1.9 shows an alluvial plot for the distribution of syntactic complexity in the analysed ATL modules over different parts of ATL transformations. It visualises the main points for answering **RQ7.1**. First, over half of the complexity resides within bindings meaning that over half of the effort spent is spent on assigning values to the output model. This highlights that ATL enables developers to focus on the main task of a transformation. Moreover, only a small percentage (3%) of the total complexity stems from In-Patterns, i.e., the code concerned with selecting elements to transform. This leads us to draw the a similar conclusion as Hebig et al. (2018). Conditioning on types, as ATL does it, provides a well suited abstraction for model transformation development.

Looking at the distributions for individual transformation components for **RQ7.2**, we found that the majority of bindings in ATL map one attribute of an input model element to one attribute of an output model element. This suggests that the main effort in writing ATL transformations stems from defining how the output should look like. This further highlights the suitability of ATL for transformation development.

Looking at refining mode transformations for **RQ7.3** shows a shift in the structure of transformation modules. In refining transformations much more complexity originates from the In-Patterns and filter expressions therein compared to out-place transformations. We attribute this to the fact

[3] `https://www.eclipse.org/atl/atlTransformations/`

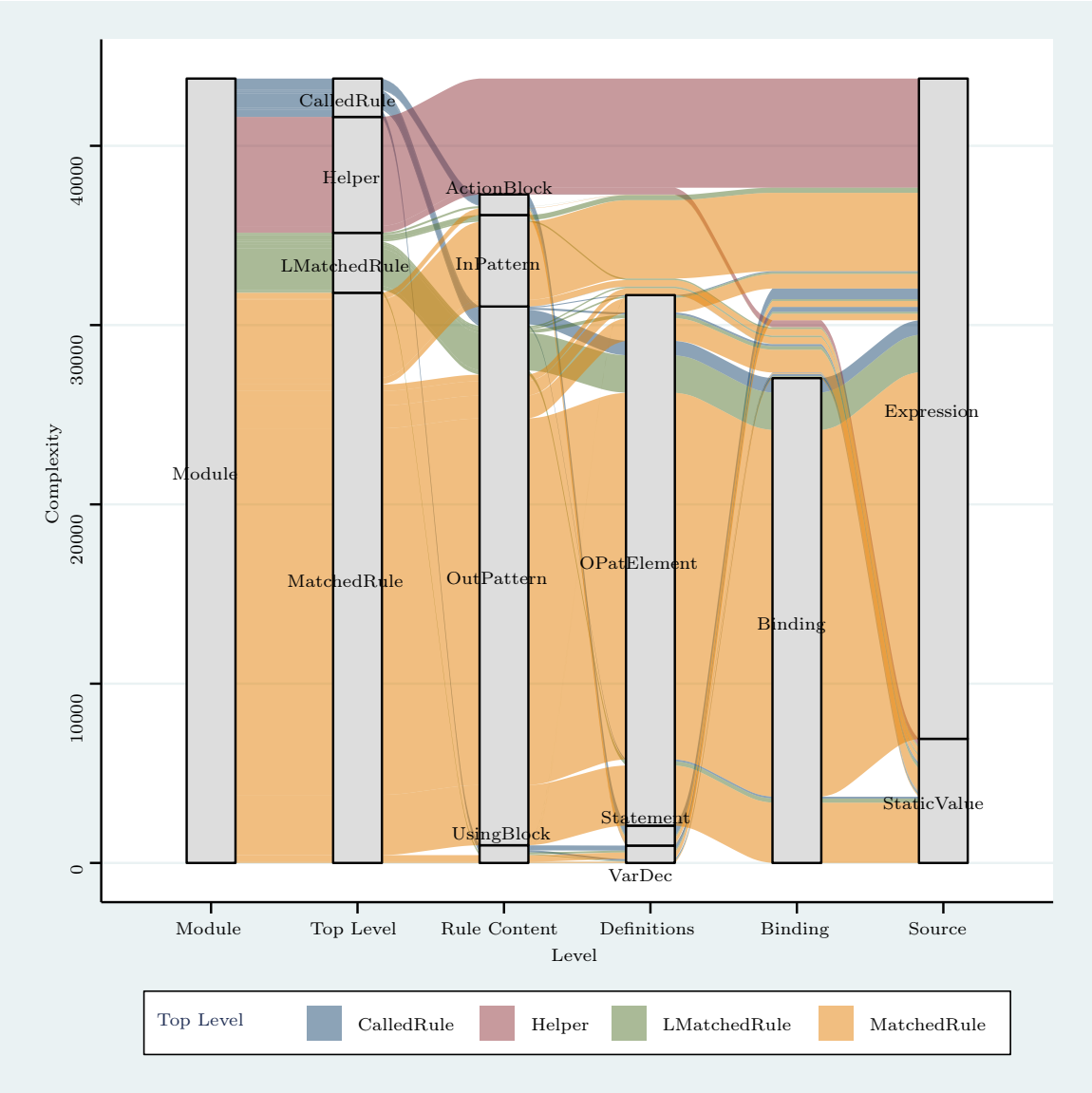


FIGURE 1.9: Distribution of syntactic complexity in ATL

that refining transformations are concerned with changing things in specific elements, thus requiring selection expressions more complex than simply selecting by type.

To explore the importance of automatic trace handling, **RQ7.4** was conceived and answered. We found that about 15% of all bindings require trace information. This suggests less relevance of the automatic trace handling feature than hypothesised. However, in Paper E we show, that this small portion can have huge complexity impacts when it has to be implemented manually.

Lastly, for **RQ7.5** we investigated how much of ATL transformation modules utilise implicit rule ordering to find out how well this abstraction is accepted. Here we found that 79% of all rules are matched rules, meaning rules that utilise implicit rule ordering and selection. To us this suggests that automatic rule selection provides a well suited abstraction for transformation development and can lessens the burden on developers substantially.

The results of Paper D show, that ATL provides several useful abstractions and shifts the focus of transformation development onto the definition of transformation logic. Only little complexity resides in describing how the transformation should be executed or how elements should be selected. Therefore, the expressiveness of ATL for the investigated transformations is high. The way ATL transformations are structured is well suited for its purpose because only 1/5 of all transformations require manual intervention of additional guards.

Apart from the results for the research questions of Paper D, we also found that both GitHub

and the ATL Zoo contain mainly transformations aged a decade or older. This is concerning because it limits studies relying on repository mining in their generalisability to more recent transformation examples.

Figure 1.10 provides an overview of Paper D's contribution.

RQ7
• Data on how much complexity originates from what part of ATL transformations • Indication that the main effort in ATL is spent on mapping input attributes to output attributes • Data showing the effectiveness of ATL's refining mode in reducing code for in-place transformations • Data on how much complexity originates from code involving tracing • Data showing how well received ATL's functionality for implicit rule ordering and transformation target selection is

Figure 1.10: Contribution of Paper D

1.5.5 Paper E: A Historical Perspective on ATL Versus Java Based on Complexity and Size

As shown in the results of Paper B, the properties of GPLs influence how people perceive the quality of MTLs. Moreover, in Paper A we identified the lack of comparisons between GPLs and MTLs. In Paper E, we try to work towards closing this gap and further work on **G3** by answering **RQ8**:

> **RQ8:** How does the complexity of transformations written in ATL compare to those written in Java?

This is done by comparing code distributions over different transformation aspects between ATL and Java. As with Paper D, we split our main research question into additional sub-research questions that properly define variables and metrics to measure and evaluate. The developed research questions are designed as extensions of the research questions from Paper D on which much of the work is based. The following research questions are investigated:

> **RQ8.1:** How much can the complexity and size of transformations written in Java SE14 be improved compared to Java SE5?
>
> **RQ8.2:** How is the complexity of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to each other and ATL?
>
> **RQ8.3:** How is the size of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to each other and ATL?
>
> **RQ8.4:** How does the size of query aspects of transformations written in Java SE5 & SE14 compare to each other and ATL?

We used 12 ATL transformations and translated them into programs that follow coding styles used in Java SE14 and Java SE5 respectively. Java SE14 was chosen for being the state of the art during conduction of the study and Java SE5 was chosen for being current version when ATL was first introduced. The translation uses a translation schema we developed specifically for this study. We manually analysed the Java transformations to assess which code is associated with what transformation aspect. This data was used to associate calculated metrics, i.e., LOC and word count, for Java code with the transformation aspects and compare the distribution with the distribution of syntactic complexity in ATL transformations. The word count measure counts the number of words that are separated either by whitespaces or other delimiters used in the languages, such as a dot (`.`)

and different kinds of parentheses (`()[]{}`). It has already been successfully used by Anjorin et al. (2019) to compare several transformation languages. The syntactic complexity measure is based on the collection of ATL measures by Lano et al. (2018). It determines the amount of tokens used in ATL code and is thus comparable to the word count measure.

We further directly compared the metrics LOC, weighted method count (WMC), based on McCabe complexity, and word count, calculated on Java SE5 and Java SE14 solutions, to investigate whether new language advancements provided improvement for transformation development.

When comparing the complexity of the transformations written in Java SE14 and Java SE5 for **RQ8.1**, we found that both the WMC and lines of code are greatly reduced in Java SE14. However, no significant changes in the number of required words exist. We attribute this to newer language features in Java that reduce the amount of explicit control flow one has to write. These features enable a more functional, data flow driven style of coding which leads to fewer, but much wider lines of code.

Surprisingly the distribution of complexity over the different transformation aspects for **RQ8.2** reveals only slight improvements in Java SE14. In both language versions large portions of the code are overhead produced by manually implementing tracing, model traversal and setup. Even the distribution of how much overhead stems from what transformation aspect stays similar. The only notable exception for this is model traversal, which we were able to significantly reduce the complexity of in Java SE14 by outsourcing it into a generic library.

The results for **RQ8.3** again highlight the overhead entailed when using Java. While in ATL over half of all code is used for defining bindings, in Java it is only about 25% as much.

Lastly, for **RQ8.4** we created a linear regression model to predict the word count of Java expressions based on the corresponding OCL expression. We found that in Java SE5 the complexity of Java expressions raises by a factor of 1.5 while in Java SE14 the complexity raises by a factor of 1.1. This suggests that OCL no longer provides much benefit compared to general purpose code.

Overall the results of Paper E show that new language features in Java SE14 now enable a style of writing transformations with significantly less cyclomatic complexity. At the same time, it is still impossible to hide those transformation aspects that ATL abstracts away from properly. The relative amount of complexity associated with the development overhead stays the same as in Java SE5. We also discovered that the amount of code required to write model transformations in Java stayed the same. All in all, our results point toward Java being useful for transformation development only when there is little tracing necessary and the number of model element types to transform is small.

An overview of the contributions provided by Paper E is shown in Figure 1.11.

RQ8
• Data showing that newer Java version help reduce cyclomatic complexity of transformation implementations • Data showing that newer Java version do not help to reduce the amount of words that need to be written for implementing transformations • Evidence that much overhead code is required to implement domain specific functionality (Model Traversal & Tracing) in Java that ATL provides out of the box • Regression model for predicting the word count of Java SE5 & SE14 expressions based on OCL expressions

Figure 1.11: Contribution of Paper E

1.6 Discussion

The results of the papers included in this thesis provide much needed data in the area of empirical evaluation of model transformation languages. They raise a number of important points to discuss. First, how the lax use of claimed advantages of MLTs in literature led to the current state of discontent and misbelieve on much of the technology surrounding MTLs. Second, the lack of proper

tooling is the most raised critique on model transformation languages. This leads to the question of who is responsible for solving it and how. Third, reasons why little empirical studies on MTLs are conducted. Fourth, we gained a lot of insights into the pros and cons of using either a MTL or GPL for developing transformations. These are discussed in the form of a checklist on key points to consider when deciding on whether to use a GPL or a MTL for transformation development. Lastly, we found noteworthy limitations of using cyclomatic complexity to measure the complexity of data-driven, functional code.

1.6.1 The State of Claims in Literature and How We Got There

The results of Paper A show two things clearly: There are numerous claims about model transformation languages and they are simply brought into being for seemingly no apparent reason.

We believe that claims on MTLs follow a natural progression through research that has not (yet) been completed. A claim starts off as an idea of what might be an advantage or limitation. The idea might be reasonable or visionary. Sometimes such a claim gets demonstrated on a simple example. Sometimes it is just mentioned due to being a vision. The next step is an assessment of how relevant the idea is. If it is not relevant, it gets discarded. If it is relevant, it should get picked up and demonstrated thoroughly or evaluated empirically.

However, for nearly all claims about model transformation languages this last step never happened. Granted, due to the number of claims, it is not feasible to expect this to happen for each claim, but at least the most important claims should follow this path. Instead, most claims just get picked up as a 'fact' or at least are presented in a way that makes them seem like they are taken as a fact. Sometimes this is done without referencing their origin or providing a reasonable explanation as to why they are believed to be true. This further lessens the amount of context given to assess the truthfulness of a statement.

We can only assume why this happens. It could be sloppiness or conscious deceit. It could be that the researcher reiterating the claim has experience leading them to believe it to be reasonable. In any case, much of the chain of reasoning is lost. This presents a problem for people outside of the community, as they are in need of this chain of reasoning to have confidence in what is stated. The end result of this is, that confidence in the research area erodes away. Not only confidence by people outside of the community, but also by those close to it or even involved in it. This is the impression that we got several times when talking to other researchers.

MTL research needs to move towards providing more empirical data on claimed advantages. This provides two major advantages. First, the missing confidence in the technology can be built up. Second, it can guide further research. Instead of blindly building what researchers think will provide a benefit, they can base their decisions on existing data about gaps in languages and technologies.

1.6.2 The Tooling Problem

The responses to our interviews, reported on in Paper B, make it apparent, that many people believe current tooling to be one of the most limiting factors for wider adoption of MTLs. Many participants criticised the proof-of-concept quality of tools and requested more industry ready tools. The strong coupling of MTL tools and the Eclipse eco-system was also seen as approaching a crossroads. The coupling was described as a good idea for early, fast and well-supported development. Nowadays, however, it presents a limitation for adoption as it enforces the complete technology stack to be adopted.

All in all there are many calls for 'someone' to provide better tools, so that the languages and methodology can be taken up more easily. It is not clear who is responsible for this, though.

An argument can be made, that research has done its job in demonstrating that tools are feasible and someone else has to put in the additional effort to make them industry ready. It can also be argued that, due to the lack of empirical evidence, proof-of-concept tools are not enough and more needs to be provided by research. It could also be the case that the tools are mature enough but the know-how to integrate them into existing systems is missing.

We believe that all these areas need to be explored. Research needs to provide clear indication, empirical or demonstrative, that MTL tools can be made industry ready and support all required activities.

We also need research into legacy integration. This enables research to do two things simultaneously. Novel research with novel results can be conducted while providing direct impact for industry applicability.

1.6.3 The Problem With Empirical Research

Empirical research on MTLs has two main problems. First, it is hard to pinpoint where quality attributes originate and how to measure them. Second, a lot of effort is required to create empirical studies which makes them hard to conduct in general (France 2008).

MTL researchers often have had experiences that make them feel that dedicated model transformation languages are better suited for the cases at hand. However, while they gained these experiences through writing many transformations, it is still not easy to pinpoint what part of the used languages make them better suited. The results presented in our Papers A, B and C can remedy this problem to some extent. We provide clear indication of what factors play a role for what quality attributes of MTLs.

The problem of effort involved in conducting empirical studies remains. Many of the identified quality attributes of MTLs require human study subjects to evaluate. Finding enough suitable participants can be time consuming. This is demonstrated by the study conducted by Hebig et al. (2018), one of the only comparative studies between MTLs and GPLs. It took the authors several months to complete the study. Moreover, such studies are also time consuming to set up and to analyse.

Our experience supports this observation as well. The interview study, reported on in Paper B, took over one year from conception to reporting. The questionnaire, reported on in Paper C, also took almost one year to set up, revise and conduct.

An approach to reduce the required effort to find suitable study subjects and gather all necessary data from them is to rely on repository mining. The suitability of this approach is demonstrated by the results of Papers D and E.

Nonetheless, studies involving human subjects are paramount for evaluating quality attributes such as productivity. We therefore suggest, that researchers focus on using both approaches, depending on what attributes are investigated, to increase the body of empirical data on model transformation languages.

1.6.4 MTL vs. GPL: a guide

From our experience comparing Java and ATL to develop model transformations, we draw a number of key points to consider when making the decision which language to use. While we draw these conclusions from comparing Java and ATL, we do believe the points to consider are also relevant when choosing between other MTLs and GPLs. In general, if a complex and repetitive task, that a MTL can abstract away from, is at the centre of development they are a suitable candidate. If this is not the case, the effort of integrating the MTL might be better put into other tasks. Finding the right problem to apply MTLs and MDSE in general to has already been identified as an important aspect to consider by Whittle et al. (2013).

In the following, we present the key points to consider, ordered by on what level of detail the consideration takes place.

At what point of the lifecycle of a project is the decision being made? Deciding between using a MTL or a GPL is often prefaced with deciding whether to use MDSE techniques or not. If a project has been in development for a long period of time, it can be time consuming to try and integrate a MTL for one specific task. It is much easier to use MTLs when the project being developed has been set up to use MDSE from the beginning.

Are there explicit meta-model definitions of the transformed models? Regardless of the development process used, if meta-models for the transformed models exist, model transformation languages are easy to use. MTLs rely on these explicit definitions to support developers in their transformation construction and they are used by transformation engines during execution. Implicitly defined data structures make it hard for MTLs to integrate into the development process. There are, however, some approaches that try to break this barrier such as Dresden OCL (Demuth et al. 2004) which can work on any Java object structure. With the advent of more internal model transformation languages this barrier might not be as relevant in the future. Currently, however,

explicit meta-model definitions are mostly required. As a result, one has to decide if the extra effort to create these definitions is worth it. This is highly use-case dependent.

Are there special requirements like validation, analysis, bidirectionality or incrementality? The heterogeneous structure of MTLs provides an advantage when specific parts of the transformation need to be analysed. The explicit constructs in these languages eases validation and analysis approaches because much less effort is required to extract the relevant transformation parts. Moreover, dedicated constructs with a transformation engine behind them allow for hiding functionality like bidirectionality and incrementality. This can reduce development effort, if these use-cases constitute a large portion of development. Admittedly, there currently do not exist any empirical studies to back up this claim, though it is a commonly shared belief. We are also currently in the process of providing empirical data for this claim in a study of similar setup to what was done for Paper E (Greiner et al. 2023).

Is there a lot of focus on the structure of models to decide what transformations should be applied? There is a consensus between researchers that having to describe complex graph patterns for matching is labour intensive and error prone in GPLs. This was mentioned multiple times during the interviews reported on in Paper B. There again exists no empirical evidence of this, but the number of dedicated query languages for graph structures/databases (Facebook 2016; Francis et al. 2018) indicate some merit to the claim.

Is the transformation explicit or implicit within the system? The easier one can separate the transformation from the rest of the system, the easier it is to integrate it into a MTL. If there is a strong cohesion between the transformation and other aspects of the system it is hard to do this in a separate language. Again, internal MTLs might remedy this problem in the future.

How high is the number of distinct element that are being transformed? If finding the correct element and rule to apply to it is a large part of the transformation, GPLs might not be well suited for the task. They do not provide any abstraction for this and thus much development effort has to be put into a problem, that is already solved in a MTL. We have shown this in Papers D and E.

Does the focus of the transformation lie in structural changes or computation? If there are computationally complex manipulations done to the data within the elements that are being transformed, a GPL might be better suited. They provide powerful language constructs that are optimised for describing what steps should be undertaken to manipulate primitive data. If the focus of the transformation lies more in finding the correct elements to transform and do smaller changes to the data within them, MTLs are better suited. This again is a direct observation of the data we collected for Papers D and E.

Is the amount of required tracing between input and output high or low? Dedicated support for tracing is one of the main advantages of MTLs that we found in Paper E. Thus, if a high amount of tracing between input and output of the transformation is required, MTLs are much better suited for the task. It is still feasible to do this in a GPL, but it requires manual work and thus has a higher chance of introducing errors.

1.6.5 Cyclomatic Complexity in Data-Driven Programming

A startling finding in Paper E was how much the cyclomatic complexity of model transformations dropped between Java SE5 and Java SE14. This is surprising because, apart from obvious deduplication, there is still a lot of conditional branching, when iterating over data structures, involved in defining the transformations in Java SE14.

The main reason why cyclomatic complexity is reduced this much lies in the language constructs used to express these semantics. Iterating over Collections and selecting elements to use is done in the traditional way of using loops in Java SE5. In Java SE14, we opted for using as much functional concepts as possible. This moves development from using explicit loops to the use of streams with mapping or filtering methods which hide the process of iterating over the data structures. Because these method calls are seen as single nodes with one incoming and one outgoing edge in the AST, they do not increase the cyclomatic complexity of the code while still expressing the same semantics as a corresponding loop.

The question arises whether this differentiation is fair. Is the complexity of the code that is written less? An argument can be made that this is the case, as there is less mental load for the developer to understand the core semantics of a piece of code. On the other hand, the developer

still has to think about the implications of what is written. They still have to understand that conditional branching happens during execution.

1.7 Threats to Validity

In this section, we give an overview of the threats to validity of results of this thesis and how we met them. The discussion is structured around the classification of validity threats as defined by Cook et al. (1979). A detailed discussion of the threats for each included paper is given in their respective chapters (Chapters 2 to 6).

1.7.1 Construct Validity

Construct validity describes the extent to which the right measures were applied to investigate the intended research questions (Cook et al. 1979). Construct validity is threatened by erroneous study setups and lack of quality control during the study.

To increase the construct validity of our studies, we relied on existing, much tested guides for all study designs. The appropriateness of the guides and applied methodology was thoroughly discussed. All manual steps involved in our studies were cross-checked by at least one co-author and interviews and survey were pilot tested prior to conduction. All analysis steps also relied on reputable guidelines and established methodologies.

Some threats to construct validity still remain in spite of all these precaution. For example, the use of existing statements as the basis for discussion in our interview study can introduce an unconscious bias in the participants. Participants might be more positive or pessimistic about the topic under discussion, depending on the phrasing of the presented statement. Furthermore, statements containing specific lines of reasoning might lead participants down one line of thought, preventing them from coming up with their own reasoning. We tried to limit these threats by presenting statements that are positively phrased, negatively phrased, contain specific reasoning and contain no particular reasoning. Nonetheless, a threat to construct validity based on these design decisions remains.

The appropriateness of used metrics for our empirical studies also poses a construct validity threat. Cyclomatic complexity has been associated with a number of quality attributes for object oriented programming languages (Jabangwe et al. 2015). The meaningfulness of lines of code, on the other hand, is much debated (Armour 2004; C. Jones 2000) and word count has not been utilised often. We base their appropriateness on the fact that word count can provide a language agnostic complement to LOC and argue that it has been successfully applied in other studies (Anjorin et al. 2019). However, a potential threat to the construct validity based on using these metrics can not be dismissed.

1.7.2 Internal Validity

Internal validity describes the extent of causal relationships between investigated variables in a study (Cook et al. 1979). Internal validity is threatened by manual errors or misunderstandings during the study.

Much like for construct validity, we relied on manual validation through one or two co-authors to decrease the risk of errors threatening the internal validity of our studies.

Threats we were unable to completely alleviate exist in our qualitative studies. Authors in papers and experts during interviews can ascribe differing meanings to the same terms. This can lead to misinterpretation of the original statements. In interviews, we tried to combat this by asking participants to describe their understanding of overloaded terms. In the literature review, we tried to consider the context in which the terms were used to improve our understanding of the authors intended meaning. These efforts aim to remedy the problem but can not fully eliminate it.

1.7.3 External Validity

External validity describes the extent to which the results of a study can be generalised (Cook et al. 1979). External validity is threatened by application of bad sampling methods.

There is much uncertainty surrounding the target population on which to generalise our study results. As a result, it is hard to estimate the exact extent of external validity of our studies. We

aimed to base all our studies on an exhaustive data or participation pool, but limitations due to availability are undeniable. Particularly the interview and questionnaire studies rely on voluntary and convenience sampling threatening their generalisability. All studies utilising repository mining aimed to collect a broad selection of transformations to study, but limitations due to public availability still persist. The literature survey used exhaustive database searches and snowballing to reduce the threat to external validity but it is still possible that relevant studies were missed.

1.7.4 Conclusion Validity

Conclusion validity describes the extent to which conclusions are reproducible (Cook et al. 1979). Conclusion validity is threatened by unreliable measurement methods, data sources and not making the underlying data and methodology available.

To increase conclusion validity of our studies we always make all data sources and raw data available for public use. The publications themselves contain detailed description of how data was accumulated, processed and analysed.

Still, there are several limitations to the conclusion validity of our studies. For example, the interview study and questionnaire rely mostly on responses from researchers within the field of model driven engineering. This makes them an unreliable source of data, because of positive bias towards technology involved in their field of research. Surprisingly, participants in our studies dealt with the topic more critical than expected.

The comparative study between Java and ATL relies on a translation schema we developed ourselves. This poses a threat to conclusion validity because more than one way of translating ATL transformations into Java exists.

1.8 Conclusion and Future Work

This study contributes much needed systematisation and empirical ground work to the body of knowledge on model transformation languages. We do so by applying a broad spectrum of empirical approaches to provide a categorisation of quality attributes associated with MTLs (**G1**), a qualitative and quantitative model of influence factors that facilitate and hamper the perceived quality of MTLs (**G2**) and an empirical analysis of the impacts of transformation specific language constructs as well as a comparison between Java and ATL (**G3**).

Model transformation languages are associated with a broad spectrum of quality attributes. The attributes associated with MTLs range from broad categories (being just better) to transformation specific (better for writing model transformations). We show that there is currently next to no empirical evidence supporting any of these associations. Moreover, constant reiteration of claims leads to a distorted picture of facts, so the meaningfulness of much of the current research can be questioned.

Based on much community wide input we created a structure model portraying factors that influence the perception of the most important quality attributes. The model also contains quantitative data on how strong the influence is.

We supplement the state of empirical research with results on the origin of expressiveness in ATL and a comparative study on ATL and Java transformations. Furthermore, we present considerations on the improvement of writing model transformations in Java over time. Based on these results we discuss what key points to consider when deciding between ATL and Java. These are: State of project before introduction, i.e., new project or legacy project; availability of explicit model definitions; special requirements on analysability, directionality or incrementality; focus of transformation, i.e., pattern selection or computation of input to output; number of distinct meta-model elements involved in the transformation; requirements on tracing.

For future work we see the need for further research in two main areas. First, the body of empirical data on comparison between MTLs and GPLs needs to be expanded. The focus of such studies should be on the quality attributes and factors shown as relevant in our structure model. And second, research to design and develop technology highly relevant for industry adoption needs to be done.

Empirical research can be focused on multiple areas. This starts with expanding our work on comparing Java and ATL with additional data that does not require translation of transformation scripts in one language into the other. Such work can greatly increase the external validity of our results or provide contrasting results that have to be discussed.

It is also necessary to focus on different use-cases for transformations such as incrementality or bidirectionality. Both of these transformation modes are claimed to make development only feasible in dedicated model transformation languages, but this has not yet been evaluated. Another use-case to look at is pattern-matching for graph structures. It is often mentioned as a labour intensive and error prone task if done in a GPL, but there exists no data to back this up.

Apart from empirical studies, we also see a lot of potential for further design-science research in the area of model transformation languages. Such work should focus on current gaps for industrial application. Integration of model transformation concepts into legacy systems is one such area. This can be achieved, e.g., by making MTLs easier to integrate into existing systems or by developing new concepts or processes to integrate MTLs gradually. It is also conceivable that internal MTLs are a key concept for this application area.

Finally, a major issue with adoption of MTLs is the lack of easily accessible learning resources and information on them. There currently exists an enormous barrier to entry and it is thus vitally important to educate people on how to use the available tools and languages properly. We feel that outreach programs such as MDENet[4] need to be expanded upon. Model transformation specific programs can be integrated in these platforms to take advantage of their interconnectedness. They can educate how model transformations fit into the concept of model driven engineering and provide guidance with first steps in utilising them. This gap has already been identified for the whole paradigm of MDE in 2011 by Hutchinson et al. (2011a) but has yet to be tackled properly.

[4] `community.mde-network.org`

Chapter 2

Paper A

Claimed advantages and disadvantages of (dedicated) model transformation languages: a systematic literature review

S. Götz, M. Tichy, R. Groner

International Journal on Software and Systems Modeling (SoSyM), volume 20, pages 469–503, 2021
Springer Nature

Abstract

There exists a plethora of claims about the advantages and disadvantages of model transformation languages compared to general-purpose programming languages. With this work, we aim to create an overview over these claims in the literature and systematize evidence thereof. For this purpose, we conducted a systematic literature review by following a systematic process for searching and selecting relevant publications and extracting data. We selected a total of 58 publications, categorized claims about model transformation languages into 14 separate groups and conceived a representation to track claims and evidence through the literature. From our results, we conclude that: (i) the current literature claims many advantages of model transformation languages but also points towards certain deficits and (ii) there is insufficient evidence for claimed advantages and disadvantages and (iii) there is a lack of research interest into the verification of claims.

2.1 Introduction

Ever since the dawn of Model-Driven Engineering at the beginning of the century, model transformations, supported by dedicated transformation languages (Hinkel 2013), have been an integral part of model-driven development. Model transformation languages (MTLs), being domain specific languages, have ever since been associated with advantages in areas like productivity, expressiveness and comprehensibility compared to general purpose programming languages (GPLs) (Mernik et al. 2005; Sendall et al. 2003; Tratt 2005). Such claims are reiterated time and time again in literature, often without any actual evidence. Nowadays, such an abundance of claims runs through the whole literature body, that one can be forgiven when losing track of which claims verifiably apply and which are still purely visionary.

The **goal** of this study is to identify and categorize claims about advantages and disadvantages of model transformation languages made throughout the literature and to gather available evidence thereof. We do not intend to provide a complete overview over the current state of the art in research. For this purpose we performed a systematic review of claims and evidence in literature.

The main **contributions** of our study are:

- a systematic review and overview over the advantages and disadvantages of model transformation languages as claimed in literature;
- insights into the state of verification of aforementioned advantages and disadvantages;

This study is intended for researchers to (i) raise awareness for the current state of research and (ii) incentivise further research in areas where we identified gaps. The study can also be of interest to practitioners who wish to gain an overview over what research claims about MTLs compared to a practitioners view of the matter.

To systematize information from literature we performed a systematic literature review (Boot et al. 2016; Kitchenham et al. 2007) based on the research questions we defined (see Sect. 2.3.1). As a first step during the review we selected 58 publications from which to extract claims and evidence for advantages and disadvantages of model transformation languages. Afterwards we categorized claims and systematized the evidence to produce (i) a categorization of claimed advantages and disadvantages into 15 separate categories (namely *analysability*, *comprehensibility*, *conciseness*, *debugging*, *ease of writing a transformation*, *expressiveness*, *extendability*, *just better*, *learnability*, *performance*, *productivity*, *reuse and maintainability*, *tool support*, *semantics and verification*, *versatility*) and (ii) a systematic representation of which claims are verified through what means. From our results we conclude that:

1. Current literature claims many advantages and disadvantages of model transformation languages.
2. A large portion of claims are very broad.
3. There is insufficient or no evidence for a large portion of claims.
4. There is a number of claims that originate in claims about DSLs without proper evidence why they hold for MTLs too.
5. There is a lack of research interest in evaluation and especially verification of claimed advantages and disadvantages.

We hope our results can provide an overview over what MTLs are envisioned to achieve, what current research suggest they do and where further research to validate the claimed properties is necessary.

The remainder of this paper is structured as follows: Section 2.2 introduces the background of this research, model driven engineering and model transformation languages. In Sect. 2.3 we will detail the methodology used for the conducted literature review. We present our findings in Sect. 2.4. Afterwards, in Sect. 2.5, we discuss the results of our findings. This section will also include propositions for much needed validation of claims about model transformation languages synthesized from the literature review. Sect. 2.6 contains information about related work and in Sect. 2.7 potential threats to the validity of this research are discussed. Lastly Sect. 2.8 draws a conclusion for our research.

2.2 Background

In this section we provide the necessary background for our study and explain the context in which our study integrates.

2.2.1 Model-Driven Engineering

In 2001 the Object Management Group published the software design approach called *Model Driven Architecture* (OMG 2001) as a means to cope with the ever growing complexity of software systems. MDA placed models at the centre of development rather than using them as mere documentation artefacts. The approach envisions an automated, continuous specialization from abstract models towards code. Starting with so called *Computation Independent Models* (CIMs) each specialization step should provide the models with more specific information about the intended system. Transforming them from *CIM* into *Platform Independent Models* (PIMs) then into *Platform Specific Models* (PSMs) and finally into production ready source code.

The different abstraction levels were designed to enable practitioners to be as platform, system and language independent as possible. The notion of using models as the central artefact during development is what is commonly referred to as *Model-Driven (Software-) Engineering* (MDE/MDSE) or *Model-Based (Software-) Engineering* (MBE/MBSE) (Ciccozzi et al. 2019).

The structure of a model is defined by a so called meta-model whose structure is then also defined by meta-models of their own.

2.2.2 Domain specific languages

"A domain-specific language (DSL) provides a notation tailored towards an application domain and is based on relevant concepts and features of that domain" (Van Deursen et al. 2002). The idea behind this design philosophy is to increase expressiveness and ease of use through more specific syntax. As such DSLs provide an auspicious alternative for solving tasks associated with a specific domain. Representative DSLs include *HTML* for designing webpages or *SQL* for database querying and manipulation.

2.2.3 Model transformation languages

Models are transformed into different models of the same or a different meta-model via so called *model transformations*. Driven by the appeal of DSLs, a plethora of dedicated MTLs have been introduced since the emergence of MDE as a software development approach (Arendt et al. 2010; Balogh et al. 2006; Jouault et al. 2006; Kolovos et al. 2008). Unlike general purpose programming languages, MTLs are designed for the sole purpose of enabling developers to transform models. As a result model transformation languages provide explicit language constructs for tasks performed during model transformation such as model matching. Similar to GPLs model transformation languages can differ vastly in several aspects. Starting with features that can be found in GPLs as well like language paradigm and typing all the way to transformation specific features such as directionality (Czarnecki et al. 2006). There are numerous of features that can be used to distinguish model transformation languages from one another. For a complete classification of these features please refer to Kahani et al. (2019), Mens et al. (2006) or Czarnecki et al. (2006).

Model transformation languages, being DSLs, promise dedicated syntax tailored to enhance the development of model transformations.

2.3 Methodology

Our review procedures are based on the descriptions of literature and mapping reviews from Boot et al. (2016). First of all a protocol for the review was defined. The protocol, as defined in Boot et al. (2016), describes (I) the research background (see Sect. 2.2), (II) the objective of the review and review questions (see Sect. 2.3.1), (III) the search strategy (see Sect. 2.3.2), (IV) selection criteria for the studies (see Sect. 2.3.3), (V) a quality assessment checklist and procedures (see Sect. 2.3.4), (VI) the strategy for data extraction and (VII) a description of the planned synthesis procedures (see Sect. 2.3.5). A complete overview of all steps of our literature review can be found in Fig. 2.1.

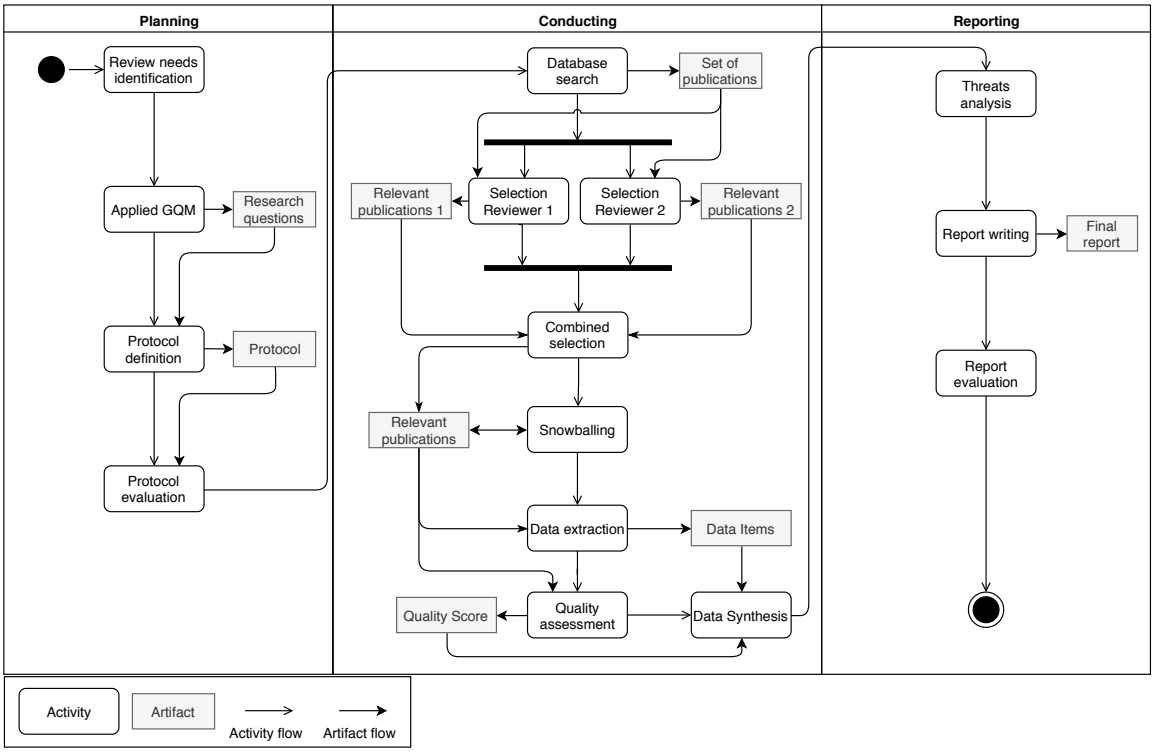


FIGURE 2.1: Protocol overview.

The remainder of this section will describe in detail each of the introduced protocol elements, with the exemption of the research background which we already covered in Sect. 2.2.

2.3.1 Objective and Research Questions

To formulate the objective as well as to derive the research questions for our review we first applied the *Goal-Question-Metric* approach (V. R. Basili et al. 1994) which splits the overall goal into four separate concerns, namely *purpose*, *issue*, *object* and *viewpoint*.

***Purpose*:** Find and categorize

***Issue*:** claims of and evidence for advantages and disadvantages

***Object*:** of model transformation languages

***Viewpoint*:** from the standpoint of researchers and practitioners.

Based on the described goal we then extracted the two main research questions for our literature review:

RQ1: What advantages and disadvantages of model transformation languages are claimed in literature?

RQ2: What advantages and disadvantages of model transformation languages are validated through empirical studies or by other means?

The aim of **RQ1** is to provide an extensive overview over what kinds of advantages or disadvantages are explicitly attributed to using dedicated model transformation languages compared to using general purpose programming languages. We consider such an overview to be necessary, because the number of claims and their repetition in literature to date makes it difficult to keep track of which claims verifiably apply and which are still purely visionary. Naturally to be able to distinguish between substantiated and unsubstantiated claims it is also required to record which claims are supported by evidence. With **RQ2** we aim to do exactly that. Combining the results of **RQ1** and **RQ2** then makes it possible to determine if, and how, a positive or negative claim

about MTLs is verified. Additionally this also enables us to identify those claims that have yet to be investigated.

2.3.2 Search Strategy

Our search strategy consists of seven consecutive steps. A visual overview of the complete search process can be found in Fig. 2.3. The figure visualizes steps *Database search* to *Snowballing* from Fig. 2.1 in more detail.

In the first step we defined the search string to be used for automatic database searches. For this we identified major terms concerning our research questions. Each new term was made more specific than the previous one. The resulting terms and justifications for including them were:

- *Model Driven Engineering*: The overall context we are concerned with. This was included to ensure only papers from the relevant context were found.
- *Model Transformation*: The more specific context we are concerned with.
- *Model Transformation Language*: Since our focus is on the languages to express model transformations.

We used a thesaurus to identify relevant synonyms for each term in order to enhance our search string. In addition, we included one representative model transformation language with graphical syntax, one imperative language, one declarative language and one hybrid language as well as the term *domain specific language* and its synonyms. The selection of the representative languages was made on the basis of their widespread use, active development and in the case of QVT because it is the standard for model transformations adopted by the Object Management Group. All these additional terms were included as synonyms for the *model transformation language* term.

We dropped the terms *advantage* and *disadvantage* after initial searches, because they resulted in a too narrow of a result set which excluded key publications (Hebig et al. 2018; Hinkel et al. 2019b) manually identified by the authors.

To combine all keywords we followed the advice of Kofod-Petersen (2015) to use the Boolean ($\vee$) to group together synonyms and the Boolean ($\wedge$) to link our major term groups.

This resulted in the search string shown in Fig. 2.2 which was applied in full text searches.

(Model Driven Engineering $\vee$ MDE $\vee$ Model Based Engineering $\vee$
MBE $\vee$ Model Driven Development $\vee$ MDD $\vee$
Model Driven Software Engineering $\vee$ MDSE $\vee$
Model Driven Software Development $\vee$
MDSE $\vee$ Model-Driven Software Development $\vee$
Model-Driven Engineering $\vee$ Model-Based Engineering $\vee$
Model-Driven Software Engineering)
$\wedge$
(Model Transformation $\vee$ Transformation $\vee$
Model Transformations $\vee$ Transformations)
$\wedge$
(Model Transformation Language $\vee$ Transformation Language $\vee$
ATL $\vee$ Henshin $\vee$ QVT $\vee$ TL $\vee$
Transformation Languages $\vee$ DSL $\vee$ domain specific language $\vee$
Model Transformation Languages)

FIGURE 2.2: Search string used for automatic database searches

We decided on the following four search engines to use for automated literature search:

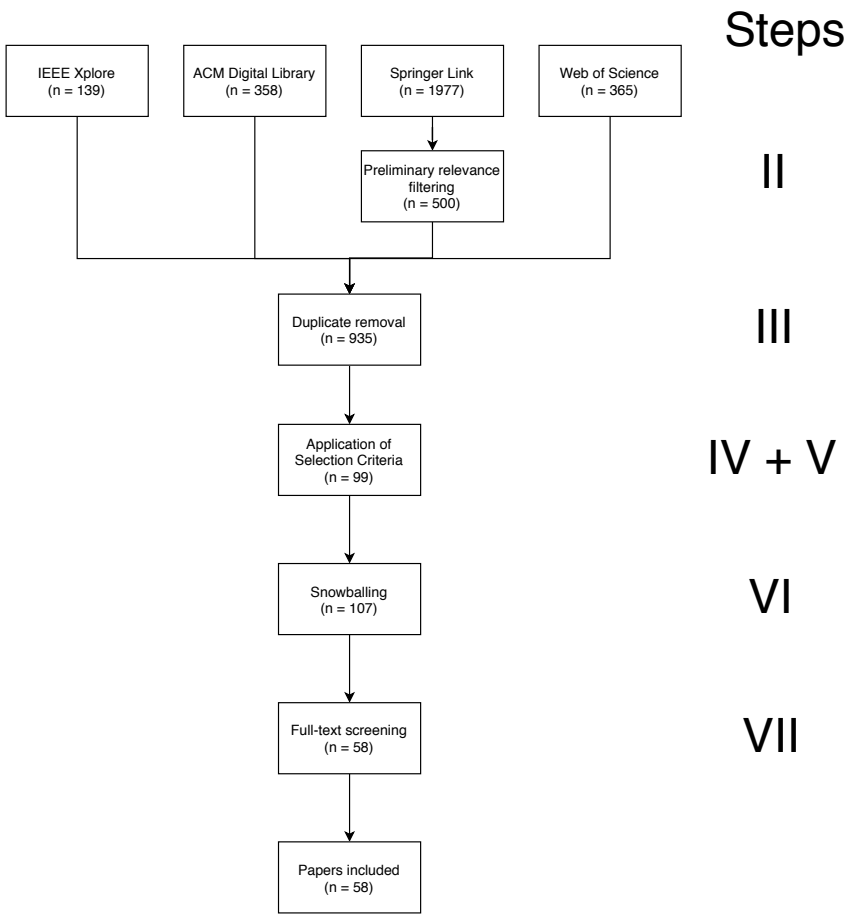


FIGURE 2.3: The search and selection process.

- ACM Digital Library
- IEEE Xplore
- Springer Link
- Web Of Science

Search engines were chosen based on their overall coverage, completeness, the availability of accessible publications and usage in other literature reviews in this field such as Barat et al. (2017) and Loniewski et al. (2010). The online library *Science Direct*, which is often used in this domain, was excluded from our list due to us only having limited access to the publications in the data base. We decided that the overhead of requesting access to all publications for which our proceedings would require a full text review (see step four) would take up too much time thus we excluded the database from our automatic search process. Badampudi et al. (2015) also show that combining the automatic database searches with an additional snowballing process can make up for a reduced list of searched databases. We also decided against using Google Scholar as a search engine due to our experience with it producing too many irrelevant results and having a large overlap with ACM Digital Library and IEEE.

We conducted several preliminary searches on all four databases during the construction of the search string, to validate the resulting publications included key publications.

After the definition and validation of the search string, the second step consisted of full text searches using the search engines of *ACM Digital Library*, *IEEE Xplore Digital Library* and *Web of Science*.

For the *Springer Link* database we realized early on that a full text search would result in too many hits and instead opted to query only the titles for the keyword *model transformation language* and its synonyms and filtered these results by applying a full text search based on the remaining

keywords and their synonyms. The remaining results still far exceeded those of all other databases combined. We further realized during preliminary sifting, that the neither title nor abstracts of publications beyond the first 200 results suggested a relevance to our study. For that reason we decided to cap our search at 500 publications, doubling the size of results from the point where the relevance of publications started to slide. This decision is supported by the fact that any publication which ended up in our data extraction set was found within the first 200 results.

All automated database searches were conducted between the 17th and 28th of June 2019.

In the third step all duplicates that resulted from using multiple search engines were filtered out based on the publication title and date. This also included the removal of publications that had extended versions published in a journal. This resulted in a total of 935 publications.

During the forth step two researchers independently used the selection criteria (see Sect. 2.3.3) on the titles and abstracts to select a set of relevant publications. The researchers categorized literature as either *relevant* or *irrelevant*. And in cases where they could not deduce the relevance based on the title and abstract the publication was marked as *undecidable*.

Afterwards in step five the results for each publication of the independent selection processes were compared. In cases where the two researchers agreed on *relevant* or *irrelevant* the paper was included or excluded from the final set of publications. In cases of either a disparity between the categorizations or an agreement on *undecidable*, the full text of the publications was consulted using adaptive reading techniques to decide whether it should be included or excluded. Adaptive reading in this context meant going from reading the introduction to reading the conclusion and if a decision was still not reached reading the paper from start to finish until a decision could be reached. The step resulted in a total of 99 publications to use as a start set for the sixth step.

In the sixth step we applied exhaustive backward and forward snowballing, meaning, as described in many previous studies (Auer et al. 2018; Somasundaram et al. 2003), until no new publication was selected. The snowballing procedures followed the guidelines laid out by Wohlin (2014). Our start set was comprised of all 99 publications from step 5. We then applied backward and forward snowballing to the set. For backward snowballing we used the reference lists contained in the publications and for forward snowballing we used Google Scholar as suggested by Wohlin (2014) and because from our experience it provides the most reliable source for the cited by statistic. To the cited and citing publications we then applied our inclusion and exclusion criteria as described in step 4. All publications that were deemed as relevant were then used as the starting set for the next round of snowballing until no new publications were selected as relevant. The result of this step was a set of 107 *relevant* publications.

Lastly, in step seven, we filtered out all publications that did not explicitly mention advantages or disadvantages of model transformation languages by reading the full text of all publications. This step was introduced to filter out the noise that arose from a broader search string and less restrictive inclusion criteria (see Sect. 2.3.3). The remaining 58 publications form our final set on which data synthesis was performed on (a list of all included publications with an unique assigned ID can be found in Appendix A).

2.3.3 Selection Criteria

We decided that a publication be marked as relevant, if it satisfies at least one inclusion criteria and does not satisfy any exclusion criteria. The inclusion criteria were chosen to include as many papers that potentially contain advantages or disadvantages as possible. A publication was included if:

***IC1*:** The publication introduces a model transformation language.

***IC2*:** The publication analyses or evaluates properties of one or multiple model transformation languages.

***IC3*:** The publication describes the application of one or multiple model transformation languages.

IC1 is an inclusion criteria, because the introduction of a new language should include a motivation for the language and possibly even a section on potential shortcomings of the language. Such shortcomings can be attributed either to the design of the language or to the concept of model transformation languages as a whole.

A publication that is covered by *IC2* can help answer both **RQ1** and **RQ2** depending on the analysed/evaluated properties.

IC3 forms our third inclusion criteria since experience reports can be a good source for both strengths and weaknesses of any applied technique or tool.

Our exclusion criteria were:

***EC1*:** Publications written in a language other than English.

***EC2*:** Publications that are tutorial papers, poster papers or lecture slides.

***EC3*:** Publications that are a Doctoral/Bachelor/Master thesis.

EC1 ensures that the scientific community is able to verify our extracted data from publications.

Because tutorial papers, poster papers and lecture slides are less reliable and do not provide enough information to work with, they are excluded with *EC2*.

Lastly, to reduce the required workload, we excluded all thesis publications with *EC3* as full text reviews would take up too much time. We also argue that relevant thesis findings are most likely also published in journal or conference papers.

2.3.4 Quality Assessment Checklist and Procedures

Assessing the quality of publications found during the selection process is an essential part of a literature review (Boot et al. 2016).

For that reason, we adopted a list of six quality attributes for studies. The quality attributes (seen in Table 2.1) are taken from Shevtsov et al. (2018) which adapted quality criteria from Weyns et al. (2012). Each quality item has a set of three characteristics for which a value between 0 and 2 is assigned. The quality score of a publication is calculated by summing up the values for each characteristic, making 12 the maximum quality score for a publication. The quality score did not influence the decision to include or exclude a publication.

2.3.5 Data Extraction Strategy

Based on our research questions, and general documentation concerns, we devised a total of eight data items to extract from each selected publication. Table 2.2 lists all extracted data items.

Data items *D1*-*D3* are recoded for documentation purposes.

To gather explicitly claimed advantages and disadvantages of model transformation languages *D4* and *D5* are necessary items to include.

Another goal of our literature review is to find out which advantages or disadvantages are empirically verified. It is therefore necessary to extract information about whether empirical evidence exists and which advantage or disadvantage it is concerned with (*D6*). Similarly, citations used to back up claimed advantages or disadvantages are also documented (*D7*). Our goal is it to either track down references that provide evidence and find sources of common claims about advantages and disadvantages of model transformation languages.

Lastly, in order to evaluate the quality of publications the quality score *D8* for each publication is recorded.

All data items were extracted during full text reviews of all selected publications.

2.3.6 Synthesis Procedures

The synthesis of the collected data was split into multiple parts with multiple results for each research question.

2.3.6.1 RQ1: What advantages and disadvantages of model transformation languages are claimed in literature?

The first part of the synthesis for **RQ1** was a simple collection of all claimed advantages and disadvantages. This was done in order to create a basic overview.

Table 2.1: Quality assessment criteria (Weyns et al. 2012).

	Q1: Problem definition
2	The authors provide an explicit problem description.
1	The authors provide a general problem description.
0	There is no problem description.
	Q2: Problem context
2	If there is an explicit problem description for the research, this problem description is supported by references.
1	If there is a general problem description, this problem description is supported by references.
0	There is no description of the problem context.
	Q3: Research design
2	The authors explicitly describe the plan (different steps, timing,etc.) they have used to perform the research, or the way the research was organized.
1	The authors provide some general words about the research plan or the way the research was organized.
0	There is no description of how the research was planned/organized.
	Q4: Contributions
2	The authors explicitly list the contributions/results.
1	The authors provide some general words about the results.
0	There is no description of the research results.
	Q5: Insights
2	The authors explicitly list insights/lessons learned.
1	The authors provide some general words about insights/lessons learned.
0	There is no description of the derived insights.
	Q6: Limitations
2	The authors explicitly list problems and/or limitations.
1	The authors provide some general words about limitations and/or problems.
0	There is no description of the limitations.

Next an analysis of all collected items was done in order to devise categories for the advantages and disadvantages. To develop categories we used initial coding and focused coding as described by Charmaz (2014). First all claims were analysed claim by claim to extract common phrases or similar topics. These were then used to group together claims and develop descriptive terms when then served as the name for the category formed by the grouped claims. The categories themselves were split into a positive section and a negative section to contrast negative and positive mentions with each other.

Using the devised categorization allows for quick identification of contradictory claims. Such claims then have to be further analysed in terms of origin, context and supporting evidence.

2.3.6.2 RQ2: What advantages and disadvantages of model transformation languages are validated through empirical studies or by other means?

To analyse evidence of claimed advantages and disadvantage we started by assessing the quality of each respective publication using the quality score system from Sect. 2.3.4.

Afterwards we devised a visual representation for claims and evidence thereof in publications. The representation allows a straightforward identification of substantiated and unsubstantiated claims and tracking of citations back to the origin of cited claims. This in turn enabled us to easily identify whether citations back up stated claims or serve as nothing more than a reference to a publication which claims the same thing.

Table 2.2: Data items.

ID	Data	Purpose
D1	Author(s)	Documentation
D2	Publication year	Documentation
D3	Title	Documentation
D4	Named advantage(s) of MTL(s)	RQ1
D5	Named disadvantage(s) MTL(s)	RQ1
D6	Empirical evidence of advantage(s) or disadvantage(s)	RQ2
D7	Cited evidence	RQ2
D8	Quality score	Documentation

2.4 Findings

In this section we provide a summary of the synthesized data as well as an analysis of the demographics and quality of publications. The summary will be in narrative form, supported by plots and graphs as suggested by Boot et al. (2016). Before describing our findings with regard to the research questions from Section 2.3.1, we first offer statistics and information about the demographic data of the collected literature as well as an overview over their quality which we assessed using the quality criteria from Section 2.3.4.

2.4.1 Demographics

Fig. 2.4 provides an overview over the quantity of included publications per year. A interesting thing to note is, that it took only two years from the introduction of the *Model Driven Architecture* in 2001 to the first mentions of advantages of model transformation languages. One of the most cited papers about model transformations in our literature review was published that year too (**P63**). Its title shapes introductions of publications in the community even today: *Model transformation: The heart and soul of model-driven software development.*

Scrutinizing claims about MTLs however, just recently started to be a focus of research. With the first study (**P59**) dedicated to evaluating advantages of MTLs being published in 2018. To us this suggests that research might be slowly catching on to the fact that evaluation of specific properties of MTLs is necessary instead of relying on broad claims. Simply relying on the fact that model transformation languages are DSLs and that DSLs in general fare better compared to non domain specific languages (D. Batory et al. 2002; Hailpern et al. 2006; Kieburtz et al. 1996) is not enough.

Industrial case studies about the adoption of MDSE have been performed much earlier than 2018 but such studies mainly focus on the complete MDSE workbench and do not analyse the impact of the used MTLs in great detail. The case study **P670** for example, while stating that "The technology used in the company should provide advanced features for developing and executing model transformations", does not go into detail about neither current shortcomings nor any other specifics of model transformation languages used during the development process.

Overall there are *32* publications that mention advantages and *36* publications that mention disadvantages. Moreover *four* publications provide empirical evidence for either advantages or disadvantages while *12* publications use citations to support their claims and *14* publications use other means such as examples and experience (more on this in Sect. 2.4.4).

Lastly Table 2.3 shows which transformation languages were directly involved in publications used in our data extraction. We counted a transformation languages as being involved if it was used, analysed or introduced in the publication. Simply being mentioned during enumerations of example MTLs was not sufficient.

The table paints an interesting picture. ATL far exceeds all other model transformation languages in involvement and most languages are only discussed in a single publication.

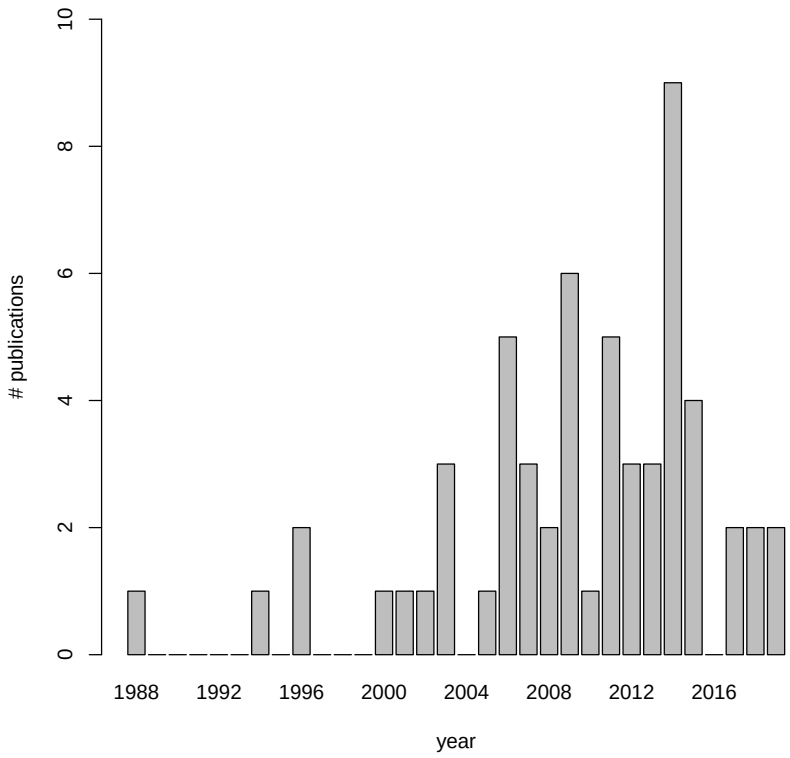


FIGURE 2.4: Number of publications that mention or evaluate advantages or disadvantages of MTLs per year.

2.4.2 Quality of publications

The results from the quality assessment, summarized in Figure 2.5, shows that both the problem context and definition as well as the overall contributions are well defined in a majority of publications. Insights drawn from the work described in these publications, while less comprehensive in many cases, are also described most often. However thorough descriptions of the research design, the used methods or steps taken are less common. A trend which is even more prominent for the presentation and discussion of limitations that act upon the studies. Similar observations have already been made by other literature reviews in different domains (Galster et al. 2014; Shevtsov et al. 2018).

2.4.3 RQ1: Advantages and Disadvantages of Model Transformation Languages

We used data items D4 and D5 to answer our first research question, namely which advantages or disadvantages of dedicated model transformation languages are claimed in literature. The resulting statements were sorted into 15 different categories (seen in Fig. 2.6) which arose naturally from the collected statements. An overview over all claims sorted into the different categories can be found in Table A.1. The table ascribes each claim with a unique ID (Cxx) for reference throughout this work. The table also contains evidence used to support a claim (if existent) to which we will come back later in Sect. 2.4.4. For almost all categories there exist papers that describe model transformation languages as being advantageous as well as publications that describe them as disadvantageous in the category. In the following we discuss the statements made in publications for each the category.

Table 2.3: Number of publications that mention specific MTLs.

Model transformation language	# of mentions
ATL	16
EMT	1
ETL	3
GreAT	1
Henshin	1
Iquery	1
JTL	1
MOFLON	1
MT	1
NTL	2
QVT-O	4
QVT-R	2
SDM	1
SIGMA	1
SiTra	1
Tefkat	1
TGG	1
TN	1
VMTL	1

2.4.3.1 Analysability

Throughout our gathered literature there is only one publication, **P45**, that mentions analysability. According to them a declarative transformation languages comes with the added advantage of being automatically analysable which enables optimizations and specialized tool support (*C1*). While a detailed discussion of this claim within the publication remains owed, the authors provide examples of how static analysis allows the engine to implicitly construct an execution order. While our literature review found only a single publication that explicitly mentions analysability as an advantage of model transformation languages, there do exist multiple publications (Marcel F. van Amstel et al. 2011b; Arendt et al. 2010; Varró et al. 2006) that contain analysis procedures for model transformations.

2.4.3.2 Comprehensibility

Comprehensibility is a much disputed and multifaceted issue for model transformation languages. A total of eleven publications touch on several different aspects of how the use of MTLs influences the understandability of written transformations.

The first aspect is the use of graphical syntax compared to a textual one which is typically used in general-purpose programming languages. In **P63** the authors talk about *"perceived cognitive gains"* of graphical representations of models when compared to textual ones (*C6*). A pronouncement that is echoed in **P43** which state that graphical syntax for transformations is more intuitive and beneficial when reading transformation programs (*C2*).

While all these claims about graphical notation increasing the comprehensibility of transformations stand undisputed in our gathered literature, there are other facets in which graphical notation is said to be disadvantageous. We will come back to them later on in Sect. 2.4.3.5.

Declarative textual syntax is another commonly used syntax for defining model transformations. The authors of **P45** contend that a declarative syntax makes it easy to understand transformation rules in isolation and combination (*C3*). However, declarative transformation languages are typically based on graph transformation approaches which can become complex and hard to read according to **P70** (*C13*). They additionally assert that the use of abstract syntax hampers the comprehensibility of transformation rules (*C12*). Furthermore **P22** insist that the use of graph patterns results in only parts of a meta-model being revealed in the transformation rules and that current transformation languages exhibit a general lack of facilities for understanding transformations (*C8*). **P22** also reports that understanding transformations in current model transformation languages is

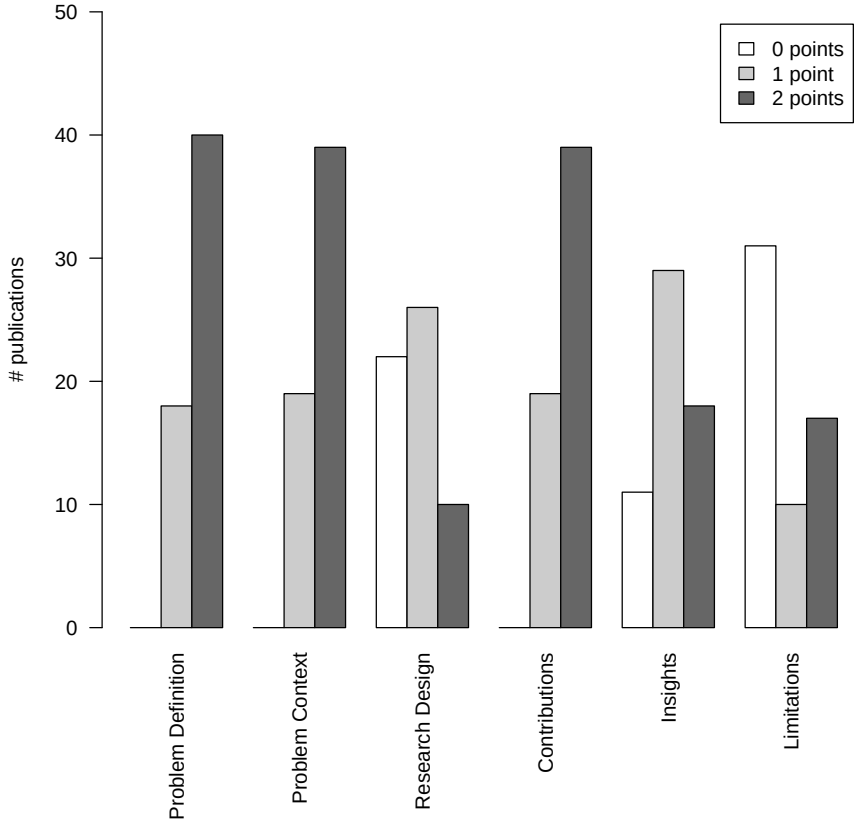


FIGURE 2.5: Quality score distribution.

hampered, specially by the fact that many of the involved artefacts such as meta-models, models and transformation rules are scattered across multiple views (*C9*). **P29** brings forward the concern that large models are also a factor that hampers comprehensibility since there exist no language concepts to master this complexity (*C11*). Adding to this point, **P27** describes that for non experts (e.g stakeholders) transformations written in a traditional model transformation language are *"very complex to understand"* because they lack the necessary skills (*C10*). The authors of **P95** on the other hand claim that the usage of dedicated MTLs, which incorperate high-level abstractions, produces transformations that are more concise and more understandable (*C7*). This sentiment is shared in **P44** which explains the belief that using GPLs for defining synchronizations brings disadvantages in comprehensibility compared to model transformation languages (*C3*).

Understanding a transformation requires, among other things, understanding which elements are affected by it and in which context a transformation is placed. Using a model transformation language is beneficial for this as shown in the study described in **P59** (*C5*).

2.4.3.3 Conciseness

Interestingly there seems to be a consensus on the conciseness of model transformation languages compared to GPLs.

In general dedicated model transformation languages are seen as more concise (**P63** *C17*, **P95** *C21*) which, apart from textual languages, is also stated for graphical languages in **P75** (*C18*).

The fact that MTLs are more abstract making them more concise and thus better is claimed multiple times in **P80** (*C19*), **P52** (*C15*), **P3** (*C14*) and **P95** (*C20*) while **P673** claims that the abstraction in MTLs helps to reduce their overall complexity (*C22*).

The SLOC metric has also been drawn from as a way to compare MTLs other MTLs and even GPLs. According to an experiment described in **P59**, using a rule based model transformation reduces the transformation code by up to 48% (*C16*). Whether or not this is any indication of superiority is a disputed subject (Barb et al. 2014).

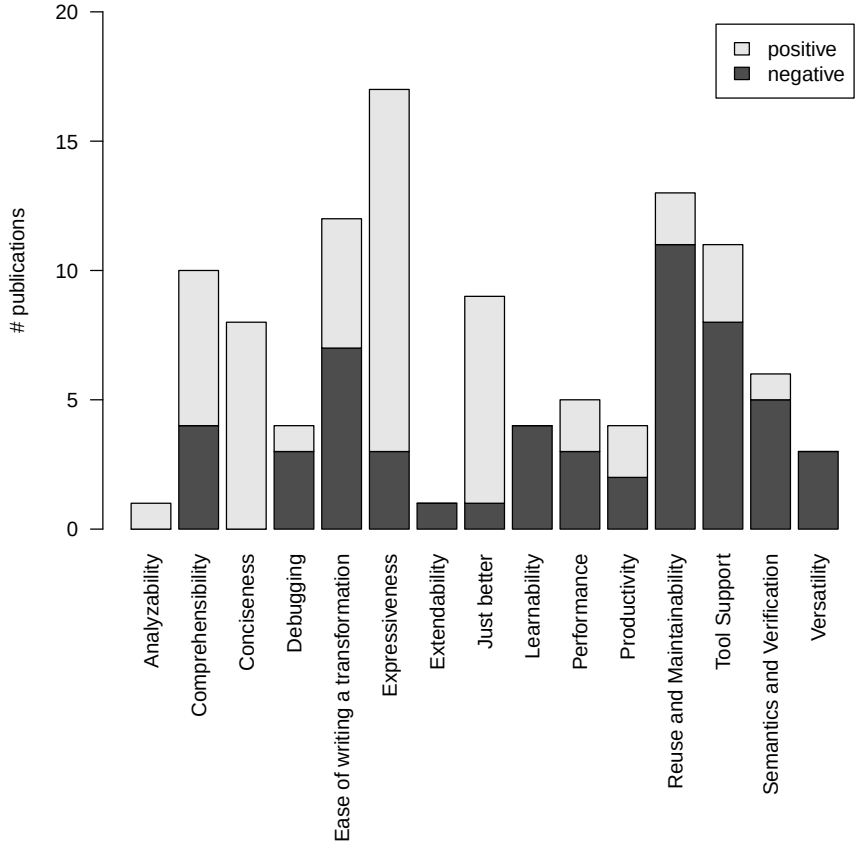


FIGURE 2.6: Number of publications that claim an advantage or disadvantage of MTLs in a category.

2.4.3.4 Debugging

Debugging support is much less disputed than comprehensibility. Of the five publications that talk about debugging in model transformation languages none praise the current state of debugging support.

P22 (*C24*, *C25*) and **P90** (*C27*) both describe that currently no sufficient debugging support exist for MTLs. And while in **P95** it is stated that debugging of transformations in a dedicated languages is likely better than when the transformation is written in a general-purpose language (*C23*) they fail to bring forth a single example for their assertion.

Lastly **P45** lauded declarative syntax for its benefit in comprehension but also note that imperative syntax is easier to debug in general (*C26*).

2.4.3.5 Ease of writing a transformation

The main purpose of model transformation languages is to improve the ease with which developers are able to define transformations. Hence this should also be a main benefit when compared to general-purpose languages. However the authors of the study described in **P59** found:*"no sufficient (statistically significant evidence) of general advantage of the specialized model transformation language QVT-O over the modern GPL Xtend"* (*C39*). This is not to say that there are none as the authors admit the conclusions were *"made under narrow conditions"* but is still a concerning finding. Much more so because claims about such benefits of using MTLs persist through literature. Claims such as those described in **P29** (*C29*), **P672** (*C32*) and **P50** (*C30*), which state that their simpler syntax make it easier to handle and transform models. These claims draw from statements about the expressiveness, to which we will come to in the next section, and reason that better expressiveness must lead to an easier time in writing transformations. A potential reason that hampers model transformation languages from evidentially being better for writing transformations is cited in **P27** (*C34*) and **P28** (*C35*). They both state that using a model transformation language requires skill,

experience and a deep knowledge of the meta-models involved (**P56** *C38*). In our opinion, however, this holds true regardless of the language used to transform models.

Moreover, many model transformation languages use declarative syntax which can be unfamiliar for many programmers, according to **P45** (*C37*) and **P63** (*C40*), which are much more familiar with the status quo, i.e. imperative languages. The authors of **P22**, on the other hand, state, that imperative MTLs often require additional code since many issues have to be accomplished explicitly compared to implicitly in declarative languages (*C33*).

Lastly graphical syntax is said to make writing model transformations easier as the syntax is purported to be more intuitive for this task compared to a textual one in **P3**. In **P43** (*C36*) and **P672** (*C41*), however, the authors claim that graphical syntax can be complicated to use, that textual syntax is more compact and does not force users to spend time to beautify the layout of diagrams.

2.4.3.6 Expressiveness

As described in Sect. 2.2.2, the idea behind domain specific languages is to design languages around a specific domain, thus making it more expressive for tasks within the domain (Mernik et al. 2005). Since model transformation languages are DSLs it should not be a surprise that their expressiveness in the domain of model transformations is mentioned almost exclusively positive by a total of 19 different publications found in our literature review.

A large portion (**P95**, **P80**, **P94**, **P63**, **P15**, **P40**, **P52**, **P70**) of publications that refer to expressiveness state, that the higher level of abstraction that results from specific language constructs for model manipulation increases the conciseness and expressiveness of MTLs. **P80** additionally asserts, that model transformation languages are just easier to use (*C61*).

Another portion (**P2**, **P15**, **P45**, **P677**, **P27**, **P63**, **P95**, **P27**) explains that the expressiveness is increased by the fact that model transformation engines can hide complexity from the developer. One such complex task is pattern matching and the source model traversal as mentioned in **P2** (*C42*), **P15** (*C43*) and **P45** (*C53*) respectively. According to them not having to write the matching algorithms increases the expressiveness and ease of writing transformations in MTLs. Implicit rule ordering and rule triggering is another aspect that **P15** (*C46*), **P45** (*C51*) and **P677** (*C65*) claim increases the expressiveness of a transformation language. Related to rule ordering is the internal management and resolution of trace information which is stated by **P15** (*C44*), **P45** (*C50*), **P677** (*C65*) and **P95** (*C64*) to be a major advantage of model transformation languages. Furthermore, **P45** asserts that, implicit target creation is another expressiveness advantage that MTLs can have over general-purpose languages (*C52*). Lastly the study described in **P59** observed that copying complex structures can be done more effective in MTLs (*C56*).

However we also uncovered some shortcomings in current syntaxes. **P10** argues that the lack of expressions for transforming a single element into fragments of multiple targets is a detriment to the expressiveness of transformation languages, going as far as to allege that without such constructs model transformation languages are not expressive enough (*C68*). **P32** implies that MTLs are unable to transform OCL constraints on source model elements to target model elements (*C69*). And lastly **P33** critiques that model transformation languages lack mechanisms for describing and storing information about the properties of transformations (*C70*).

2.4.3.7 Extendability

Being able to extend the capabilities of a model transformation language seems to be less of a concern to the community. This can be seen by the fact that only **P50** touches this issue. They explain that external MTLs can only be extended (*"if at all"*) with a specific general-purpose language (*C71*). Internal model transformation languages of course do not suffer from this problem since they can be extended using the host language (Jesús Sánchez Cuadrado et al. 2006; Hinkel et al. 2019a; Křikava et al. 2014).

2.4.3.8 Just better

Apart from specific aspects in which the literature ascribes advantages or disadvantages to model transformation languages, there are also several instances where a much broader claim is made.

P86 for example states that there exists a consensus that MTLs are most suitable for defining model transformations (*C78*). This claim is also reiterated in several other publications using

statements such as *"the only sensible way"* or *"most potential due to being tailored to the purpose"* (**P9**, **P23**, **P63**, **P64**, **P66**). However one publication claims, that both GPLs and MTLs are not well suited for model migrations, and that instead dedicated migration languages are required (**P34** *C80*).

2.4.3.9 Learnability

The learnability issues of tools have been shown to positively correlate with usability defects (Alves et al. 2016) and thus their general acceptance.

However the learnability of model transformation languages is rarely discussed in detail. **P30** (*C81*), **P58** (*C83*) and **P81** (*C84*) all express concerns about the steep learning curve of model transformation languages and **P52** explain that transformation developers are often required to learn multiple languages which requires both time and effort (*C82*).

2.4.3.10 Performance

The execution performance of transformations is an important aspect of model transformations. Often times the goal is to trigger a chain of multiple transformations with each change to a model. Hence good transformation performance is paramount to the success of model transformation languages.

Opinion on performance in literature is divided. On one hand there are publications such as **P52** (*C88*) and **P80** (*C89*) which describe that the performance of dedicated MTLs is worse than that of compiled general-purpose programming languages. While on the other hand there is **P95** which states that some introduced transformation languages are more performant (*C85*), citing articles from the Transformation Tool Contest (TTC), and **P675** which shows a performance comparison of transformations written in Java and GrGen where GrGen performs better than Java (*C86*). There are also more nuanced views on the subject. **P45** describes that practitioners sometimes perceive the performance as worse and that there exist factors that hamper the performance (*C87*). The listed factors are the fact that the transformation languages are often interpreted, a mismatch with hardware and less control over the algorithms that are used. However they also describe that specialized optimizations can bridge the performance gap.

2.4.3.11 Productivity

Increased productivity through the use of DSLs is a much cited advantage (Mernik et al. 2005) (*C6D*). Unsurprisingly it resurfaces in various forms in the context of model transformation languages as well. For instance, in **P45** it is described, that the use of declarative MTLs improves the productivity of developers (*C91*). **P29** goes even further, claiming that the use of any model transformation language results in higher productivity (*C90*).

This is contrasted by the hypothesis that productivity in general-purpose programming languages might be higher due to the fact that it is easier to hire expert users which was put forward in **P59** (*C93*). Lastly **P32** raises the concern that some of the interviewed subjects perceive model transformation languages as not effective, i.e. not helpful for the productivity of developers (*C92*).

2.4.3.12 Reuse and Maintainability

In our gathered literature maintainability is used as a motivation for modularization and reuse concepts. **P29**,**P60** and **P95** all claim that reuse mechanisms are necessary to keep model transformations maintainable. Combined with a total of eight (**P4**, **P10**, **P29**, **P33**, **P41**, **P60**, **P95**, **P78**) publications that state that reuse is hardly, if at all, established in current model transformation languages, this paints a bleak picture for both maintainability and reuse. The need for reuse mechanisms has already been recognized in the research community as stated by **P77** in which the authors explain that a plethora of mechanisms have been introduced (*C95*) but are hindered by several barriers such as insufficient abstraction from meta-models and platform or missing repositories of reusable artefacts (*C103*).

There exists only a single claim that directly addresses maintainability. **P44** state that bidirectional model transformation languages have an advantage when it comes to maintenance (*C94*).

Apart from the maintainability of written code, there is also the maintainability of languages and their ecosystems. Surprisingly, this is hardly discussed in literature at all. Only **P52** explains that evolving and maintaining a model transformation language is difficult and time consuming (*C101*).

2.4.3.13 Semantics and Verification

Three publications (**P39**, **P23**, **P58**) all suggest that most model transformation languages do not have well defined semantics which in turn makes verification and verification support difficult (**P22** *C109*). **P44** however explains that bidirectional transformations are advantageous with regards to verification (*C107*).

2.4.3.14 Tool support

Tools are another important aspect in the MDE life-cycle according to Hailpern et al. (2006). They are essential for efficient transformation development. Regrettably, MTLs lack good tool support according to **P23**, **P45**, **P52** and **P80** and if tools exist, they are not close to as mature as those of general-purpose languages as stated in **P74** (*C119*). Additionally, the authors of **P94** explain that developers of MTLs need to put extra effort into the creation of tool support for the language (*C121*). This might however be worthwhile, because **P44** presumes that dedicated tools for model transformation languages have the potential to be more powerful than tools for GPLs in the context of transformations (*C114*). And due to the high analysability of MTLs, **P45** explains, that tool support could potentially thrive (*C115*). Internal MTLs, on the other hand, are able to inherit tool support form their host languages as reported by **P23** (*C113*). This helps to mitigate the overall lack of tool support, at least for internal MTLs.

An interesting discussion to be held, is how important tool support for the acceptance of MTLs actually is. Whittle et al. (2013) describe that organizational effects are far more impactful on the adoption of MDE, while the results of Burgueño et al. (2019) contradict this observation citing interviewees from commercial tool vendors that stopped the development of tools due to lack of customer interest.

2.4.3.15 Versatility

It should be self-evident that languages that are designed for a special purpose do not possess the same level of versatility and area of applicability than general-purpose languages. Hence, it is not surprising that all mentions of versatility of model transformation languages in our gathered literature paint MTLs as less versatile compared to GPLs (**P52** (*C124*), **P80** (*C125*), **P94** (*C127*)).

2.4.4 RQ2: Supporting evidence for Advantages and Disadvantages of MTLs

We found a number of different ways used by authors of our gathered literature to support their assertions. The largest portion of 'supporting evidence' is made up of cited literature, i.e., a claim is followed by a citation that supposedly supports the claim.

The second way claims are supported is by example, i.e., authors implemented transformations in MTLs and/or GPLs and reported on their findings. Another aspect of this is relying on experience, i.e. authors state that from experience it is clear that some pronouncement is true or that it is a well established fact within the community that a claim is true.

Third, there is empirical evidence, i.e., studies designed to measure specific effects of model transformation languages or case studies designed to gather the state of MTL usage in industry.

Last, there are those assertions that are not supported by any means. Authors simply suggest that an advantage or disadvantage exists. We assume that some claims made in this way implicitly rely on experience but do not state so. Nevertheless, since there is no way of testing this assumption we have to record such claims exactly the way they are made, without any evidence.

In the following sections, we will talk in detail about how each group of evidence is used in literature to support claims about advantages or disadvantages of model transformation languages. As mentioned previously Table A.1 contains a complete overview over each claim and through what evidence the claim is supported.

2.4.4.1 Citation as evidence

Using citations to support statements is a core principle in research. It should therefore come as no surprise that citations are used to support claims about model transformation languages. An interesting aspect to explore for us was to trace how the cited literature supports the claim. For that, as stated in Sect. 2.3, we created a graphical representation to trace citations used as evidence through literature. The graph can be found in Fig. 2.7. It is inspired by UML syntax for object diagrams. The head of an 'object' contains a *publication id* while the body contains the categories for which advantages *(+)* or disadvantages *(-)* are claimed in the publication. Each category within the body is accompanied by an ID which can be used to find the corresponding claim within Table A.1. We use different *borders* around publications to denote the type of evidence provided by the publication and *arrows* from one category within a publication to a different publication stand for the use of a citation to support a claim. Lastly, if the content of a publication does not concern itself with model transformation languages but instead with DSLs the publication id is followed by '*(DSL)*'.

Our graph allows to easily gauge information about the following things:

- What publication claims an advantage or disadvantage of MTLs in which category?
- What type of evidence (if any) is used to support claims in a publication?
- Which exact claims are supported through the citation of what publication?

In the following we discuss observations about citations as evidence that can be made with help from the citation graphs.

First, only a total of 25 citations, split among 12 out of the 58 gathered publications, are used to support claims. This constitutes less than ten percent of all assertions found during our literature review. 7 of the 25 citations cite a publication that itself only states claims without any evidence thereof (**P63**, **P94**, **P673**, **P674**, **P800**). A further 11 end in a publication that uses examples or experience (see also Sect. 2.4.4.3) (**P664**, **P665**, **P667**, **P671**, **P672**, **P676**, **P77**, **P64**, **P804**, **P801**). Next there are 3 citations that cite publications which in turn cite further publications to support their claims (**P677**, **P675**), leaving only 4 citations that cite empirical studies (**P669**, **P670**, **P803**) (see also Sect. 2.4.4.2). To us this is worrying because the practice of citing literature that only restates a assertion corrodes the confidence readers can have in citations as supporting evidence.

From the graph it is clearly evident that there exists no single cited source for claims about model transformation languages. This is clearly indicated by the fact that only five publications (**P63**, **P77**, **P673**, **P675**, **P803**) are cited more than once. Twice to be exact. And no publication is cited more than two times. Moreover, of those five publications **P675** and **P803** are each cited by a single publication respectively. **P675** is cited twice by **P80** and **P803** by **P675**. Related thereto, nearly each claim, even within the same category, is being supported through different citations.

Furthermore, only claims about *conciseness*, *expressiveness*, *reuse & maintainability*, *tool support*, *performance* and statements that MTLs are *just better* are supported using citations. It is interesting to note that claims within these categories which are supported by citations, are either all positive or all negative. This is not to say that there are no contrasting claims, see for example *C113* and *C116* in **P23**, only that, if citations are used for a category the supported claims are either all positive or all negative.

Another thing to note is that in some instances claims about model transformation languages are being supported by citing publications on domain specific languages in general. This can be seen in **P80**. The claims *C60* and *C61* are both supported by a citation of **P675** which is a publication that concerns itself with DSLs. Interestingly **P675** itself then cites both publications about DSLs (**P800**, **P801**, **803**) and a publication about model transformation languages (**P804**) to support claims stated within the publication.

Coming back to citations of empirical studies we have to report that while there exist 4 citations of empirical studies only a single claim about model transformation languages (*C116* in **P23**) is actually supported thereby. This is due to **P803** being an empirical study about DSLs and **P669** and **P670** both being cited as evidence for *C116*.

Lastly, apart from those publications that only make a single claim, no publication supports all their claims using citations. Extreme cases of this can be seen in **P45** and **P52** which make a total of 16 claims, only supporting three of them with citations while leaving the other 13 unsubstantiated.

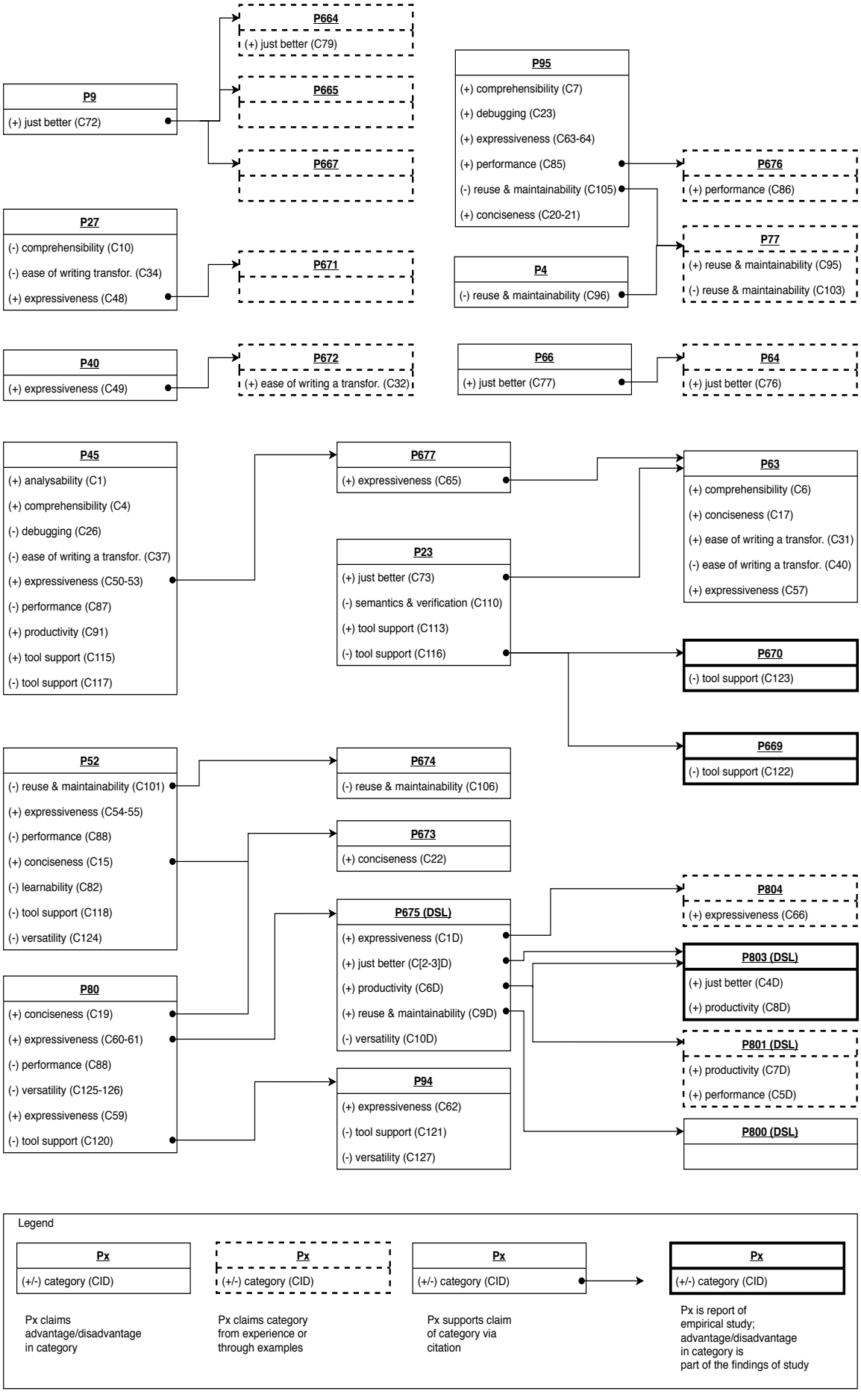


FIGURE 2.7: Graph tracking citations of claims of various categories through literature.

2.4.4.2 Empirical evidence

To our disappointment we have to report a lack of overall empirical evidence for properties of model transformation languages. Only four publications (**P32**, **P59**, **P669**, **P670**) in our gathered literature assess characteristics of model transformations using empirical means (see Fig. 2.7 and Table A.1). Of those four only **P59** focuses on MTLs as its central research object while the other three are case studies about MDA that happen to contain results about transformation languages. **P803** too is an empirical study, but as mentioned in Sect. 2.4.4.1 focuses on domain specific languages in general not on MTLs. In order to provide the necessary context for scrutinizing the claims extracted from the publications we provide a short overview over the central aspects of **P32**, **P59**, **P669**, **P670** in the following.

The study described in **P59** was comprised of a large-scale controlled experiment with over 78 subjects from two universities as well as a preliminary study with a single individual. Subjects had to solve 231 tasks using three different languages (ATL, QVT-O and Xtend). The tasks focused on one of three aspects in transformation development, namely comprehending an existing transformation, changing a transformation and creating a transformation from scratch. After analysing the results, the authors come to the disillusioning conclusion, that there is "no statistically significant benefit of using a dedicated transformation language over a modern general-purpose language".

The authors of **P32** report on an empirical study on the efficiency and effectiveness of MDA. A total of 38 subjects, selected from a model-driven engineering course, were asked to implement the book-purchasing functionality of an e-book store system. Afterwards the subjects evaluated the perceived efficiency and effectiveness of the used methodology. This also included questions about the used QVT language which was perceived as only marginally efficient.

Both **P669** and **670** are reports of industrial case studies. The objective of the study in **P669** was to investigate the state of practice of applying MDSE in industry. To achieve this, they collected data from tool evaluations, interviews and a survey. Four different companies were consulted to collect the data. Again while some reported results concerned themselves with transformations, model transformation languages were not explicitly discussed. Similarly, **P670** reports on an industrial case study involving two companies aiming to collect factors that influence the decision to adopt MDE. For that purpose multiple pre-selected individuals at both companies were interviewed. Just as **P669** the study did not directly focus on transformations or transformation languages.

As evident from Fig. 2.7, the results from **P32** and **P59** have yet to be used in literature for supporting claims about MTLs. Since both of them have only been published recently we are however optimistic about this prospect.

2.4.4.3 Evidence by example/experience

Using examples to demonstrate shortcomings of any kind has a long standing tradition not only in informatics. Using examples to demonstrate an advantage however, can result in less robust claims (especially toy or textbook examples Shaw (2003)). As such it is important to differentiate whether a claim is made by demonstrating a shortcoming or benefit.

In our gathered literature, ten publications use examples to support a claim. Interestingly, examples are mainly used to support broad claims about model transformation languages. This can be observed by the fact that **P34** and **P64** use examples to try and demonstrate that GPLs are not well suited for transforming models while **P664**, **P665**, **P667**, **P672**, **P804** and **P676** try to demonstrate the general superiority of MTLs by showing examples of transformations written in MTLs. Other claims that are supported through examples are a demonstration of the reduction in code size when using rule based MTLs in **P59** and statements about the extensive amount of reuse mechanisms for MTLs through listing gathered publications about proposed mechanisms in **P77**.

Long time practitioners of model transformation languages or programming languages in general often rely on their experience to make assertions about aspects of the language. And while the experience of long term users can create valuable insights it is still subjective and can therefore vary in accuracy. In our case six publications directly state that their assertions come from experience. **P3** report on their experiences using different languages to implement transformations, coming to the conclusion that graphical rule definition is more intuitive. An experience shared by **P40**. **P43** name user feedback as grounds for claiming that visual syntax has advantages in comprehension but makes writing transformations more difficult. And **P672** share that they are under the impression that graph transformations are the superior method for defining refactorings.

Since experience is subjective contradicting experiences are bound to occur sometime. While the authors of **P10** believe from experience that current MTLs are not abstract enough for expressing transformations **P671** feel that the difficulty of writing transformations in a MTL does stem from the chosen MDD method rather than the syntax of the language.

2.4.4.4 No evidence

Fig. 2.7 and especially Table A.1 make it clear that a large portion of both positive and negative claims about model transformations are never substantiated. In fact of the 127 claims ~69% are unsubstantiated. Adding those that are supported by a citation that in the end turns out to be unsupported as well, brings the number up to ~77%. Particularly the categories concerning the usability of MTLs such as *comprehensibility*, *ease of writing a transformation* and *productivity* lack meaningful evidence. All three of them being cornerstones of language engineers arguments for the superiority of model transformation languages makes this especially worrisome.

We believe that a realization in the community about this fact is necessary. The necessity or superiority of model transformations has to be properly motivated. This means that it is not sufficient to claim advantages or disadvantages without providing at least some form of explanation on why this claim is valid (more on this in Sect. 2.5.3).

2.5 Discussion

In this section we reflect on the previously presented findings. Our focus for this is fourfold. First we feel it is necessary to draw parallels between our categorization and attributes of product quality. Next we want to briefly discuss how claims are made in regards to transformation language features. Afterwards a discussion about lack of empirical studies about properties of model transformation languages is warranted. And last we feel a discussion about the research direction for the community is also necessary.

2.5.1 Claims about model transformation languages in context of software quality

There are undeniable parallels between the categories we developed for claims and characteristics of software quality as defined by *ISO/IEC 25010:2011* (2011). This can be seen by the fact that many of our categories can be directly placed within the characteristics of the software product quality model (namely Functional Suitability, Performance Efficiency, Compatibility, Usability, Reliability, Security, Maintainability, Portability).

Both *expressiveness* and *semantics and verification* are part of Functional Suitability. *Performance* and *productivity* can be classified under Performance Efficiency. Furthermore are *comprehensibility*, *conciseness*, *debugging*, *ease of writing a transformation*, *learnability* and *tool support* part of *Usability*. Maintainability covers *analysability* and *reuse & maintainability*. And lastly *extendability* and *versatility* can be classified under Portability. This leaves only our generic category *just better* without a corresponding characteristic which is to be expected.

However there are also Compatibility, Reliability and Security which have no corresponding categories from our categorisation. This does not necessarily mean that current research is not focused on aspects related to these quality criteria. It instead suggest a lack of concrete statements regarding them. And while Security is justifiably less of a concern for model transformation languages, both the Compatibility of different approaches and their Reliability should definitely be focused on (see also Sect. 2.5.4).

Lastly, even though most claims we collected during our review could be categorised within the software product quality model we opted to develop a classification based on the claims alone since we believe the resulting categories to more specialised and allow for a more nuanced view on the subject matter than the generic characteristics defined by *ISO/IEC 25010:2011* (2011).

2.5.2 Claims about model transformation languages in context of language features

An effort by us to categorize the extracted claims along an existing taxonomy of model transformation language features such as the one by Czarnecki et al. (2006) failed because a large portion of claims (~70%) are made broadly without reference to specific features of MTLs that aid the advantage or disadvantage.

We suggest that claims on benefits and disadvantages of model transformation languages be made more specific and include mentions of the features that aid or hamper the benefits. For example incrementality aids the performance of model transformations since only parts of a transformation have to be re-executed and bidirectional transformation languages provide special support for incremental execution giving them an edge in performance.

2.5.3 Lack of evidence for MTL advantages and disadvantages

Current literature exhibits a deficit in evidence (empirical or otherwise) for asserted properties of model transformation languages. We believe there to be several factors which can explain this lack of evidence.

First, designing and conducting rigorous studies to examine model transformation languages requires a substantial amount of time and effort. Studies are further complicated by the lack of easily available study subjects due to the community being relatively small compared to the body of general purpose programming language users. The study described in **P59**, for example, had to be conducted over the timespan of three semesters and at two universities just to attain 78 subjects. And even when a pertinent number of study subjects is found, ensuring comparable levels of experience within the subjects is another challenge, even more so when collaborating with industrial partners (Sjoberg et al. 2002).

Relying on the fact that transformation languages are DSLs and hence bear all the benefits that are proclaimed for those, might also be a factor. Describing the advantages of DSLs in the introduction of a paper about transformation languages is far from uncommon in literature. And while we too believe that there are benefits when using DSLs, we would caution against broad usage of the fact that model transformation languages are DSLs to claim them advantageous over general purpose languages (as is done in publications such as **P29**, **P63** or **P804**). Especially, because the manpower that goes into the development of the ecosystems of GPLs far exceeds that of MTLs.

Another problem is, that statements can become 'established' facts by virtue of being cited by a paper which is in turn cited. Suppose one author claims that model transformation languages are more expressive than GPLs. A second author claims the same thing and references the first author to provide context. Next a third author, assuming the second author verifies their claim via the citation, cites the second author to support a similar claim. Over time this can lead to the statement being treated as a fact rather than an assumption made multiple times. This can be seen on multiple occasions in Fig. 2.7. **P63** makes an unsubstantiated claim (*C57*) that the expressiveness of MTLs is superior to that of GPLs. This claim is then reiterated by **P677** (*C65*) citing **P67**. Lastly **P677** is cited by **P45** to support their assertion about the expressiveness of model transformation languages (*C50-53*). Such a chain is not even the worst case in our results. The chain **P80** → **P675** → **P801-804** is even more worrisome, in that some of the claims stated in **P80** (*C75*) actually originate in claims about domain specific languages from **675** (*C1D*). **P80** claims two advantages of MTLs using **P675** as reference. **P675** again uses citations to support their claims. However, the papers cited by **P675** do not make statements about model transformation languages but DSLs in general. This shows how such chains can create a blurred factual picture. Moreover, in the presented cases it is still possible to find the origin of claims and realize how the claims were changed throughout the citation chains. If authors deemed it unnecessary to support claims that are 'established' facts this is no longer possible. Quite likely this is the case for a non-negligible number of publications (see Table A.1) where no citations or any other substantiation for claimed properties of MTLs are given.

As previously described, it is not uncommon for authors to ascribe properties to model transformation languages due to them being DSLs. However, a language does not necessarily have to be more expressive, easier to use or easier to maintain simply by being domain specific. In fact we believe, that everything about a DSL stands and falls with the domain itself as well as the design of the language. As a result all advantages and disadvantages for DSLs, described in literature, only

define potential properties. Thus it is necessary to evaluate advantages and disadvantages anew for each domain. Especially in complex domains such as model transformations.

2.5.4 Research direction

In our opinion the research community has to acknowledge that the current way of language development is not expedient. There needs to be a shift away from constant development of new features and transformation languages with, at best, prototypical evaluation. Tomaž Kosar et al. (2016) share this sentiment after a mapping study on the development of DSLs in general (see Sect. 2.6).

Instead it is necessary to extensively evaluate current transformation languages. First, to identify their actual strengths and weaknesses. Then to compare these results with the expected (and desired) results to determine which aspects of MTLs still need improving.

We believe the categories from Sect. 2.4 to be a good reference for possible areas to evaluate.

It is not necessary to evaluate each category empirically: for some categories empirical evaluation might not be sensible at all. Such categories include analysability, and semantics and verification for example, since there exist no universally accepted measures to base evaluation on. Additional literature reviews are also conceivable. Analogous to how **P77** gathered different reuse mechanisms, a comprehensive review of verification and analysis approaches can be useful to assess the *analysability* and *verifiability* of model transformation languages.

Designing and executing appropriate studies also entails significant effort which is why it becomes necessary to carefully weigh up which properties should be evaluated. Additionally, some categories should also be examined more urgently than others.

The *ease of writing a transformation* and *comprehensibility* are two such categories for which evaluation is most pressing. Also given that in the domain of programming languages (especially object oriented programming) many studies exploring the comprehensibility and ease of use such as Burkhardt et al. (2002), Kurniawan et al. (2004), and Rein et al. (2019) already exist. Study designs similar to the one described in **P59** are in our opinion most suitable for this purpose. This is supported by the fact that many studies for comparing programming languages follow a similar structure in that a common problem or task is solved in multiple languages and the resulting code is analysed (Aruoba et al. 2014; Henderson et al. 1994; Prechelt 2000). It may also be useful to design the cases in such a way that the complete capabilities of the used transformation languages have to be used. In the study described in **P59** for example advanced features such as QVTs *late resolve* were not part of the evaluation. Such a design can help to better understand if the most 'advanced' features of transformation languages have practical value and how complex a GPL for these features is.

Comprehensibility can also be tested in isolation by requiring subjects to describe functionality of given transformations written in both a dedicated model transformation language and a GPL.

According to Mohagheghi et al. (2013a) one of the main motivations for adopting MDE in industry is to improve *productivity*, hence we believe that evaluation of the productivity when using model transformation languages should be a focus too. Admittedly measuring productivity is a challenging task, a fact that has been observed as early as 1978 (T. C. Jones 1978). But since then numerous ways have been proposed and tested out in practice (V. Basili et al. 1979; Boehm et al. 1995) which should allow for productivity studies on MTLs to be carried out. A potential study into the productivity could require subjects to develop transformations in either a model transformation language or a general purpose language within a certain time frame followed by measuring and comparing how productive the subjects were in both cases. Researchers can also draw from the large corpus of productivity studies on different aspects of programming such as Dieste et al. (2017), Frakes et al. (2001), and Wiger et al. (2001).

The *performance* of model transformations can have huge impact on development, especially when multiple transformations have to be executed in succession. Many language engineers already pay tribute to that fact by providing performance comparisons between their languages and other MTLs or general purpose languages such as Java (Hinkel et al. 2019a; Křikava et al. 2014). And the Transformation Tool Contest (TTC) provides a venue for comparing MTLs. However, we believe extensive comparisons between the performance of model transformation languages and general-purpose programming languages to be necessary to abolish the prejudice that dedicated transformation languages cannot outperform current compilers. Comparing of performance between different programming languages that are used for the same purpose is a well established practice demonstrated by comparisons between Java and C++ for robotics programming done by Gherardi

et al. (2012) or C++ and F90 for scientific programming by Cary et al. (1997). Performance comparisons are also common practice in other domains such as GPU programming where specialised DSLs are used and performance is of high importance (Fang et al. 2011; Karimi et al. 2010). It is conceivable to compare the performance of transformations written in dedicated MTLs and GPLs by either manually solving the same tasks as described previously or by using existing mechanisms (such as for example Calvar et al. (2019)) for transforming transformation scripts written in a MTL into GPL code.

We also believe that special focus needs to be given to the question of what model transformation languages are expected to achieve (such as easy synchronization of multiple artefacts or fast transformations through incremental transformations). First, because this can allow to direct more resources on evaluating relevant aspects of MTLs. And second, because model transformation languages will appear more streamlined and mature when the focus of development lies in improving their core features instead of overloading them with 'experimental' features. An opinion Tomaž Kosar et al. (2016) share, stating that this can enable practitioners to truly understand the effectiveness and efficiencies of DSLs.

2.6 Related Work

To the best of our knowledge, there exists no other literature review that explores advantages and disadvantages of model transformation languages. There does, however, exist some literature that can be related to our work.

A closely related survey and open discussion about the future of model transformation languages was held by Burgueño et al. (2019). They report on the results of an online survey and subsequent open discussion during the 12th edition of the International Conference on Model Transformations (ICMT'2019). The survey was designed to gather information about why developers used MTLs or why they hesitate to do so and what their predictions about the future of these languages were. An open discussion was held after the results of the online survey were presented to the audience at ICMT'2019. The results of the study point toward MTLs becoming less popular not only because of technical issues but also due to tooling and social issues as well as the fact that some GPLs have assimilated ideas from MTLs and thus making them less bad alternatives to writing transformations in dedicated languages.

Hutchinson et al. (2011a) conducted an empirical study into MDSE in industry. The authors used questionnaires and interviews to explore different factors that influence the success of MDSE in organizations and attempt to provide empirical evidence for hailed benefits of MDSE. They report on a total of over 250 questionnaire responses as well as interviews with 22 practitioners from 17 different companies. While the main focus of the study was on MDSE adoption in general, the authors do report on some findings regarding model transformations, such as negative influences of writing and testing transformations on the productivity and influences of transformations on the portability. However, no results regarding used transformation languages are included.

Mens et al. (2006) propose a taxonomy for model transformation languages. They define groups of transformation languages based on answers to a set of questions. The answers are split into multiple subgroups themselves. The authors describe in great detail different possible characteristics within the groups. In part, this also includes listings of properties for transformation languages that fall into specific groups. The authors, however, have not provided any evidence or more precise explanations. Similarly Czarnecki et al. (2006) propose a classification framework for model transformation approaches based on several approaches such as VIATRA, ATL and QVT. The framework is given as a feature diagram to allow to explicitly highlight different design choices for transformations. At the top level the feature model contains features such as rule organization, incrementality, directionality and tracing. Each feature and its sub-components are extensively discussed and demonstrated with examples of transformation tools that boast different aspects of the features. In contrast to the two described classifications our study categorizes claims about MTLs on a qualitative dimension rather than on language features.

Kahani et al. (2019) describe a classification and comparison of a total of 60 model transformation tools. Their classification differentiates tools based on two levels. The first level describes whether the tool is a model-to-model (M2M) or model-to-text (M2T) tool. The second level differentiates M2M tools based on their transformation approach meaning whether the approach is

relational, operational or graph-based and M2T tools based on the underlying implementation approach meaning visitor-based, template-based or hybrid. Unlike our study the described comparison focuses on comparing different model transformation tools on a technical basis based on six categories (general, model-level, transformation, user experience, collaboration support and runtime requirements) while we focus on qualitative aspects of claims made throughout literature about any kind of dedicated model transformation language.

Van Deursen et al. (2000) gathered an annotated bibliography on the premise of *domain-specific languages versus generic programming languages*. The bibliography contains 73 different DSLs differentiated by their application domains: *Software Engineering*, *Systems Software*, *Multi-Media*, *Telecommunication* and *Miscellaneous*. Additionally they provide a discussion of terminology as well as risks and benefits of DSLs. And while parts of the listed risks an benefits such as enhanced productivity or cost of education can be found in the listed advantages and disadvantages of our literature review, their bibliography does not contain any model transformation languages.

Tomaž Kosar et al. (2016) report on the results of a systematic mapping study they conducted to understand the DSL research field, identify research trends and to detect open issues. Their data comprised a total of 1153 candidates which they condensed into 390 publications for classification. The results from the study corroborate observations made during our literature review. The research community is mainly concerned with the development of new techniques while research into the effectiveness of languages and empirical evaluations is lacking.

Tomaz Kosar et al. (2010) describe an empirical study comparing a domain specific language with a general purpose language with a focus on learning, perceiving and evolving programs. The two languages considered were XAML as a DSL representative and the GPL C#. The experiment comprised of 36 programmers which were asked to construct a graphical interface using both XAML and C# Forms. Afterwards the subjects had to answer a questionnaire. In contrast to the results of **P59** their results show a statistically significant advantage of DSLs for learning, comprehending and evolving programs.

Jakumeit et al. (2014) provide an extensive overview over and comparison of 13 state of the art transformation tools used in the TTC 2011. The authors give detailed descriptions of the tools based on a 'Hello World' case posed at the contest. They also describe for what use cases the individual tools are best suited and provide a novel taxonomy based on which the tools are compared. The introduced taxonomy features many of the same categories we synthesized from the claims in our literature review such as expressiveness, extendability, learnability, reuse and verification but also other categories such as maturity and license.

2.7 Threats to validity

To ensure reproducibility and a high quality of the results, we followed a systematic approach as detailed in Sect. 2.3. However, possible threats to validity still remain. In this section we discuss these threats.

2.7.1 Internal Validity

Internal validity describes the extent to which a casual conclusion based on the study is warranted. This validity is threatened by possible differences in the interpretation of our selection criteria. To alleviate the potential threat two researchers independently applied the selection criteria and in cases of different decisions about the inclusion of a publication, full text cross reading was applied.

A threat to the internal validity we could not meet with prevention measures was the fact that our categorization is based on certain defining expressions like *'expressive'* and *'versatile'*. It is possible that different authors ascribe different meanings to these phrases. While we believe that for most cases this is less of a problem it is still a problem that we could not fully solve since not every publication defines their understanding of used phrases.

2.7.2 External Validity

External validity describes the extent to which the findings of a study can be generalized. For structured literature reviews a threat to this validity arises from the existence of relevant but undetected or excluded publications (Ciccozzi et al. 2019). To mitigate this threat as much as possible we used both automatic searches and exhaustive backward and forward snowballing. The automatic search

was also conducted on multiple literature databases to broaden the field of searched literature. Furthermore we employed a 'when uncertain include' strategy for including publications, as well as less strict inclusion criteria which helped prevent relevant publications from being overlooked.

2.7.3 Construct Validity

Construct validity describes the extent to which the right measures were obtained and whether the right scope was defined in relation to our research questions. The construct validity of our research is not under threat since the research questions define easily producible results. Cited advantages or disadvantages of model transformation languages can be directly extracted and the same also holds for used evidence for claims.

2.7.4 Conclusion Validity

Conclusion validity describes the extent to which conclusions based on data are reproducible.

Prior to the execution of our literature review we defined a review protocol for all phases of the review. We followed the protocol rigorously to ensure reproducibility of the study. The protocol did not only include descriptions of how the review had to be conducted but also detailed how data should be extracted from the selected literature (see Sect. 2.3). It is of course possible that, with the passage of time, a repetition of the literature review can draw different conclusions due to the added body of literature between then and now.

2.8 Conclusion

In this study, we have reported on a systematic literature review intended to extract and categorize claims about model transformation languages as well as the current state of evaluation thereof. The goal of the study was to compile a comprehensive list and the categorization of positive and negative claims about model transformation languages. We further wanted to investigate the current state of evaluation of claims as well as identify gaps in the area of evaluation of MTLs.

We combed over 4000 publications for that purpose, 58 of which we selected for the study. To this end, we followed a rigorous process by using a combination of automatic searches on literature databases, exhaustive backward and forward snowballing and multiple researchers during the selection phase. The selected publications were combed for mentions of advantages and disadvantages of MTLs and evidence of the stated claims. Lastly, we analysed and discussed the extracted claims and evidence to: (i) provide an overview over claimed advantages and disadvantages and their origin, (ii) the current state of evidence thereof and (iii) identify areas where further research is necessary.

We conclude that: (i) current literature claims many advantages of MTLs but also points towards deficits owed to the mostly experimental nature of the languages and its limited domain, (ii) there is insufficient evidence for and (iii) research about properties of model transformation languages.

The results of our study suggest that there is much to be done in terms of evaluation of model transformation languages and that effort that is currently being invested into the development of new features might be better spent evaluating the state of the art in hopes of ascertaining both what current MTLs are lacking most and where their strengths really lie.

Chapter 3

Paper B

Advantages and disadvantages of (dedicated) model transformation languages: A Qualitative Interview Study

S. Höppner, Y. Haas, M. Tichy, K. Juhnke

Empirical Software Engineering (EMSE), volume 27, article number 159, 2022
Springer Nature

Abstract

Context
Model driven development envisages the use of model transformations to evolve models. Model transformation languages, developed for this task, are touted with many benefits over general purpose programming languages. However, a large number of these claims have not yet been substantiated. They are also made without the context necessary to be able to critically assess their merit or built meaningful empirical studies around them.

Objective
The objective of our work is to elicit the reasoning, influences and background knowledge that lead people to assume benefits or drawbacks of model transformation languages.

Method
We conducted a large-scale interview study involving 56 participants from research and industry. Interviewees were presented with claims about model transformation languages and were asked to provide reasons for their assessment thereof. We qualitatively analysed the responses to find factors that influence the properties of model transformation languages as well as explanations as to how exactly they do so.

Results
Our interviews show, that general purpose expressiveness of GPLs, domain specific capabilities of MTLs as well as tooling all have strong influences on how people view properties of model transformation languages. Moreover, the Choice of MTL, the Use Case for which a transformation should be developed as well as the Skills of involved stakeholders have a moderating effect on the influences, by changing the context to consider.

Conclusion
There is a broad body of experience, that suggests positive and negative influences for properties of MTLs. Our data suggests, that much needs to be done in order to convey the viability of model transformation languages. Efforts to provide more empirical substance need to be undergone and lacklustre language capabilities and tooling need to be improved upon. We suggest several approaches for this that can be based on the results of the presented study.

3.1 Introduction

Model transformations are at the heart of model-driven engineering (MDE) (Metzger 2005; Sendall et al. 2003). They provide a way to consistently and automatically derive a multitude of artefacts such as source code, simulation inputs or different views from system models (Schmidt 2006). Model transformations also allow to analyse system aspects on the basis of models (Schmidt 2006) and can provide interoperability between different modelling languages, e.g. architecture description languages like those described by Malavolta et al. (2010). Since the emergence of the MDE paradigm at the beginning of the century numerous dedicated model transformation languages (MTLs) have been developed to support engineers in their endeavours (Arendt et al. 2010; Jouault et al. 2006; OMG 2016). Their appeal is driven by the promise of many advantages such as increased productivity, comprehensibility and domain specificity associated with using domain specific languages (Hermans et al. 2009; Johannes et al. 2009).

A recent literature study of us revealed, that, while a large number of such advantages and also disadvantages are claimed in literature, there exist only a few studies investigating to what extend these claims actually hold (Götz et al. 2021a). The study presents 15 properties of MTLs for which literature claims advantages or disadvantages. In this context, a claimed positive effect on one of the properties means an advantage whereas a negative influence means a disadvantage. The properties identified in the study are: *Analysability*, *Comprehensibility*, *Conciseness*, *Debugging*, *Ease of Writing* (a transformation), *Expressiveness*, *Extendability*, (being) *Just Better*, *Learnability*, *Performance*, *Productivity*, *Reuse & Maintainability*, *Tool Support*, *Semantics & Verification* and *Versatility*.

Our study also revealed, that most claims in literature are made broadly and without much explanation as to where the claim originates from (Götz et al. 2021a). Claims such as *"Model transformation languages make it easy to work with models."* (Liepiņš 2012), *"Declarative MTLs increase programmer productivity"* (Lawley et al. 2007) or *"Model transformation languages are more concise"* (Hinkel et al. 2019b) illustrate this. We assume that authors make such claims while having certain context and background in mind, but choose to omit it for unspecified reasons. Some likely reason for omission of the context are, that authors believe it to not be worth mentioning or to preserve space which is often sparse in publications.

Regardless of the concrete reasons, a result of this practice is a lack of cause and effect relations in the context of model transformation languages that explain both why and when certain advantages or disadvantages hold. Claims are thus easily dismissed based on anecdotal evidence. Furthermore, setting up proper evaluation is also difficult because the claims do not provide the necessary background to do so.

To close this gap, we executed a large-scale empirical study using semi-structured interviews. It involved a total of 56 researchers and practitioners in the field of model transformations. The **goal** of our study was to compile a comprehensive list of influences on properties of model transformation languages guided by the following research questions:

RQ 1: What are the factors that influence properties of model transformation languages?

RQ 2: How do the identified factors influence MTL properties?

To concentrate our efforts and best utilize all available resources, we decided to focus on 6 of the 15 properties of model transformation languages identified by us in the preceding SLR (Götz et al. 2021a). The 6 properties investigated in this study are: *Comprehensibility*, *Ease of Writing*, *(practical) Expressiveness*, *Productivity*, *Reuse and Maintainability* and *Tool support*. We have chosen these six because they all play a major role in providing reasons for the adoption of model transformation languages.

Interviewees were presented with a number of claims about MTLs from literature and asked to reveal their views on the matter, as well as assumptions and reasons that lead them to agree or disagree with the presented claims. We qualitatively analysed the interviews to understand the participants perceived influence factors and reasons for the advantages or disadvantages stated in the claims. The extracted data was then analysed to find commonalities between interviewees. This was done for single claims as well as for overarching factors and reasons that influence a variety of aspects of MTLs.

We present a comprehensive explanation of factors that, according to experts, play an essential role in the discussion of advantages and disadvantages of model transformation languages for the investigated properties. This is accompanied by a detailed exposition of **how** factors are relevant for the properties given above. Lastly, we discuss the most salient factors and argue actionable results for the community and further research.

As the first study of this type, we make the following contributions:

1. A comprehensive categorisation and listing of factors resulting in advantages or disadvantages of MTLs in the 6 properties studied.
2. A detailed description of why and how each identified factor exerts an influence on different properties.
3. Suggestions for how the presented information can be utilised to empirically investigate MTL properties.
4. Procedural proposals for improving current model transformation languages based on the presented data.

The results of our study show, that there is a large number of factors that influence properties of model transformation languages. There is also a number of factors on which this influence depends on, i.e. factors that have a moderation effect on the influence of other factors. These factors provide a solid basis that allows further studies to be conducted with more focus. They also enable precise decisions on where improvements and adjustments in or for model transformation languages can be made.

The remainder of this paper is structured as follows: Section 3.2 introduces model-driven engineering and model transformation languages, the context in which our study integrates. Section 3.3 will detail our methodology for preparing and conducting the interviews and the procedures used to analyse the data accumulated through the interviews. Afterwards Sect. 3.4 gives an overview over demographic data of our interview participants while Sect. 3.5 presents our code system and details the findings for each code based on the interviews and analysis thereof. In Sect. 3.6 we present overarching findings and in Sect. 3.7, we discuss actionable results that can be drawn from our stud that indicate avenues to focus on for the research community. Section 3.8 contains a detailed discussion of the validity threats of this research, and in Sect. 3.9 related efforts are presented. Lastly, Sect. 3.10 draws a conclusion for our research and proposes future work.

3.2 Background

This section will provide the necessary background for the context in which our study is integrated in.

3.2.1 Model-driven engineering

The *Model-Driven Architecture* (MDA) paradigm was first introduced by the Object Management Group in 2001 (OMG 2001). It forms the basis for an approach commonly referred to as *Model-driven development* (MDD) (Brown et al. 2005), introduced as means to cope with the ever growing complexity associated with software development. At the core of it lies the notion of using models as the central artefact for development. In essence this means, that models are used both to describe and reason about the problem domain as well as to develop solutions (Brown et al. 2005). An advantage ascribed to this approach that arises from the use of models in this way, is that they can be expressed with concepts closer to the related domain than when using regular programming languages (Selic 2003).

When fully utilized, MDD envisions automatic generation of executable solutions specialized from abstract models (Schmidt 2006; Selic 2003). To be able to achieve this, the structure of models needs to be known. This is achieved through so called meta-models which define the structure of models. The structure of meta-models themselves is then defined through meta-models of their own. For this setup, the OMG developed a modelling standard called *Meta-object Facility* (MOF) (OMG 2002) on the basis of which a number of modelling frameworks such as the *Eclipse Modelling Framework* (EMF) (Steinberg et al. 2008) and the *.NET Modelling Framework* (Hinkel 2016) have been developed.

3.2.2 Domain-specific languages

Domain-specific languages (DSLs) are languages designed with a notation that is tailored for a specific domain by focusing on relevant features of the domain (Van Deursen et al. 2002). In doing so DSLs aim to provide domain specific language constructs, that let developers feel like working directly with domain concepts thus increasing speed and ease of development (Sprinkle et al. 2009). Because of these potential advantages, a well defined DSL can provide a promising alternative to using general purpose tools for solving problems in a specific domain. Examples of this include languages such as *shell scripts* in Unix operating systems (Kernighan et al. 1984), *HTML* (Raggett et al. 1999) for designing web pages or AADL an architecture design language (SAEMobilus 2004).

3.2.3 Model transformation languages

The process of (automatically) transforming one model into another model of the same or different meta-model is called *model transformation* (MT). They are regarded as being at the heart of Model Driven Software Development (Metzger 2005; Sendall et al. 2003), thus making the process of developing them an integral part of MDD. Since the introduction of MDE at the beginning of the century, a plethora of domain specific languages for developing model transformations, so called model transformation languages (MTLs), have been developed (Arendt et al. 2010; Balogh et al. 2006; George et al. 2012; Hinkel et al. 2019a; Horn 2013; Jouault et al. 2006; Kolovos et al. 2008). Model transformation languages are DSLs designed to support developers in writing model transformations. For this purpose, they provide explicit language constructs for tasks involved in model transformations such as model matching. There are various features, such as directionality or rule organization (Czarnecki et al. 2006), by which model transformation languages can be distinguished. For the purpose of this paper, we will only be explaining those features that are relevant to our study and discussion in Sections 3.2.3.1 to 3.2.3.7. Table 3.1 provides an overview over the presented features.

Please refer to Czarnecki et al. (2006), Kahani et al. (2019), and Mens et al. (2006) for complete classification.

3.2.3.1 External and Internal transformation languages

Domain specific languages, and MTLs by extension, can be distinguished on whether they are embedded into another language, the so called host language, or whether they are fully independent languages that come with their own compiler or virtual machine.

Languages embedded in a host language are called *internal* languages. Prominent representatives among model transformation languages are *FunnyQT* (Horn 2013) a language embedded in Clojure, *NMF Synchronizations* and the *.NET transformation language* (Hinkel et al. 2019a) embedded in C#, and *RubyTL* (Jesús Sánchez Cuadrado et al. 2006) embedded in Ruby.

Fully independent languages are called *external* languages. Examples of external model transformation languages include one of the most widely known languages such as the *Atlas transformation language* (ATL) (Jouault et al. 2006), the graphical transformation language Henshin (Arendt et al. 2010) as well as a complete model transformation framework called VIATRA (Balogh et al. 2006).

3.2.3.2 Transformation Rules

Czarnecki et al. (2006) describe rules as being *"understood as a broad term that describes the smallest units of [a] transformation [definition]"*. Examples for transformation rules are the rules that make up transformation modules in ATL, but also functions, methods or procedures that implement a transformation from input elements to output elements.

The fundamental difference between model transformation languages and general-purpose languages that originates in this definition, lies in dedicated constructs that represent rules. The difference between a transformation rule and any other function, method or procedure is not clear cut when looking at GPLs. It can only be made based on the contents thereof. An example of this can be seen in Listing 3.1, which contains exemplary Java methods. Without detailed inspection of the two methods it is not apparent which method does some form of transformation and which does not.

In a MTL on the other hand transformation rules tend to be dedicated constructs within the language that allow a definition of a *mapping* between input and output (elements). The example

Table 3.1: MTL feature overview

Feature	Characteristic	Representative Language
Embeddedness	Internal	FunnyQT (Clojure), RubyTL (Ruby), NMF Synchronizations (C#)
	External	ATL, Henshin, QVT
Rules	Explicit Syntax Construct	ATL, Henshin, QVT
	Repurposed Syntax Construct	NMF Synchronizations (Classes), FunnyQT (Macros)
Location Determination	Automatic Traversal	ATL, QVT
	Pattern Matching	Henshin
Directionality	Unidirectional	ATL, QVT-O
	Bidirectional	QVT-R, NMF Synchronisations
Incrementality	Yes	NMF Synchronizations
	No	QVT-O
Tracing	Automatic	ATL, QVT
	Manual	NMF Synchronizations
Dedicated Model Navigation Syntax	Yes	ATL (OCL), QVT (OCL), Henshin (implicit in rules)
	No	NMF Synchronizations, FunnyQT, RubyTL

```java
public void methodExample(Member m) {
  System.out.println(m.getFirstName());
}
public void methodExample2(Member m) {
  Male target = new Male();
  target.setFullName(m.getFirstName() + " Smith");
  REGISTRY.register(target);
}
```

List. 3.1: Example Java methods

```
rule Member2Male {
  from
    s : Member (not s.isFemale())
  to
    t : Male (
      fullName <- s.firstName + ' Smith'
    )
}

rule Member2Female {
  from
    s : Member (s.isFemale())
  to
    t : Female (
      fullName = s.firstName + ' Smith'
      partner = s.companion
    )
}
```

List. 3.2: Example ATL rules

rules written in the model transformation language ATL in Listing 3.2 make this apparent. They define mappings between model elements of type `Member` and model elements of type `Male` as well as between `Member` and `Female` using *rules*, a dedicated language construct for defining transformation mappings. The transformation is a modified version of the well known Families2Persons transformation case (Anjorin et al. 2017).

3.2.3.3 Rule Application Control: Location Determination

Location determination describes the strategy that is applied for determining the elements within a model onto which a transformation rule should be applied (Czarnecki et al. 2006). Most model transformation languages such as ATL, Henshin, VIATRA or QVT (OMG 2016), rely on some form of *automatic traversal* strategy to determine where to apply rules.

We differentiate two forms of location determination, based on the kind of matching that takes place during traversal. There is the basic *automatic traversal* in languages such as ATL or QVT, where single elements are matched to which transformation rules are applied. The other form of location determination, used in languages like Henshin, is based on *pattern matching*, meaning a model- or graph-*pattern* is matched to which rules are applied. This does allow developers to define sub-graphs consisting of several model elements and references between them which are then manipulated by a rule.

The *automatic traversal* of ATL applied to the example from Listing 3.2 will result in the transformation engine automatically executing the `Member2Male` on all model elements of type `Member` where the function `isFemale()` returns `false` and the `Member2Female` on all other model elements of type `Member`.

The *pattern matching* of Henshin can be demonstrated using Figure 3.1, a modified version of the transformation examples by Krause et al. (2014). It describes a transformation that creates a couple connection between two actors that play in two films together. When the transformation

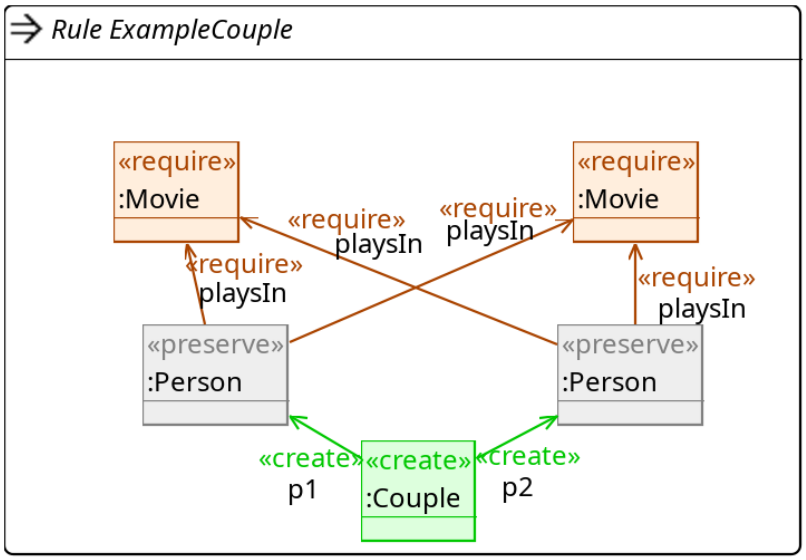


FIGURE 3.1: Example Henshin transformation

```
top relation Member2Male {
  n, fullName : String;
  domain Families s:Member {
    firstName = n };
  domain Persons t:Male {
    fullName = fullName};
  where {
    fullName = n + ' Smith'; };
}
```

LIST. 3.3: Example QVT-R relation

is executed the transformation engine will try and find instances of the defined graph pattern and apply the changes on the found matches.

This highlights the main difference between *automatic traversal* and *pattern matching* as the engine will search for a sub graph within the model instead of applying a rule to single elements within the model.

3.2.3.4 Directionality

The directionality of a model transformation describes whether it can be executed in one direction, called a unidirectional transformation or in multiple directions, called a multidirectional transformation (Czarnecki et al. 2006).

For the purpose of our study the distinction between unidirectional and bidirectional transformations is relevant. Some languages allow dedicated support for executing a transformation both ways based on only one transformation definition, while other require users to define transformation rules for both directions. General-purpose languages can not provide bidirectional support and also require both directions to be implemented explicitly.

The ATL transformation from Listing 3.2 defines a unidirectional transformation. Input and output are defined and the transformation can only be executed in that direction.

The QVT-R relation defined in Listing 3.3 is an example of a bidirectional transformation definition (For simplicity reasons the transformation omits the condition that males are only created from members that are not female). Instead of a declaration of input and output, it defines how two elements from different domains relate to one another. As a result given a `Member` element its corresponding `Male` elements can be inferred, and vice versa.

3.2.3.5 Incrementality

Incrementality of a transformation describes whether existing models can be updated based on changes in the source models without rerunning the complete transformation (Czarnecki et al. 2006). This feature is sometimes also called model synchronisation.

Providing incrementality for transformations requires active monitoring of input and/or output models as well as information which rules affect what parts of the models. When a change is detected the corresponding rules can then be executed. It can also require additional management tasks to be executed to keep models valid and consistent.

3.2.3.6 Tracing

According to Czarnecki et al. (2006) tracing *"is concerned with the mechanisms for recording different aspects of transformation execution, such as creating and maintaining trace links between source and target model elements"*.

Several model transformation languages, such as ATL and QVT have automated mechanisms for trace management. This means that traces are automatically created during runtime. Some of the trace information can be accessed through special syntax constructs while some of it is automatically resolved to provide seamless access to the target elements based on their sources.

An example of tracing in action can be seen in `line 16` of Listing 3.2. Here the `partner` attribute of a `Female` element that is being created, is assigned to `s.companion`. The `s.companion` reference points towards a element of type `Member` within the input model. When creating a `Female` or `Male` element from a `Member` element, the ATL engine will resolve this reference into the corresponding element, that was created from the referred `Member` element via either the `Member2Male` or `Member2Female` rule. ATL achieves this by automatically tracing which target model elements are created from which source model elements.

3.2.3.7 Dedicated Model Navigation Syntax

Languages or syntax constructs for navigating models is not part of any feature classification for model transformation languages. However, it was often discussed in our interviews and thus requires an explanation as to what interviewees refer to.

Languages such as OCL (OMG 2014), which is used in transformation languages like ATL, provide dedicated syntax for querying and navigating models. As such they provide syntactical constructs that aid users in navigation tasks. Different model transformation languages provide different syntax for this purpose. The aim is to provide specific syntax so users do not have to manually implement queries using loops or other general purpose constructs. OCL provides a functional approach for accumulating and querying data based on collections while Henshin uses graph patterns for expressing the relationship of sought-after model elements.

3.3 Methodology

To collect data for our research question, we decided on using semi-structured interviews and a subsequent qualitative content analysis that follows the guidelines detailed by Kuckartz (2014). Semi-structured interviews were chosen as a data collection method because they present a versatile approach to eliciting information from experts. They provide a framework that allows to get insights into opinions, thoughts and knowledge of experts (Hove et al. 2005; Kallio et al. 2016; Meyer et al. 1990). The qualitative content analysis guidelines by Kuckartz (2014) were chosen because of their detailed descriptions for all steps of the analysis process. As such they provide a more detailed and modern framework compared to the procedures introduced by Mayring (1994), which have long been a gold standard in qualitative content analysis.

An overview over our complete study design can be found in Fig. 3.2. It shows the order of activities that were planned and executed as well as the artefacts produced and used throughout the study. Each activity is annotated with the section number in which we detail the activity. We split our approach into three main-phases: *Preparation* (detailed in Section 3.3.1), *Operation* (detailed in Section 3.3.2) and *Coding & Analysis* (detailed in Section 3.3.3).

For the preparation phase, we used a subset of the claimed properties of model transformation languages identified by us (Götz et al. 2021a) to develop an interview guide. The guide focuses

around asking participants **whether** they agree with a claim from one of the properties and then envisages the usage of **why** questions to gain a deeper understanding of their opinions on the matter. After identifying and contacting participants based on the publications considered during our previous literature review (Götz et al. 2021a), we conducted 54 interviews with 55 interviewees (at the request of two participants, one interview was conducted with both of them together) and collected one additional written response. During the *Coding & Analysis* phase, we coded and analysed all 54 transcripts, as well as the written response, guided by the framework detailed by Kuckartz. In doing so, we focused first on factors and reasons for the individual properties and then on common factors and reasons between them.

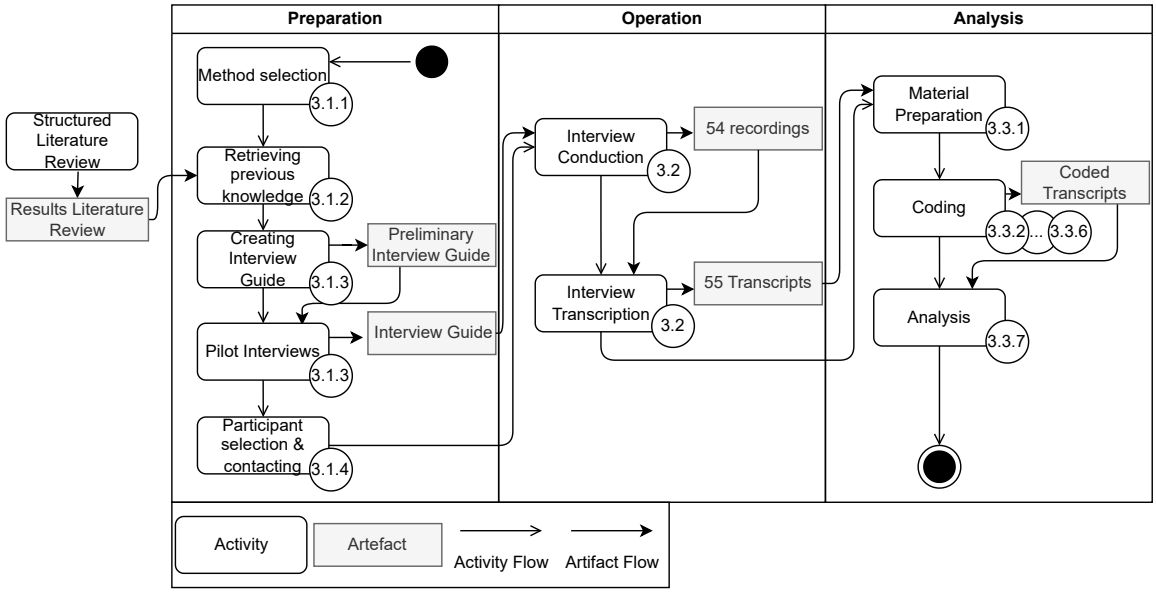


FIGURE 3.2: Overview over the study design

The remainder of this section will describe in detail how each of the three phases of our study was conducted.

3.3.1 Interview Preparation

Our interview preparation phase consists of the creation of an interview guide plus selecting and contacting appropriate interview participants. We use the guidelines by Kallio et al. (2016) for the creation of our interview guide and expand the steps detailed there with steps for selecting and contacting participants. In addition, we use the guidance from Newcomer et al. (2015) to construct our study in the best possible way.

According to Kallio et al. (Kallio et al. 2016) the creation of an interview guide consists of five consecutive steps. First, researchers are urged to evaluate how appropriate semi-structured interviews are as a data collection method for the posed research questions. Then, existing knowledge about the topic should be retrieved by means of a literature review. Based on the knowledge from the knowledge retrieval phase, a preliminary interview guide can then be formulated and in another step be pilot tested. Lastly the complete interview guide can then be presented and used. As previously stated, we enhance these steps with two additional steps for selecting and contacting potential interview participants.

In the following, we detail how the presented steps were executed and the results thereof.

3.3.1.1 Identifying the appropriateness of semi-structured interviews

The goal of our study, outlined in our research questions, is to collect and analyse reasons and background information of why people believe claims about model transformation languages to be true. Data such as this is qualitative by nature and hence requires a research method capable of producing qualitative data. According to Hove et al. (2005) and Meyer et al. (1990) expert interviews are one of the most widely used research methodologies in the technical field for this

purpose. They allow to ascertain qualitative data such as opinions and estimates. Interviews also enable qualitative interpretation of already available data (Meyer et al. 1990) which perfectly aligns with our goal. Moreover the opportunity to ask further questions about specific statements made by the participants (Newcomer et al. 2015) fits the open ended nature of our research question. For these reasons, we believe semi-structured interviews to be a well suited to answer our research questions.

3.3.1.2 Retrieving previous knowledge

In our previous publication (Götz et al. 2021a), we detailed the preparation, execution and results of an extensive structured literature review on the topic of claims about model transformation languages. The literature review resulted in a categorization of 127 claims into 15 different categories (i.e. properties of MTLs) namely *Analysability*, *Comprehensibility*, *Conciseness*, *Debugging*, *Ease of Writing a transformation*, *Expressiveness*, *Extendability*, *Just better*, *Learnability*, *Performance*, *Productivity*, *Reuse & Maintainability*, *Tool Support*, *Semantics and Verification* and lastly *Versatility*. These properties and the claims about them serve as the basis for the design of our interview study presented here.

3.3.1.3 Interview guide

The interview guide involves presenting each interview participant with several claims on model transformation languages. We use claims from literature instead of formulating our own statements, to make them more accessible. This also prevents any bias from the authors to be introduced at this step. Participants are first asked to assess their agreement with a claim before transitioning into a discussion on what the reasons for their decision are based on an open-ended question. This style of using close-ended questions as a prerequisite for open-ended or probe questions has been suggested by multiple guides (Hove et al. 2005; Newcomer et al. 2015).

We focus on a subset of six properties. This is due to the aim of keeping the length of interviews within an acceptable range for participants. According to Newcomer et al. (2015) semi-structured interviews should not exceed a maximum length of one hour. As a result, only a number of properties can be discussed per interview. In order to still talk with enough participants about each property, the number of properties examined must be reduced. The properties we discuss in the interviews and the reasons why they are relevant are as follows:

- *Comprehensibility*: Is an important property when transformations are being developed as part of a team effort or evolve over time.
- *Ease of Writing*: Is a decisive property that influences whether developers want to use a languages to write transformations in.
- *Expressiveness*: Is one of the most cited properties in literature (Götz et al. 2021a) and main selling point of domain specific languages in general.
- *Productivity*: Is a property that is highly relevant for industrial adoption.
- *Reuse & Maintainability*: Is another property that enables wider adoption of model transformation languages in project settings.
- *Tools*: High-quality tools can provide huge improvements to the development.

The list consists of the 5 most claimed properties form the previous literature review (Götz et al. 2021a) and is supplemented with *Productivity*, because we believe this attribute to be the most relevant for industry adoption.

To maximize the response rate of contacted persons, we aim for an interview length of 30 minutes. This decision is based on experiences from previous interview studies conducted at our research group (Groner et al. 2020; Juhnke et al. 2020) and fits within the maximum interview length suggested by Newcomer et al. (2015).

To best utilize the limited time per interview, the six properties are split into three sets of two properties each. In each interview one of the three sets is discussed.

For each property, one non-specific, one specific and one negative claim is used to structure all interviews involving this property around. A complete overview over all selected claims can be found in Table 3.2.

We consider non-specific claims to be those that do not provide any rationale as to why the claimed property holds, e.g. *"Model transformation languages ease the writing of model transformations."*. The non-specific claims chosen simply reflect the property itself. They serve the purpose of getting participants to state their assumptions and beliefs for the property without any influence exerted by the discussed claim.

We consider those claims as specific, that provide a rationale or reason for why the claimed property holds, e.g. *"Model transformation languages, being DSLs, improve the productivity."*. And we consider negative claims to be those, that state a negative property of model transformation languages, e.g. *"Model transformation languages lack sophisticated reuse mechanisms."*. Generally, we use claims where we believe the discussions about the reasons to provide useful insights.

There exist several reasons why we believe this setup of using the same three none-specific, specific and negative claims for each property to be appropriate. First, the non-specific claim allows participants to provide any and all factors or reasons that they believe influence a claimed property. The specific claim then allows us to introduce a reason, that participants might not have thought about. It also prompts a discussion about a particular reason or factor that is shared between all participants. This ensures at least one area for cross comparison between answers. The negative claim forces participants to also deliberate negative aspects, providing a counterbalance that counteracts bias. Furthermore, the non-specific claim provides an easy introduction into the discussion about a specific MTL property that can present the interviewer with an overview of the participants thoughts on the matter. It also allows participants to provide other influence factors not specifically covered through the discussed claims or even new factors and reasons not present in the collection of claims from our literature review (Götz et al. 2021a).

The complete interview guide resulting from the aforementioned considerations can be seen in Fig. 3.3. After introductory pleasantries we start all interviews of with demographic questions. Although some sources discourage asking demographic questions early in the interview due to their sensitive nature (Newcomer et al. 2015), we use them to break the ice between the interviewer and interviewee because our demographic questions do not probe any sensitive information.

After this initial get-to-know each other phase, the interviewer then proceeds to explain the research intentions, goals and the procedure of the remaining interview. Depending on the property-set selected for the interview, participants are then presented with a claim about a property. They are asked to rate their agreement with the claim based on a 5-point likert scale (5: completely agree, 4: agree, 3: neither agree nor disagree, 2: disagree, 1: completely disagree). The likert scale is used to allow the interviewer to better assess the participants tendency compared to a simple yes or no question. This part of the interview is intended solely to get a first impression of the view of the participant and not for a quantitative analysis. It also creates a casual point of entry for the interviewee to think about the topic under consideration. We communicate this to all participants to reduce any pressure they might feel to answer the question correctly. Afterwards an open-ended question inquiring about the reasons for the interviewees assessment is asked.

Some terms used within the discussed claims have ambiguous definitions. We tried to ask participants to explain their understanding of such terms, to prevent errors in analysis due to interviewees having different interpretations thereof. This allows for better assessment during analysis. The terms we have deemed to be ambiguous are: *'succinct syntax'*, *'mapping'*, *'specific skills'*, *'high-level abstractions'*, *'convenient facilities'*, *'sufficient tool support'*, *'powerful tool support'*, *'sophisticated reuse mechanisms'* and *'expressiveness'*. We provide a definition for the term *expressiveness*. This is, because we are only interested in a specific type of *expressiveness*, i.e. how concisely and readily developers can express something. We are not interested in expressiveness in a theoretical sense, i.e. the closeness to Turing completeness.

This process of presenting a claim, querying the participants agreement before discussing their reasons for the assessment is repeated for all 3 claims about both properties. After discussing all claims, it is explained to the participants that the formal part of the interview is finished and that they are allowed to make final remarks about all discussed topics or other properties they want to address. After this phase of the interview acknowledgements on the part of the interviewer are expressed before saying goodbye. The complete question catalogue for the interviews can be found in Appendix B.1.

Table 3.2: Properties and Claims

Property	Claim
Comprehensibility	The use of model transformation languages increases the comprehensibility of model transformations.
	Model transformation languages incorporate high-level abstractions that make them more understandable than general purpose languages.
	Most model transformation languages lack convenient facilities for understanding the transformation logic.
Ease of Writing	The use of model transformation languages increases the ease of writing model transformations.
	Model transformation languages ease development efforts by offering succinct syntax to query from and map model elements between different modelling domains.
	Model transformation languages require specific skills to be able to write model transformations.
Expressiveness	The use of model transformation languages increases the expressiveness of model transformations.
	Model transformation languages hide transformation complexity and burden from the user.
	Having written several transformations we have identified that current model transformation languages are too low a level of abstraction for succinctly expressing transformations between DSLs because they demonstrate several recurring patterns that have to be reimplemented each time.
Productivity	The use of model transformation languages increases the productivity of writing model transformations.
	Model transformation languages, being DSLs, improve the productivity.
	Productivity of GPL development might be higher since expert users for general purpose languages are easier to hire.
Reuse & Maintainability	The use of model transformation languages increases the reusability and maintainability of model transformations.
	Bidirectional model transformations have an advantage in maintainability.
	Model transformation languages lack sophisticated reuse mechanisms.
Tool Support	There is sufficient tool support for the use of model transformation languages for writing model transformations.
	Tool support for external transformation languages is potentially more powerful than for internal MTL or GPL because it can be tailored to the DSL.
	Model transformation languages lack tool support.

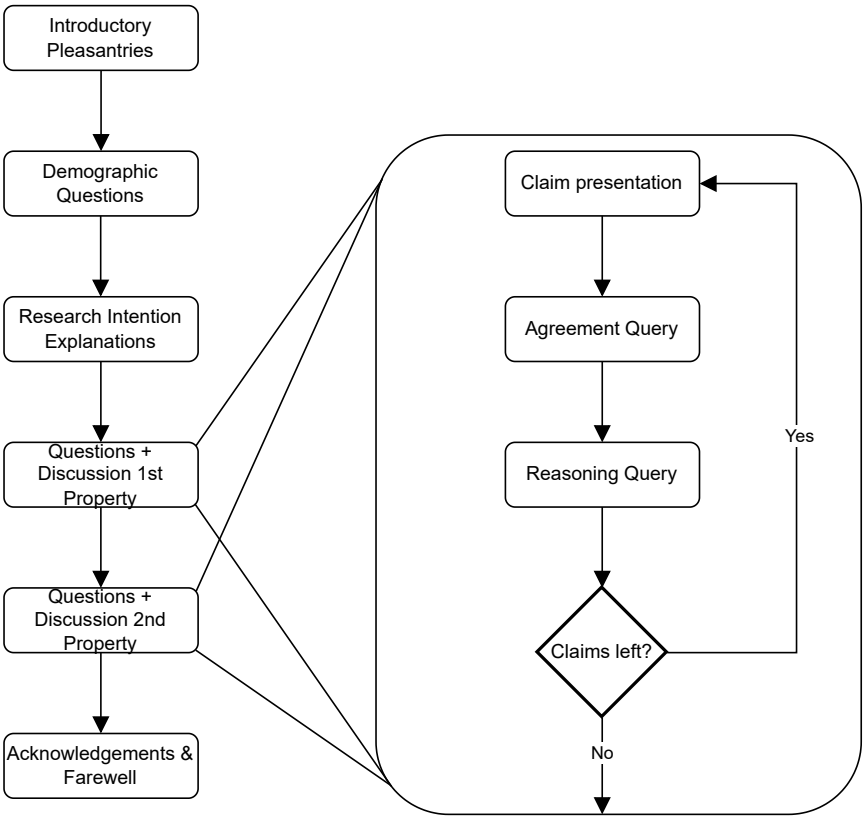


FIGURE 3.3: Interview guide

The interview guide was tested in a pilot study by the main author with one co-author that was not involved in its creation. After pilot testing, we changed the question about agreement with a claim from a yes-no question to one that uses a likert scale. We also extended the question sets with non-specific claims that do not contain any reasoning. Before adding the non specific claim, discussions focused too much on the narrow view within the presented claims.

3.3.1.4 Selecting & contacting participants

The target population for our study consists of all users of model transformation languages. To select potential participants for our study we rely on data from our previous literature review (Götz et al. 2021a). The literature review produced a list of publications that address the topic of model transformations and model transformation languages. Because search terms such as 'model to text' and the like were not used in the study, using this list limits our results to model to model transformation languages. We discuss this limitation more thoroughly in Section 3.8.2.

All authors of the resulting publications are deemed to be potential interview participants. We assume, that people using MTLs in industry do have some research background and thus have published work in the field. There is also no other systematic way to find industry users. We also assume that people who are still active in the field have published within the last 5 years. This limits outreach but makes the set of potential participants more manageable. For this reason, the list was shortened to publications more recent than 2015 before the authors of all publications was compiled. This resulted in a total of 645 potential participants.

After selection, the authors were contacted via mail. First, everyone was contacted once and then, after a week, everyone who had not responded by then was contacted again. The texts we use for both mails can be found in Appendix B.2. Ten potential participants, from the list of potential participants, were not contacted through this channel but via personalised emails, as they are personal contacts of the authors.

Within the contact mails, potential participants are asked to select a suitable date for the interview and fill out a data consent form allowing us to record and transcribe the interviews.

Overall of the 645 contacted authors, 55 agreed to participate in our interview study resulting in a response rate of 8.53%[1].

3.3.2 Interview Conduction and Transcription

All but one interview were conducted by the first author using the online conferencing tool WebEx and lasted between 20 and 80 minutes. Due to scheduling issues, one interview had to be conducted by the second author, who had a preparatory mock interview with the main interviewer. Additionally, at the request of two participants, one interview was conducted with both of them together. Since our main focus for all interviews is on discussions, we do not believe this to have any effect on its results. WebEx is the chosen conferencing tool, due to its availability to the authors and its integrated recording tool which is used to record all interviews. For data privacy reasons and for easier in-depth analysis later on, all recordings are transcribed by two authors. To increase the readability of heavily fragmented sentences they are shortened to only contain the actual content without interruptions. In case of audibility issues the transcribing authors consulted with each other to try and resolve the issue. Altogether the interviews produced just over 32 hours of audio and about 162.100 words of transcriptions.

Each day, the main author decided on which question sets to use for all participants that had agreed to partake in the interviews. The question sets had to be chosen daily, as many participants only responded to the invitation after interviews had already taken place.

The goal of the decision process was, to ensure an even spread of participants over the question sets based on relevant demographic backgrounds, namely *research*, *industry*, *MTL developer* and *MTL user*. We consider those relevant because each group has a different view point on model transformation languages and their usage for writing transformations. It is therefore important to have answers from each group for each set of questions, to reduce the risk of missing relevant opinions.

We were able to ensure that at least one representative for each demographic group provided answers for each question set. A complete uniform distribution was not possible due to overlaps in the demographic groups.

3.3.3 Coding & Analysis

Coding and analysing the interview transcripts is done in accordance with the guidelines for *content structuring content analysis* suggested by Kuckartz (2014). The guideline recommends a seven step process (depicted in Figure 3.4) for coding and analysing qualitative data. All steps are carried out with tool support provided by the MAXQDA[2] software. In the following, we explain how each process step is conducted in detail. We will use the following statement as a running example to show how codes and sub-codes are assigned and how the coding of text segments evolved throughout the process: *"Of course some MTLs use explicit traceability for instance. But even then you have a mechanism to access it. And if you have a MTL with implicit traceability where the trace links are created automatically then of course you gain a lot of expressivity because you don't have to write something that you would otherwise have to write for almost every rule."* (**P30**)

3.3.3.1 Initial Text Work

The initial text work step initiates our qualitative analysis. Kuckartz (Kuckartz 2014) suggests to read through all the material and highlight important segments as well as to write notes for the transcripts using memos. Following these suggestions, we apply *initial* coding from constructivist grounded theory (Charmaz 2014; Stol et al. 2016; Vollstedt et al. 2019) to mark and summarize all text segments where interviewees reason about their beliefs on influence factors about the discussed properties. To do so, the two authors, which conducted and transcribed the interviews, read through all transcripts and mark all relevant text segments with codes that preferably represented the segment word for word. The codes allow for easier reference in later steps and, due to tooling, we are still able to quickly read the underlying text segment if necessary.

During this step, the example statement was labelled with the code `automatic tracing increases expressiveness because no manual overhead`.

[1] when including the written response in this statistic, the resulting response rate is 8.68%.

[2] `https://www.maxqda.com/`

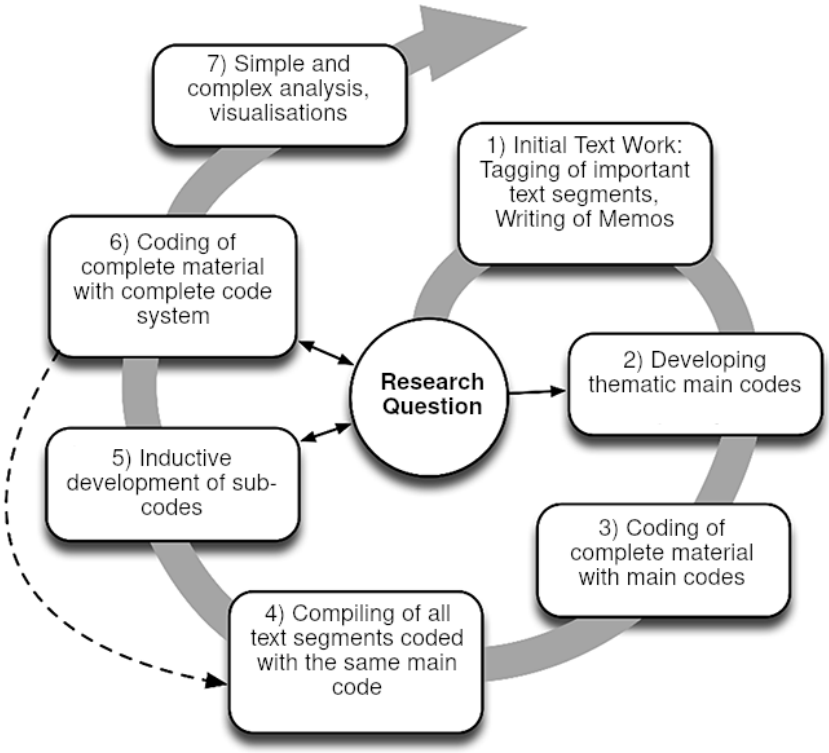


FIGURE 3.4: Process of a content structuring content analysis as presented by Kuckartz (2014)

3.3.3.2 Developing thematic main codes

For developing the thematic main codes for our study we follow the common practice of inferring them from our research questions as suggested by Kuckartz (2014). Since the goal of our research is to investigate implicit assumptions, and factors that influence the assessment of experts about properties of model transformation languages three main codes arise:

- **Properties**: Denoting which property is being discussed (e.g. *Comprehensibility*).
- **Factors**: Denoting what influences a discussed property according to an interviewee (e.g. *Bidirectional functionality of a MTL*).
- **Factor assessment**: Denoting an evaluation of how a factor influences a property (e.g. *positive* or *negative* or *mixed depending on other factors*).

The sub-codes for the property code can be directly defined based on the six properties from our previous literature review (Götz et al. 2021a). As such they are deductive (a-priori) codes that are intended to mark text segments based on the properties that are being discussed in them.

3.3.3.3 Coding of all the material with main codes

In order to code of all the material with the main codes one author analyses all interview transcripts. While doing so, the conversations about a discussed claim are marked with the code that is based on the property stated in the claim. To exemplify this, all discussions on the claim *"The use of MTLs increases the comprehensibility of model transformations."* are coded with the main code *comprehensibility*.

This realisation of the process step breaks with Kuckartz's specifications in multiple ways. First, we do not code the material with the main codes **Factors** and **Factor assessment**, because all factors and factor assessments are already coded with the summarising *initial* codes. These will be refined into actual sub-codes of **Factors** and **Factor assessment** in a later step. Second, we directly code segments with the sub-codes for the **Property** main code, because the differentiation comes

naturally with the structure of the interviews and delaying this refinement makes no sense. And third, this way of coding makes it possible that unimportant segments are also coded, something that Kuckartz suggests not to do. However, we actively decided in favour of this, because it accelerates the coding process enormously. Furthermore, only overlaps of the property codes with the other codes are considered, in later steps, thus automatically excluding unimportant text segments from consideration.

During this step, the coding for the example text segment was extended with the code `Expressiveness`. While this does not look like much of an enhancement on the surface, it is paramount to allow for systematic analysis in later steps.

After this step the example segment had its *initial code*, summarising the essence of the statement, and the explicit *property sub-code* `Expressiveness`, providing the first systematic categorisation of the segment.

3.3.3.4 Compilation of all text passages coded with the same main code

This step forms the basis for the subsequent iterative process of inductively developing sub-codes for each main code. Due to the use of the MAXQDA tool, this step is purely technical and does not require any special procedure outside of the selection of the main code that is being considered in the tool.

3.3.3.5 Inductive development of sub-codes

The inductive development of sub-codes forms the most important coding step in our study. Inductive development here means that the sub-codes are developed based on the transcripts contents.

Kuckartz (2014) suggests to read through all segments coded with a main code to iteratively refine the code into several sub-codes that define the main category more precisely. We optimize this step by analysing all the *initial* codes from the *Initial Text Work* step, to construct concise and comprehensive codes for similar *initial* codes that could be used as sub-codes for the **Factor** or **Factor assessment** main codes. In doing so we follow the *focused* coding procedure of constructivist grounded theory to refine the initial code system.

All sub-codes of the **Factor** main code, that are refined using this process, are thematic codes, meaning they denote a specific topic or argument made within the transcripts. As a result, the sub-codes represent factors explicitly named by interviewees that influence the different properties. In contrast, all sub-codes of the **Factor assessment** main code, that are refined using this process, are evaluative codes, meaning they represent an evaluation, made by the authors, about an effect. More specifically, the codes represent an evaluation of how participants believe factors influence various properties.

Because of the importance of this coding step, the sub code refinement is created in a joint effort by three of the authors. First, over a period of three meetings, the authors develop comprehensive codes based on the *initial* codes of 18 interviews through discussions. Then the main author complements the resulting code system by analysing the remaining set of interview transcripts, while the two other authors each analyse half of them. In a final meeting any new sub code, devised by one of the authors, is discussed and a consensus for the complete code system is established.

During this step no code segment is extended with additional codes. Instead new codes derived from the *initial codes* are saved for usage in the following steps.

From the example code segment and its *initial code*, a sub-code *automatic tracing* for the **Factors** code was derived. The finalised sub-code `Traceability` was decided upon based on the combination with other derived codes of similar meaning, like *traces*.

3.3.3.6 Coding of all the material with complete code system

After the final code system is established, the main author processes all transcripts to replace the *initial* codes with codes from the final code system. For this, each coded statement is re-coded with codes indicating the influence factors expressed by the interviewees as well as a factor assessment, if possible. This final coding step is done by the main author while all three co-authors each check 10 coded transcripts to validate the correct and consistent use of all codes and to make sure all relevant statements are considered. The results from the reviews are discussed in pairwise meetings between the main author and the reviewing co-author before being incorporated in a final coding approved by all authors.

During this step, the *initial code* for the example segment was dropped and replaced by the codes `MTL advantage` and `Traceability`.

The final codes assigned to the example text segment thus were: *Expressiveness*, *Traceability* and *MTL advantage*. The reasoning given within the statement as to why automatic tracing provides an expressiveness advantage, are manually extracted during analysis using tooling provided by MAXQDA.

3.3.3.7 Simple and complex analysis and visualisation

The resulting coding and the coded text segments are then used as the basis for our analysis which, in accordance with our research question, focuses on identifying and evaluating factors that influence the properties of MTLs. As recommended by Kuckartz (2014), this is first done for each **Property** individually before analysis across all properties is conducted (as shown in Figure 3.5).

For analysing the influence factors of an individual property, we use the MAXQDA tooling to find segments coded with both a factor and the considered property. Using this approach we first compile a list of all factors relevant for a property, before then doing an in-depth analysis of all the gathered statements for each factor. Here the goal is to elicit commonalities and contradictions between the opinions of our interviewees that can be used to establish a theory on how each factor influences each property individually.

In terms of our example text segment, the segment and all other segments coded with `Expressiveness` and `Traceability` were read and analysed. The goal was to see if reduced overhead from implicit trace capabilities played a role in the argumentation of other participants and to gather all the other mentioned reasons.

For the analysis over all properties combined we apply the *theoretical* coding process of constructivist grounded theory (Charmaz 2014; Stol et al. 2016) to develop a model of influences. To do so, the **Factor assessment**s are used to examine how the factors influence the respective properties, what the commonalities between properties are and where the differences lie. The goal here is to develop a cohesive theory which explains the influences of factors on the individual properties but also on the properties as a whole and potential influences between the factors themselves.

In terms of our example text segment, the results from analysing `Expressiveness` and `Traceability` segments were compared to results from analysing segments coded with other property codes and `Traceability`. The goal was to find commonalities and differences between the analysed groups.

3.3.3.8 Privacy and Ethical concerns

All interview participants were informed of the data collection procedure, the handling of the data and their rights surrounding the process, prior to the interview.

During selection of potential participants the following data was collected and processed.

- First & last name.
- E-Mail address.

For participants that agreed to the partake in the interview study the following additional data was collected and processed during the course of the study.

- Non anonymised audio recording of the interview.
- Transcripts of the audio recordings.

All data collected during the study was not shared with any person outside of the group of authors. Audio recordings were handled only by the first and second author.

The complete information and consent form can be found in Appendix B.4. All participants have consented to having their interview recorded, transcribed and analysed based on this information. All interview recordings were stored on a single device with hardware encryption and deleted as soon as transcriptions were finalised. The interview transcripts were processed to prevent identification of participants. For this, identifying statements and names were removed.

Apart from the voice recordings and names, no sensitive information about the interviewees was collected.

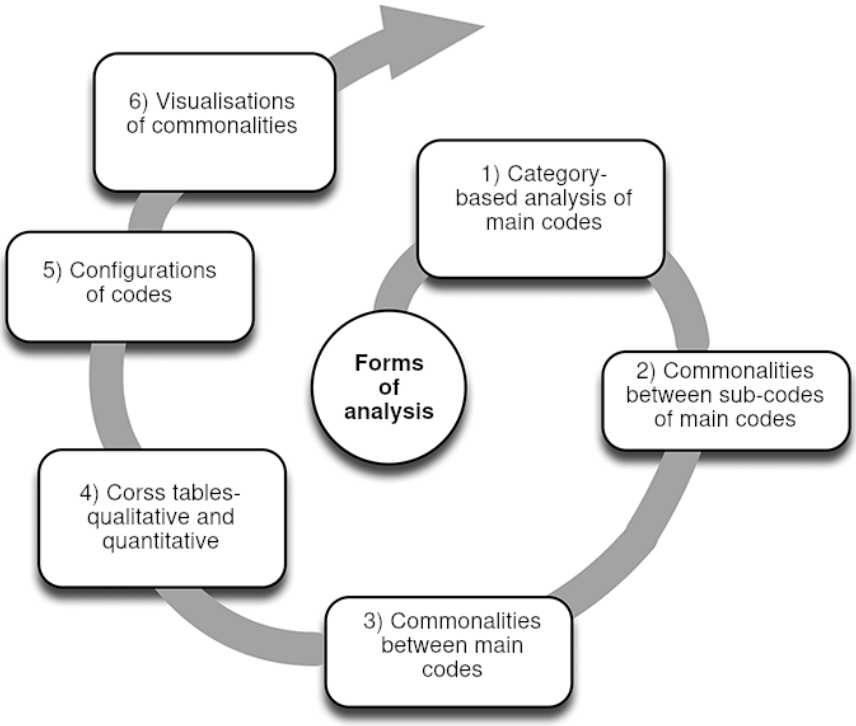


FIGURE 3.5: Analysis forms in a content structuring content analysis as presented by Kuckartz (2014)

The study design was not presented to an ethical board. The basis for this decision are the rules of the German Research Foundation (DFG) on when to use a ethical board in humanities and social sciences[3]. We refer to these guidelines because there are none specifically for software engineering research and humanities and social sciences are the closest related branch of science for our research.

3.4 Demographics

We interviewed a total of 55 experts from 16 different countries with varied backgrounds and experience levels and collected one comprehensive written response. Table B.1 in Appendix B.3 presents an overview of the demographic data about all interview participants. Experts and their statements are distinguished via an anonymous ID (**P1** to **P56**).

3.4.1 Background

As evident from Figure 3.6 participants with a research background constitute the largest portion of our interviewees. Overall there is an even split between participants solely from research and those that have at least some degree of industrial contact (either through industry projects or by working in industry). Only 3 participants stated to have used model transformations solely in an industrial context. This is in part offset by the fact that 25 of interviewees have executed research projects in cooperation with industry or have worked both in research and industry (22 and 3 respectively). While there is a definitive lack of industry practitioners present in our study, a large portion of interviewees are still able to provide insights into model transformations and model transformation languages with an industry view.

Lastly, 10 of our participants are, in some capacity, involved in the development of model transformation languages. They can provide a different angle on advantages or disadvantages of MTLs compared to the 46 participants that use them solely for transformation purposes.

[3] `https://www.dfg.de/foerderung/faq/geistes_sozialwissenschaften/`

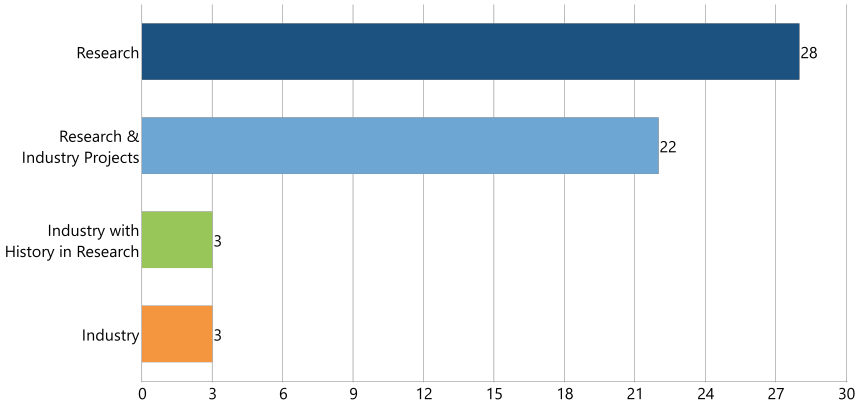


FIGURE 3.6: Distribution of participants background

3.4.2 Experience

50 interviewees expressed to have 5 or more years of experience in using model transformations. Moreover, 24 of the participants have over 10 years of experience in the field. Lastly there was a single participant that had only used model transformations for a brief amount of time during their masters thesis.

3.4.3 Used languages for transformation development

To better assess our participants and to qualify their answers with respect to their background we asked all interviewees to list languages they used to develop model transformations. Figure 3.7 summarises the answers given by participants while categorizing languages in one of three categories namely *dedicated MTL*, *internal MTL* and *GPL*. This differentiation is based on the classifications from Czarnecki et al. (2006) and Kahani et al. (2019).

The distinction between GPL and dedicated/internal MTL is made, to gain an overview over how large the portion of users of general purpose languages for the development is, compared to the users of model transformation languages. Furthermore, it also allows for comprehending the viewpoint participants will take when answering questions throughout the interview, i.e. do they compare general purpose languages with model transformation languages based on their experience with both or do they give specific insights into their experiences with one of the two approaches. Internal MTL is separated from dedicated MTL because one claim within the interview protocol specifically explores the topic of internal model transformation languages.

52 participants have used dedicated model transformation languages such as ATL, Henshin or Viatra for transforming models. Only half as many (27) stated to have used general purpose languages for this goal. Lastly, only 5 indicated the use of internal MTLs.

When looking at the specific dedicated MTLs used ATL is by far the most prominent one used by interviewees. A total of 37 participants mention having used ATL. This is more than double the amount of the second most used language namely Henshin which is only mentioned by 17 interviewees. The QVT family then follows in third place with QVT-R having been used by 13 participants, QVT-O by 11. A complete overview over all dedicated model transformation languages used by our interviewees can be found in Figure 3.8. Note that several interviewees mentioned using more than one language, making the total number of data points in this figure larger than 52.

In the group of GPL languages used for model transformation (summarised in Figure 3.9), Java is the most used language with 14 participants stating so. Note that several interviewees mentioned using more than one language, making the total number of data points in this figure larger than 27. Java is closely followed by Xtend which is mentioned by 12 interviewees. Then follows a steep drop of in popularity with Java Emitter Templates having been used by only four participants.

Lastly, only four internal model transformation languages, namely RubyTL, NTL, NMF Synchronizations and FunnyQT, are mentioned. This shows a lack of prominence thereof. Moreover none of the languages is used by more than two interviewees.

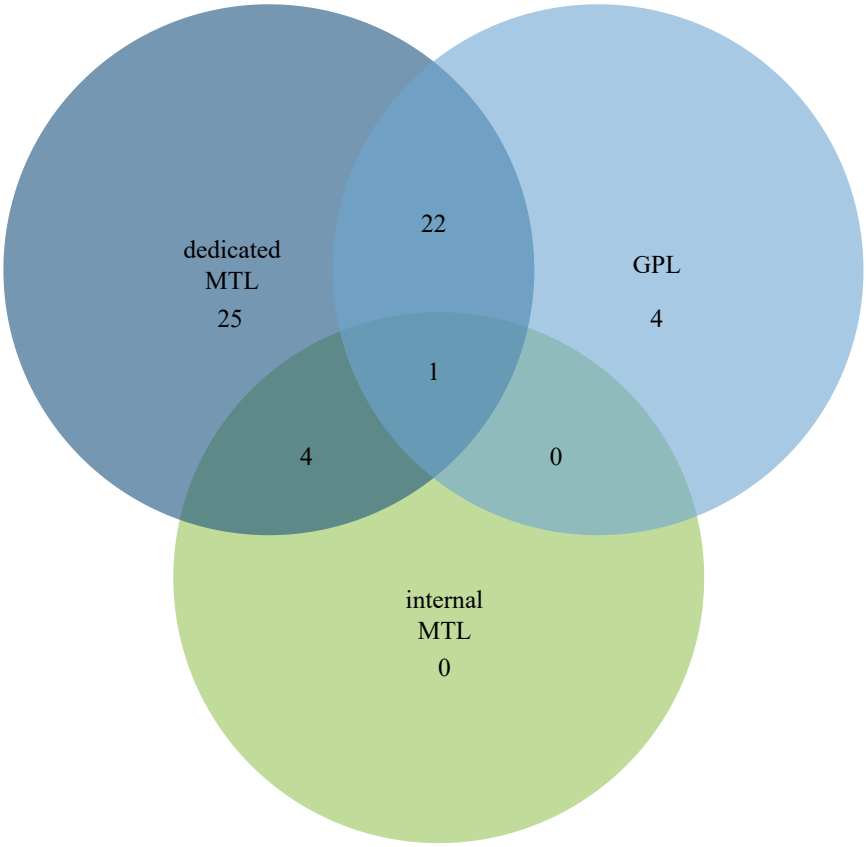


FIGURE 3.7: Venn diagram depicting the language usage of participants

3.5 Findings

Based on the responses of our interviewees and our analysis, we developed a framework to classify influence factors. It allows us to categorize how factors influence properties of MLTs and each other according to our interviewees. Note that we split the property *Reuse & Maintainability* into two properties for the purpose of reporting. This is done because interviewees chose to consider them separately. Thus reporting on them separately allows for presenting more nuanced results.

The factors themselves are split into six top-level factors namely *GPL Capabilities*, *MTL Capabilities*, *Tooling*, *Choice of MTL*, *Skills* and *Use Case*. The first factor, *GPL Capabilities*, encompasses sub-factors related to writing model transformations in general purpose languages. *MTL Capabilities* encompasses sub-factors that originate from transformation specific features of model transformation languages. *Tooling* contains factors surrounding tool support for MTLs. *Choice of MTL* details how the choice of language asserts its influence. The factor *Skills* encompasses sub-factors associated with skills. Lastly, the *Use Cases* factor contains sub-factors that relate to the involved use case an its influences.

Within the framework we differentiate between two kinds of factors. The first kind are factors, that have a positive or negative impact on properties of MTLs. These include the factors *GPL Capabilities*, *MTL Capabilities* and *Tooling* as well as their sub-factors. The second kind are factors that, depending on their characteristic, moderate how other factors influence properties, e.g. depending on the language, its syntax might have a positive or negative influence on the comprehensibility of written code. We call such factors *moderating* factors. These include the factors *Choice of MTL*, *Skills* and *Use Case* and their sub-factors.

Table 3.3 provides an overview over the answers given by our interviewees. The table shows factors on its rows and MTL properties on its columns. A `+` in a cell denotes, that interviewees mentioned the factor to have a positive effect on their view of the MTL property. A `-` means interviewees saw a negative influence and `+/-` describes that there have been mentions of both positive and negative influences. Lastly, a `M` in a cell denotes, that the factor represents a moderating

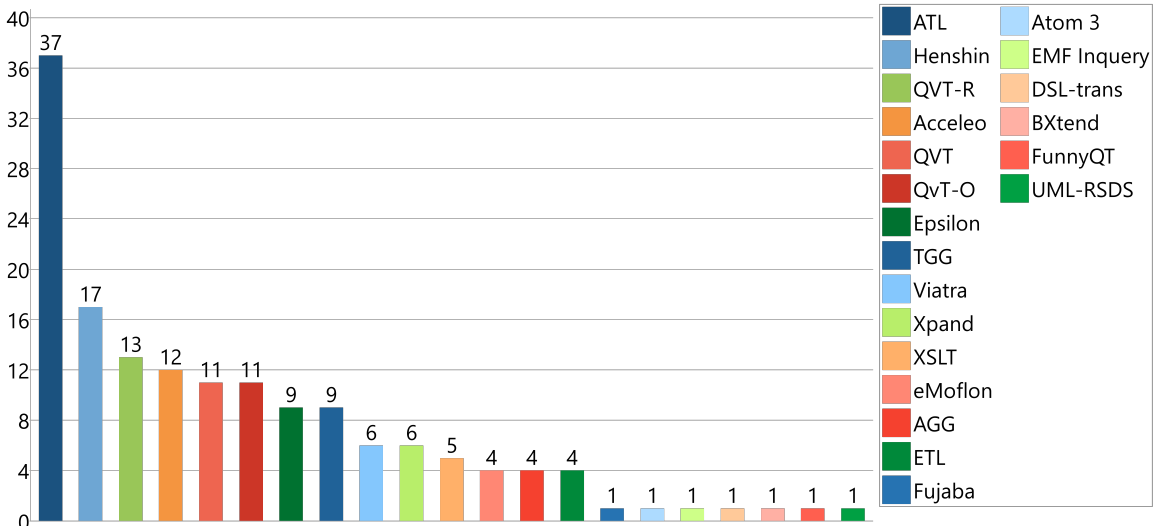


FIGURE 3.8: Number of participants using a specific dedicated MTL

factor for the MTL property, according to some interviewees. The detailed extent of the influence of each factor is described throughout Sections 3.5 and 3.6.

In the following we present all top-level factors and their sub-factors and describe their place within our framework. For each factor we detail its influence on properties of model transformation languages or on other factors, based on the statements made by our interviewees.

3.5.1 GPL Capabilities

Using general purpose languages for developing model transformations, as an alternative to using dedicated languages was extensively discussed in our interviews. Interviewees mentioned both advantages and disadvantages that GPLs have compared to MTLs that made them view MTLs more or less favourably.

The disadvantages of GPLs compared to MTLs stem from additional features and abstractions that MTLs bring with them and will be discussed later in Section 3.5.2. The advantages of GPLs on the other hand can not be placed within the *MTL Capability* factors. These will instead be presented separately in this section.

According to our interviewees, advantages of GPLs are a relevant factor for all properties of MTLs.

General purpose languages are better suited for *writing* transformations that require lots of computations. This is because they were streamlined for these kinds of activities and designed for

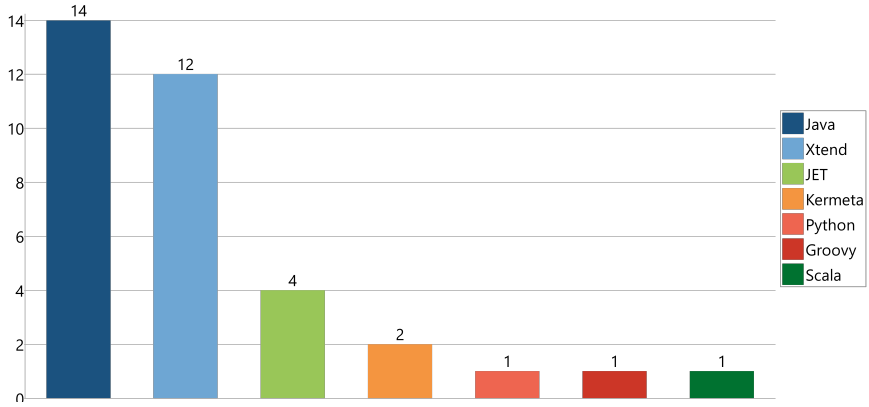


FIGURE 3.9: Number of participants using a specific GPL

Table 3.3: Overview over quality attribute influences per factor

Top-level Factor	Sub-Factor	Comprehensibility	Ease of Writing	Expressiveness	Maintainability	Productivity	Reusability	Tool Support
GPL Capabilities		+/-	+/-	-	-	+/-	+	+/-
	Domain Focus	+	+	+/-	+	+/-		+
	Bidirectionality	+/-	+/-	+	+/-	+		
	Incrementality	+	+/-	+				
	Mappings	+	+/-	+	+		+/-	
MTL Capabilities	Traceability	+	+/-	+/-		+		
	Model Traversal	+	+	+		+		
	Pattern Matching	+		+		+		
	Model Navigation	+	+	+				
	Model Management	+	+	+				
	Reuse Mechanisms						+/-	
	Learnability		-					
	Analysis Tooling	+				+		+/-
	Code Repositories						-	-
	Debugging Tooling	+/-						+/-
	Ecosystem				-	-		-
	IDE Tooling		+/-		-			-
Tooling	Interoperability							-
	Tooling Awareness							-
	Tool Creation Effort							-
	Tool Learnability		-					-
	Tool Usability		-			-		-
	Tool Maturity							-
	Validation Tooling							-
Choice of MTL		M	M	M	M	M	M	M
Skills	Language Skills	M	M		M		M	
	User Experience/Knowledge		M		M	M		
	(Meta-) Models		M					
Use Case	I/O Semantic gap	M	M			M		
	Size	M	M					

this task, with language features like streams, generics and lambdas. As a result, general purpose languages are far more advanced for such situations compared to model transformation languages, which sacrifice this for more domain expressiveness [$Q_{gpl}1$].

Much like the language design for GPLs, their tools and ecosystems are mature and designed to integrate well with each other. Moreover, according to several interviewees, their tools are of high quality making developers feel more *Productive* [$Q_{gpl}2$].

Lastly, multiple participants noted, that there are much more GPL developers readily available for companies to hire, thus making GPLs more attractive for them. This helps the *Maintainability* of existing code as such experts are more likely to *Comprehend* GPL code [$Q_{gpl}3$]. Whether this aspect also improves the overall *Productivity* of transformation development in a GPL was disagreed upon, because it might be that developers trained in a MTL could produce similar results with less resources.

It was also mentioned, that much more training resources are available for GPL development, making it easier to start learning and using a new GPL compared to a MTL.

3.5.2 MTL Capabilities

The capabilities that model transformation languages provide that are not present in GPLs, are important factors that influence properties of the languages. This view is shared by our interviewees that raised many different aspects and abstractions present in model transformation languages.

The influence of capabilities specifically introduced in MTLs is diverse and depends on the concrete implementation in a specific language, the skills of the developers using the MTL and the use case in which the MTL is to be applied. We will discuss all the implications raised by our interviewees regarding the transformation specific capabilities of MTLs for the properties attributed to MTLs in detail, in this section.

3.5.2.1 Domain Focus

Domain Focus describes the fact that model transformation languages provide transformation specific constructs, abstractions or workflows. Interviewees remarked the domain focus, provided by MTLs, as influencing *Comprehensibility*, *Ease of Writing*, *Expressiveness*, *Maintainability*, *Productivity* and *Tool Support*. But the effects can differ depending on the specific MTL in question.

There exists a consensus that MTLs can provide better domain specificity than GPLs by introducing domain specific language constructs and abstractions. This increases *Expressiveness* by lifting the problem onto the same level as the involved models allowing developers to express more while writing less. MTLs allow developers to treat the transformation problem on the same abstraction level as the involved modelling languages [$Q_{df}1$]. This also improves the *ease of development*.

Several interviewees argued, that when moving towards domain specific concepts the *Comprehensibility* of written code is greatly increased. The reason for this is, that because transformation logic is written in terms of domain elements, unnecessary parts are omitted (compared to GPLs) and one can focus solely on the transformation aspect [$Q_{df}2$].

Having domain specific constructs was also raised as facilitating better *Maintainability*. Coevolving transformations written in MTLs together with hardware-, technology-, platform- or model changes is said to be easier than in GPLs because *"Once you have things like rules and helpers and things like left hand side and right hand side and all these patterns then [it is] easier to create things like meta-rules to take rules from one version to another version [...]"* (**P23**).

Domain focus also enforces a stricter code structure on model transformations. This reduces the amount of variety in which they can be expressed in MTLs. As a result, developing *Tool Support* for analysing transformation scripts gets easier. Achieving similarly powerful tool support for general purpose languages, and even for internal MTLs, can be a lot harder or even impossible because much less is known solely based on the structure of the code. Analysis of GPL transformations has to deal with the complete array of functionality of general purpose language constructs [$Q_{df}4$]. While MTLs can be Turing complete too, they tend to limit this capability to specific sections of the transformation code. They also make more information about the transformation explicit compared to GPLs. This allows for easier analysis of properties of the transformation scripts which reduces the amount of work required to develop analysis tooling.

The influence of domain abstractions on *Productivity* was heavily discussed in our interviews. Interviewees agreed that, depending on the used language, *Productivity* gains are likely, due to their

domain focus. However, one interviewee explained that precisely because of *Productivity* concerns companies in the industry might use general purpose languages. The reason for this boils down to the *Use Case* and project context. Infrastructure for general purpose languages might already be set up and developers do not need to be trained in other languages [$Q_{df}5$]. Moreover, different tasks might require different model transformation languages to fully utilise their benefits, which, from an organisational standpoint, does not make sense for a company. So instead one GPL is used for all tasks.

3.5.2.2 Bidirectionality

According to our interviewees bidirectional functionality in a model transformation language influences its *Comprehensibility*, *Ease of Writing*, *Expressiveness* and *Maintainability* and *Productivity*. Its effects on these properties then depends on the concrete implementation of the functionality in a *MTL*. It also depends on the *Skills* of the developers and the concrete *Use Case*.

Our interviewees mentioned that the problem of bidirectional transformations is inherently difficult and that high level formalisms are required to concisely define all aspects of such transformations. Many believe that because of this solutions using general purpose languages can never be sufficient. Statements in the vein of *"in a general purpose programming language you would have to add a bit of clutter, a bit of distraction, from the real heart of the matter"* (**P42**) were made several times. This, combined with having less optimal querying syntax, then shifts focus away from the actual transformation and decreases both the *Comprehensibility* and *Maintainability* of the written code.

Maintainability is also hampered because GPL solutions scatter the implementation throughout two or more rules (or methods or files) that have to be adapted in case of changes [$Q_{bx}2$]. Expressive and high level syntax in MTLs helps alleviate these problems and increases the *ease* at which developers can *write* bidirectional model transformations.

Interviewees also commented on the fact that, thanks to bidirectional functionalities, consistency definitions and synchronisations between both sides of the transformation can be achieved easier. This improves the *Maintainability* of modelling projects as a whole and allows for more *Productive* workflows. Manual attempts to do so have been stated to be error-prone and labour-intensive.

It was also pointed out that the inherent complexity of bidirectionality leads to several problems that have to be considered. MTLs that offer syntax for defining bidirectional transformations are mentioned to be more complex to use as their unidirectional counterparts. They should thus only be used in *cases* where bidirectionality is a requirement. Moreover, one interviewee mentioned that developers are not generally used to thinking in bidirectional way [$Q_{bx}3$].

Lastly, the models involved in bidirectional transformations also play a role regardless of the language used to define the transformation. Often the models are not equally powerful making it hard to actually achieve bidirectionality between them, because of information loss from one side to the other [$Q_{bx}4$].

3.5.2.3 Incrementality

Dedicated functionality in MTLs for executing incremental transformations has been discussed as influencing *Comprehensibility*, *Ease of Writing* and *Expressiveness*. Similar to bidirectionality its influence is again heavily dependent on the *Use Case* in which incremental languages are applied as well as the *Skills* of the involved developers.

Declarative languages have been mentioned to facilitate incrementality because the execution semantics are already hidden and thus completely up to the execution engine. This increases the *Expressiveness* of language constructs. It can, however, hamper the *Comprehensibility* of transformation scripts for developers inexperienced with the language because there is no direct way of knowing in which order transformation steps are executed [$Q_{inc}1$].

On the other hand interviewees also explained that writing incremental transformations in a GPL is unfeasible. Manual implementations are error-prone because too many kinds of changes have to be considered and chances are high that developers miss some specific kind. Due to the high level of complexity that the problem of incrementality inherently posses interviewees argued that *writing* such transformations in MTLs is much *easier* [$Q_{inc}2$].

The same argumentation also applied for the *Comprehensibility* of transformations. All the additional code required to introduce incrementality to GPL transformations is argued to clutter the code so much that developers "*[will be] in way over their head[s]*" (**P13**).

As with bidirectionality interviewees agreed, that the *Use Case* needs to be carefully considered when arguing over incremental functionality. Only when ad-hoc incrementality is really needed should developers consider using incremental languages. In cases where transformations are executed in batches, maybe even over night, no actual incrementality is necessary and then "*general purpose programming languages are very much contenders for implementing model transformations*" (**P42**). It was also explained that using general purpose languages for simple transformations is common practice in industry as they are "*very good in expressing [the required] control flow*" (**P42**) and because none of the aforementioned problems for GPLs have a strong impact in these cases.

3.5.2.4 Mappings

The ability of a MTL to define mappings influences that languages *Comprehensibility*, *Ease of Writing*, *Expressiveness*, *Maintainability* and *Reuse* of model transformations. Developer *Skills*, the used *Language* and concrete *Use Case* also play an important role in the kind of influence.

Interviewees agreed, that the *Expressiveness* of transformation languages utilising syntax for mapping is increased due to them hiding low level operations [$Q_{map}1$]. However, as remarked by one participant, the semantic complexity of transformations can not be hidden by mappings, only the computational complexity.

According our interviewees mappings form a natural way of how people think about transformations. They impose a strict structure on how transformations need to be defined, making it easy for developers to start of *writing* transformations. The structure also aids general development, because all elements of a transformation have a predetermined place within a mapping. Being this restrictive has the advantage of directing ones thoughts and focus solely on the elements that should be transformed [$Q_{map}2$]. To transform an element, developers only need to write down the element and what it should be mapped to.

The simple structure expressed by mappings also benefits the *Comprehensibility* of transformations. It allows to easily grasp which elements are involved in a transformation, even by people that are not experienced in the used language. Trying to understand the same transformation in GPLs would be much harder because *"[one] would not recognize [the involved elements] in Java code any more"* (**P32**). Instead, they are hidden in between all the other instructions necessary to perform transformations in the language. Interviewees also mentioned that, due to the natural fit of mappings for transformations, it is much easier to find entry points from where to start and understand a transformation and to reconstruct relationships between input and output. This is aided by the fact that the order of mappings within a transformation does not need to conform with its execution sequence and thus enables developers to order them in a comprehensible way [$Q_{map}3$].

One interviewee explained that, from their experience, mappings lead to less code being written which makes the transformations both easier to *comprehend* and to *maintain*. However, they conceded that the competence of the involved developers is a crucial factor as well. According to them, language features alone do not make code maintainable. Developers need to have language engineering skills and intricate domain knowledge to be able to design well maintainable transformations [$Q_{map}4$]. Both are skills that too little developers posses.

Moreover, several interviewees raised the concern, that complex *Use Cases* can hamper the *Comprehensibility* of transformations. Understanding which elements are being mapped can be hard to grasp if several auxiliary functions are used for selecting the elements. Here one interviewee suggested that a standardized way of annotating such selections could help alleviate the problem.

It was also mentioned that, while mappings and other MTL features increase the *Expressiveness* of the language, they might make it harder for developers to start learning the languages. Because a lot of semantics are hidden behind keywords, developers need to first understand the hidden concepts to be able to utilise them correctly [$Q_{map}5$].

Other features that highlight how much *Expressiveness* is gained from mappings have also been mentioned. Mappings hide how relations between input and output are defined. This creates a formal and predictable correspondence between them and thus enables *Tracing*. Moreover, the correspondence between elements allows languages to provide functionality such as *Bidirectionality* and *Incrementality* [$Q_{map}6$].

Because many languages that utilise mappings can forgo definitions of explicit control flow, mappings allow transformation engines to do execution optimisations. However, one interviewee explained that they encountered *Use Cases* where developers want to influence the execution order, forcing them to introduce imperative elements into their code effectively hampering this advantage.

It has also been mentioned that in *complex cases* the code within mappings can get complicated to the point where non experts are unable to *comprehend* the transformation again. This problem also exists for *writing* transformations as well. According to one interviewee mappings are great for linear transformations and are thus very dependent on the involved (meta-)models. Also in *cases* where complex interactions needs to be defined mappings do not present any advantage over GPL syntax and sometimes it can even be easier to define such logic in GPLs [$Q_{map}7$].

Lastly, mappings enable more modular code to be written. This in turn facilitates *reuse*, because reusing and changing code results in local changes instead of several changes throughout different parts of GPL code [$Q_{map}8$].

3.5.2.5 Traceability

The ability in model transformation languages to automatically create and handle trace information about the transformation has been discussed by our interviewees to influence *Comprehensibility*, *Ease of writing*, *Expressiveness* and *Productivity*. However, the concrete effect depends on the *MTL* and the *skill* of users.

All interviewees talking about automatic tracing agreed that it increases the *Expressiveness* of the language utilising it. In GPLs this functionality would need to be manually implemented using structures like hash maps. Code to set up traces would then also need to be added to all transformation rules [$Q_{trc}1$].

However, interviewees disagreed on how much this actually impacts the overall transformation development. Most interviewees felt like automatic trace handling *Eases Writing* transformations and even increases *Productivity* since no manual workarounds need to be implemented. This is because manual implementation requires developers to think about when and in which cases traces need to be created and how to access them correctly. It also enables languages that allow developers to define rules independent from the execution sequence. One interviewee however felt like this was not as effort intensive as commonly claimed and thus automatic trace handling to them is more of a nice to have feature than a requirement for writing transformations effectively. Moreover, for complex *Use Cases* of tracing such as QVTs late resolve, the *Users* are required to understand the principle of tracing [$Q_{trc}2$]. And according to another interviewee teaching how tracing and trace models work is hard.

Comprehending written transformations can also be aided by automatic trace management. Manual implementations introduce unnecessary clutter into transformation code that needs to be understood to be able to understand a whole transformation. This is especially true if effort has been put into making tracing work efficiently, according to one interviewee. Understanding a transformation is much more straight forward when only the established relationships between the input and output domains need to be considered, without any additional code to setup and use traces [$Q_{trc}3$].

Lastly, one interviewee raised the issue that manual trace handling might be necessary to write complex transformations involving multiple source and target models, as current engines are not intended for such *Use Cases*.

3.5.2.6 Automatic Model Traversal

According to our interviewees, the automatic traversal of the input model to apply transformations influences *Ease of Writing*, *Expressiveness*, *Comprehensibility* and *Productivity*. They also explain that depending on the implementation in a concrete *MTL* the effects can differ. *Use Cases* are also mentioned to be relevant to the influence of automatic traversal.

Automatic model traversal hides the traversal of the input model and how and when transformations are applied to the input model elements. Because of this many interviewees expressed that this feature in MTLs increases their *Expressiveness*. The reduced code clutter also helps with *Comprehensibility*.

It also *Eases the Writing* of transformations because developers do not need to worry about traversing the input and finding all relevant elements, a task that has been described as complicated by interviewees. This can be of significant help to developers. One interviewee explained, that they ran an experiment with several participants where they observed model traversal to be *"one of the biggest problems for test persons"* (**P49**).

Not having to manually define traversal reduces the amount of code that needs to be written and thus increases the overall *Productivity* of development, according to one interviewee. However, there

can also be drawbacks from this practice. Hiding the traversal automatically leads to the context of model elements to be hidden from the developer. In cases where the context contains relevant information this can be detrimental and even mask errors that are hard to track down [$Q_{trv}1$].

Lastly, automatic input traversal enables transformation engines to optimize the order of execution in declarative MTLs. And MTLs where no automatic execution ordering can be performed have been described as being "*close to plain GPLs*" (**P52**).

3.5.2.7 Pattern-Matching

Some model transformation languages, such as Henshin, allow developers to define sub-graphs of the model graph, often using a graphical syntax, to be matched and transformed. This pattern-matching functionality influences the *Comprehensibility*, *Expressiveness* and *Productivity*, according to our interviewees. It is, however, strongly dependent on the specific *language* and *Use Case*. The feature is only present in a small portion of MTLs and brings with it its own set of restrictions depending on the concrete implementation in the language.

Pattern-matching functionality greatly increases the *Expressiveness* of MTLs. Similar to the basic model traversal no extra code has to be written to implement this semantic. However, the complexity of the abstracted functionality is even higher, since it is required to perform sub-graph matching to find all the relevant elements in a model. These patterns can also become arbitrarily complex and thus all interviewees talking about pattern-matching agreed that manual implementations are nearly impossible. Nevertheless one interviewee mentioned, that all languages they used that provided pattern-matching functionality (Henshin and TGG) had the drawback of providing no abstractions for resolving traces which takes away from its overall usefulness for certain *Use Cases* [$Q_{pm}1$].

Not having to implement complex pattern-matching algorithms manually is also mentioned to increase the *Productivity* of writing transformations because this task is labour-intensive and error-prone.

Improvements for the *Comprehensibility* of transformations have also been recognized by some interviewees. They explained that the, often times graphical, syntax of languages with pattern-matching functionality allows to directly see the connection between involved elements. In GPLs this would be hidden behind all the code required to find and collect the elements. As such MTL code is *"less polluted"* (**P52**) than GPL code. Moreover, the *Comprehensibility* is also promoted by the fact that in some languages the graphical syntax shows the involved elements as they would be represented in the abstract syntax of the model.

3.5.2.8 Model Navigation

Dedicated syntax for expressing model navigation has influence on the *Comprehensibility* and *Ease of Writing* of model transformations as well as on the *Expressiveness* of the *MTL* that utilises it.

Having dedicated syntax for model navigation helps to *ease development* as it allows transformation engineers to simply express which elements or data they want to query from a model while the engine takes care of everything else. Furthermore, it has been mentioned that this has a positive effect on transformation development because developers do not need to consider the efficiency of the query compared to when defining such queries using nested loops in general purpose languages [$Q_{nav}1$].

Because languages like OCL abstract from how a model is navigated to compute the results of a query, interviewees attributed a higher *Expressiveness* to them than GPL solutions and described code written in these languages as more concise. Several interviewees attribute a better *Comprehensibility* to OCL as a result of this conciseness, arguing that well designed conditions and queries written in OCL are easy to read [$Q_{nav}2$].

OCL has however also been criticised by an interviewee. According to them, the language is too practically oriented, misses a good theoretical foundation and lacks elegance to properly express ones intent. They explain that because of this, the worth of learning such a language compared to using a more common language is uncertain.

3.5.2.9 Model Management

The impact of having to read and write models from and to files, i.e., model management, has been discussed by several interviewees. Automatic model management was discussed in our interviews as

influencing the *Comprehensibility*, *Ease of Writing* and *Expressiveness* of model transformations in MTLs.

The argument for all three properties boils down to developers not having to write code for reading input models or writing output models, as well as the automatic creation of output elements and the order thereof. Interviewees agreed that implicit language functionality for these aspects raised the *Expressiveness* of languages. It reduces clutter when reading a written transformation and thus improving the *Comprehensibility*. Finally, developers do not have to deal with model management tasks, e.g. using the right format, that are not relevant to the actual transformation which helps with *writing* transformations [$Q_{man}1$].

3.5.2.10 Reuse Mechanism

Mechanisms to reuse model transformations mostly influence the *Reusability* of model transformations in MTLs. Their concrete influence depends on the used *Language* and how reuse is handled in it. Interviewees also reported on cases where the users *Skills* with the language was relevant because novices might not be familiar with how the provided facilities can be utilised to achieve reuse.

There exists discourse between the interviewees about reuse mechanisms and their usefulness in model transformation languages. Several interviewees argued that MTLs do not have any reuse mechanisms that go beyond what is already present in general purpose languages. They believe that most, if not all, the reuse mechanisms that exist in MTLs are already present in GPLs and as such MTLs do not provide any reuse advantages [$Q_{rm}1$]. According to them such reuse mechanisms include things like rule inheritance from languages like ATL or modules and libraries.

Other interviewees on the other hand suggested that while the aforementioned mechanisms stem from general purpose languages, they are still more transformation specific than their GPL counterpart. This is, because the mechanisms work on transformation specific parts in MTLs rather than generic constructs in GPLs [$Q_{rm}2$, $Q_{rm}3$]. Because of this focus, interviewees argue that they are more straight forward to use and thus improve *Reusability* in MTLs.

Interviewees also explained that there exist many languages that do not provide any useful reuse or modularisation mechanisms and that even in those that do it can be hard to achieve *Reusability* in a useful manner. However, one participant acknowledged that in their case, the reason for this might also relate to the inability of the *Users* to properly utilize the available mechanisms.

It has also been mentioned that reuse in model transformations is an inherently complex problem to solve. Transferring needs between two transformations which apply on different meta-models is difficult to do. As such, model transformation are often tightly tied to the domain in which they are used which makes reuse hard to achieve and most reuse between domains is currently done via copy & paste. This argument can present a reason why, as criticised by several interviewees, no advanced reuse mechanisms are broadly available.

The desire for advanced mechanisms has been expressed several times. One interviewee would like to see a mechanism that allows to define transformations to adapt to different input and output models to really feel like MTLs provide better reusability than GPLs. Another mentioned, that all reuse mechanisms conferred from object orientation rely on the tree like structure present in class structures while models are often more graph like and cyclic in nature. They believe that mechanisms that address this difference could be useful in MTLs.

Another disadvantage in some MTLs that was raised, is the granularity on which reuse can be defined. In languages like Henshin, for example, reuse is defined on a much coarser level than what is possible in GPLs.

Not having a central catalogue, similar to maven for Java, from which transformations or libraries can be reused, has also been critiqued as hindering reuse in model transformation languages.

3.5.2.11 Learnability

The learnability of model transformation languages has been discussed as influencing the *Ease of Writing* model transformations.

It has been criticised by several interviewees, that the learning curve for MTLs is steep. This is, in part, due to the fact that users not only need to learn the languages themselves, but also accompanying concepts from MDE which are often required to be able to fully utilise model transformation languages. The learning curve makes it difficult for users to get started and therefore hampers the *Ease of Writing* transformations [$Q_{ler}1$]. This effect could be observed among computer science students at several of the universities of our interviewees. The students were described to having

difficulties adapting to the vastly different approach to development compared to more traditional methods. A potential reason for this could be that people come into contact with MDE principles too late, as noted by an interviewee [$Q_{ler}2$].

3.5.3 Tooling

While *Tool Support* is a MTL property that was investigated in our study, the tooling provided for MTLs, as well as several functional properties thereof, have been raised many times as factors that influence other properties attributed to model transformation languages as well. Most of the time this influence is negative, as tooling is seen as one of the greatest weak points of model transformation languages by our interviewees.

Many interviewees explained, that the most common languages do in fact have tools. The problem, however, lies in the fact that some helpful tools only exist for one language while others only exist for another language. As a result there is always some tool missing for any specific language. This leads people to feel like *Tool Support* for MTLs is bad compared to GPLs. Though there was one interviewee that explained that for their *Use Cases*, all tools required to be productive were present.

In the following, we will present several functional properties and tools that interviewees expressed as influential for *Tool Support* as well as other properties of MTLs.

3.5.3.1 Analysis Tooling

Analysis tools are seen as a strong suit of MTLs. Their existence in MTLs is said to impact *Productivity*, *Comprehensibility* and perceived *Tool Support*.

According to the interviewees, some analyses can only be carried out on MTLs, as the abstraction in transformations in GPLs is not high enough and too much information is represented by the program logic and not in analysable constructs. As one interviewee explained, this comes from the fact that for complex analysis, such as validating correctness, languages need to be more structured. Nevertheless, participants mainly mentioned analyses they would like to see, which is an indication that, while the potential for analysis tools for MTLs is high, they do not yet see usable solutions for it, or are unaware of it. This is highlighted by one interviewee that explained that they are missing ways to check properties of model transformations, even though such solutions exist for certain MTLs [$Q_{db}1$].

A desired analysis tool mentioned in the interviews is rule dependency analysis and visualisation. They believe that such a tool would provide valuable insights into the structure of written transformations and help to better *comprehend* them and their intent. *"What I would need for QVT-R, for example, in more complex cases, would be a kind of dependency graph."* (P32). Moreover two interviewees expressed the desire for tools to verify that transformations uphold certain properties or preserve certain properties of the involved models.

3.5.3.2 Code Repositories

A gap in *Tool Support* that has been brought up several times, is a central platform to share transformation libraries, much like maven-central for Java or npm for JavaScript. This tool influences *Tool Support* and the *Reusability* of MTLs.

According to two interviewees, not having a central repository where transformations, written by other developers, can be browsed or downloaded, greatly hinders their view on the *Reusability* of model transformation languages. This is because it creates a barrier for reuse. For one thing, it is difficult to find model transformations that can be reused. Secondly, mechanisms that would simplify such reuse are then also missing. *"I think what is currently missing is a catalogue or a tool like maven for having repositories for transformations so you can possibly find transformations to reuse."* (**P14**)

3.5.3.3 Debugging Tooling

Debuggers have been raised as essential tools that help with the *Comprehensibility* of written model transformations. The existence of a debugger for a given language therefore influences its *Tool Support* as well as its *Comprehensibility*.

One interviewee explained that, especially for declarative languages, where the execution deviates greatly from the definition, debugging tools would be a tremendous help in understanding what is going on. In this context, opinions were also expressed that more information is needed for debugging model transformations than for traditional programming and that the tools should therefore be able to do more. Interviewees mentioned the desire to be able to see how matchings arise or why expected matches are not produced as well as the ability to challenge their transformations with assertions to see when and why expressions evaluate to certain values. *"Demonstrate to me that this is true, show me the section of the transformation in which this OCL constraint is true or false."* (**P28**).

Valuable debugging of model transformations is mainly possible in dedicated MTLs, according to one interviewee. They argue that debugging model transformations in GPLs is cumbersome because much of the code does not relate to the actual transformation thus hampering a developers ability to grasp problems in their code.

3.5.3.4 Ecosystem

The ecosystems around a language, as well as existing ecosystems, in which model transformations languages would have to be incorporated into, were remarked as mostly limiting factors for *Productivity*, *Maintainability* as well as the perceived amount of *Tool Support*.

One interviewee explained, that for many companies, adopting a model transformation language for their modelling concerns is out of the question because it would negatively impact their *Productivity*. The reason for this are existing ecosystems, which are designed for GPL usage. Moreover, it was noted that, to fully utilise the benefits of dedicated languages many companies would need to adopt several languages to properly utilise their benefits. This is seen as hard to implement as *"people from industry have a hard time when they are required to use multiple languages"* (**P49**) making it hard for them to *maintain* code in such ecosystems.

Ecosystems surrounding MTLs have also been criticised in hampering *Productivity* and perceived *Tool Support*. Several interviewees mentioned, that developers tend to favour ecosystems where many activities can be done in one place, something they see as lacking in MTL ecosystems. One interviewee even referred to this problem as the reason why they turned away from using model transformation languages completely [$Q_{eco}2$].

This issue somewhat contrasts a concern raised by a different group of interviewees. They felt that the coupling of much of MDE tooling to Eclipse is a problem that hampers the adoption of MTLs and MDE. This coupling allows many tools to be available within the Eclipse IDE but, according to them, the problem lies in the fact that Eclipse is developed at a faster pace than what tool developers are able to keep up with, leaving much of the *Tool Support* for MTLs in an outdated state, limiting their exposure and usability [$Q_{eco}3$].

3.5.3.5 IDE Tooling

One essential tool for *Tool Support*, *Ease of Writing* and *Maintainability* of MTLs are language specific editors in IDEs.

Several interviewees mentioned, that languages without basic IDE support are likely to be unusable, because developers are used to all the quality-of-life improvements, with autocompletion and syntax highlighting being the two most important features offered by such tools. Refactoring capabilities in IDEs, like renaming, have also been raised as crucial, especially for easing the *Maintainability* of transformations.

3.5.3.6 Interoperability

How well tools can be integrated with each other has been raised as a concern for the *Tool Support* of MTLs by several interviewees.

Interviewees see a clear advantage for GPLs when talking about interoperability between different MTL tools. They believe, that due to the majority of tools being research projects, little effort is spent into standardizing those in a way that allows for interoperability on the level that is currently provided for GPLs. *"But the technologies, to combine them, it is difficult [...]"* (**P36**). One interviewee described their first hand experience with this. They could not get a MTL to process models they generated with a software architecture tool because it produced non standard UML models which could not be used by the MTL. This problem has been echoed by another interviewee who explained that many MTLs do not work with non EMF compatible models.

3.5.3.7 Tooling Awareness

A few interviewees talked about the availability of information about tools and the general awareness of which tools for MTLs exist. According to them, this strongly influences the perceived lack of *Tool Support* for model transformation languages in general.

When starting out with model transformations it can be hard to find out which tools one should use or even which tools are available at all. Two interview participants mention experiencing this first hand. They further explain that there exists no central starting point when looking for tools and tools are generally not well communicated to potential users outside of research [$Q_{awa}1$].

Another interviewee suspected that the same problem also happens the other way around. They believe that some well designed MTL tools are completely unknown outside of the companies that developed them for internal use.

3.5.3.8 Tool Creation Effort

The amount of effort, that is required to be put into the development of MTL tools, has been raised by many interviewees as a reason why *Tool Support* for MTLs is seen as lacking.

All interviewees talking about the effort involved in creating tools for MTLs agree that there is a lot of effort involved in developing tools. This is not a problem in and of itself but, when comparing tooling with GPLs interviewees felt like MTLs being at a disadvantage. The disadvantage stems from the community for MTLs being much smaller and thus having less man power to develop tools which limits the amount of tools that can be developed. Several interviewees noted, that the only solution they see for this problem is industrial backing or commercial tool vendors because *"I am keenly aware of the cost to being able to develop a good programming language, the cost of maintaining it and the cost of adding debuggers and refactoring engines. It is enormous."* (**P01**).

When comparing the actual effort for creating transformation specific tools, some interviewees explained that their experience suggests easier tool development for MTLs than for GPLs. They explained that, extracting the transformation specific information necessary for such tools out of GPL code complicates the whole process, whereas dedicated MTLs with their small and focused language core provide much easier access to such information [$Q_{tce}2$].

3.5.3.9 Tool Learnability

The learning curve for someone starting off with MTLs and MTL tools is discussed as a heavy burden to the perceived effectiveness of *Tools* and even influences *Ease of Writing*.

Several interviewees criticised the fact that when starting off with a new MTL and its accompanying tools there is little support for users. Many tools lack basic documentation on how to set them up properly and how to use them. As a result users feel lost and find it difficult to start off *writing* transformations [$Q_{tle}1$].

3.5.3.10 Tool Usability

Related to the topic of learnability, the usability of tools for model transformation languages is discussed as influencing the quality of *Tool Support* for the languages as well as the *Ease of Writing* and *Productivity*.

To fully utilise the potential of MTLs useable tools are essential. Due to their higher level of abstraction, high quality tools are necessary to properly work with them and *Write* well rounded transformations [$Q_{use}1$].

This is currently not the case when looking at the opinions of our interviewees talking about the topic of tool usability. There are tools available for people to start off with developing transformations but they are not well rounded and thus not ready for professional use, according to one interviewee. This is supported by several other interviewees opinions, many tools are faulty, which hinders the workflow and reduces *Productivity* [$Q_{use}2$]. It has also been stated that if there were high quality useable tools available, they would be used. The reality for many users is, however, more in line with the experience of one interviewee who stated that they were unable to get many tools (for bidirectional languages) to even work at all.

3.5.3.11 Tool Maturity

A reason given for many of the criticised points surrounding MTL tools is their maturity. It is said to be a pivotal factor for everything related to *Tool Support*.

The maturity of tools for model transformation languages was commented on a lot. Tools need to be refined more in order to raze many of their current faults. The fact that this is not currently done relates back to the effort that is involved with it and the limited personnel available to do so. This is highlighted in an argument made by one of the interviewees who feels, that the community should not be hiding behind the argument of maturity [$Q_{mat}1$].

3.5.3.12 Validation Tooling

Tools or frameworks to support the validation and testing of transformations written in MTLs have been discussed to influence the perceived *Tool Support* for nearly all MTLs.

Too much of the available tool support focuses solely on the writing phase of transformation development. There is little tool support for testing developed transformations, which has been raised as an area where much progress can, and has to be, made. Especially when comparing the current state of the art with GPLs, MTLs are seen as lacking [$Q_{val}1$]. Not only are there little to no tools like unit testing frameworks, there is also too few transformation specific support such as tools to specifically verify binding or helper code in ATL.

3.5.4 Choice of MTL

The choice of MTL is an obvious factor that influences how other factors, such as the *MTL Capabilities*, influence the properties of model transformation languages. However, it should be explicitly mentioned, because it has been brought up countless times by interviewees while not often being considered in literature. Depending on the chosen model transformation language its capabilities and whole makeup changes, which has strong implications on all aspects of model transformation development.

A large number of the interviewees have commented on this. They either directly raised the concern, by prefacing a discussion with a statement such as *"[...] it depends on the MTL"*, or indirectly raised the concern, when comparing specific languages that do or do not exhibit certain capabilities and properties.

3.5.5 Skills

Skills of involved stakeholders is another group of factors that does not have a direct influence on how MTLs are perceived but instead plays a passive role. Many interviewees cited skills as a limiting factor to other influence factors. They argue that insufficient user skills could hinder advantages that MTLs can provide and might even create disadvantages compared to the more well-known and commonly used GPLs.

In this section we present the different types of skills mentioned by our interviewees as being relevant to the discussion of properties of model transformation languages.

3.5.5.1 Language Skills

The skill of developers in a specific model transformation language was raised by several interviewees as critical in facilitating many of the advantages provided through the languages capabilities. So much so that, according to them, the ability of developers to use and read a language can make or break any and all advantages and disadvantages of MTLs related to *Comprehensibility*, *Ease of Writing*, *Maintainability* and *Reuseability*.

Basic skills in any language are a prerequisite to being able to use it. They are also necessary to understand written code. There is no difference between GPLs and MTLs. It was however mentioned, that developers are generally more used to the development style in general purpose languages. Thus users need to learn how to solve a problem with the functionality of the model transformation language to be able to successfully develop transformations in a MTL [$Q_{skl}1$]. This is especially relevant for complex transformations, where users are required to know of abstractions such as tracing or automatic traversal. Following on on this, one interviewee explained, that while

learning the language is a requirement, using a new library, e.g. one for developing model transformations, in a GPL also entails learning and as such this must not be regarded as a disadvantage.

For *reuse* it is also paramount for users to know what elements of a transformation can be reused through language functionality. As a result the *Reuseability* is again limited by the knowledge of users in the specific language.

Lastly, being able to *maintain* a transformation written in an MTL also requires users to know the language to be able to understand where changes need to be made [$Q_{skl}2$].

3.5.5.2 User Experience/Knowledge

Apart from mastering a used language, the amount of experience users have with said languages and techniques also play a vital role in bringing out the full potential of said languages. Our interviewees discussed this for *Ease of Writing*, *Maintainability* and *Productivity*.

One interviewee explained that, from their experience, the amount of experience developers have with a language greatly impacts their *Productivity* when using said language. The problem for MTLs that results from this is the fact that there is little incentive for a person that is trying to build up their CV to spend much time on dedicated languages such as MTLs [$Q_{exp}1$]. Developers are more inclined to learning and accumulating experience in languages that are commonly used in different companies to improve their chances of landing jobs. As a result people tend to have little to no experience in using MTLs. This in turn results in them having a harder time *developing* transformations in these languages, and the final product being of lower quality than what they could achieve using a GPL in which they have more experience in.

The problem is further exacerbated in teaching. *"Many MDSE courses are just given too late, when people are too acquainted with GPLs, and then its really hard for students to see the point of using models, modelling and MTLs, because it's comparable with languages and stuff they have already learned and worked with."* (**P06**).

3.5.6 Use Case

Similar as the *MTL* itself and stakeholder *Skills*, the concrete *Use Case* in which model transformations are being developed is another factor that does not directly influence how properties of MTLs are being assessed. Instead, interviewees often mention that, depending on the *Use Case*, other influence factors could either have a positive or negative effect.

Use cases are distinguished along three dimensions. The complexity of involved models based on their structure, the complexity of the transformation based on the semantic gap between source and target, and the size of the transformation based on the use case. Depending on which differentiation is referred to by the interviewees, the considerations look differently.

3.5.6.1 Involved (meta-) models

The involved models and meta-models can have a large impact on the transformation and can hence heavily influence the advantages or disadvantages that MTLs exhibit.

Writing transformations for well behaved models, meaning models that are well structured and documented, can be immensely productive in a MTL while 'badly' behaved models bring out problems that require well trained experts to properly solve in a MTL. The UML meta-model was put forth as an example for such a badly behaved meta-model by one interviewee. According to them, transformations involving UML models can be problematic due to templates, which are model elements that are parametrized by other model elements, and `eGenericType`. The problem with these complex model elements is often worsened by low-quality documentation [$Q_{mod}1$]. In cases where these badly behaved models are involved, many of the advantages from advanced features of MTLs can not be properly utilised without powerful tooling.

3.5.6.2 Semantic gap between input and output

Many interviewees formulate considerations based on the differentiation between 'simple' and 'complex' transformations in terms of the semantic gap that needs to be overcome. Transformations are considered simple when there is little semantic difference between the source and target models. Common comparisons read like: *"transforming boxes into circles"* (**P32**).

For simple transformations, model transformation languages are regarded as taking a lot of work off of the developers through the different language features discussed in Section 3.5.2. In more complex cases, transformations will get more complex and the developers experience gets more and more relevant, as more advanced language features need to be utilised, which can favour GPLs [$Q_{gap}1$].

Others argue that the advantages of MTLs only really come into play in more complex cases or when high level features, such as bidirectionality or incrementality, are required. The reasoning for this argument is, that in simple cases the overhead of GPLs is not that prominent. Moreover, for writing complex transformations, dedicated query languages in MTLs are regarded by some to be much better than having to manually define complex conditions and loops in a GPL.

3.5.6.3 Size

The Size of the transformation based on the *Use Case* is considered by some interviewees to be a relevant factor as well. In cases with many rules that depend on each other, MTLs are seen as having advantages [$Q_{siz}1$]. The size of transformations has been said to be a limiting factor for the use of graphical languages as enormous transformations would make graphical notations confusing. Modularisation mechanisms of languages also become a relevant feature in these cases.

3.6 Cross-Factor Findings

Based on interview responses, we developed a structure model from structural equation modeling (Weiber et al. 2021) that models interactions between the presented influence factors and the properties of model transformation languages.

Structure models depict assumed relationships between variables (Weiber et al. 2021). They divide their components into endogenous and exogenous variables. The endogenous variables are explained by the causal influences assumed in the model. The exogenous variables serve as explanatory variables, but are not themselves explained by the causal model. Exogenous variables either directly influence a endogenous variable or they moderate an influence of another exogenous variable on an endogenous variable.

Structure models are therefore well suited to provide a theoretical framework for the findings of our work. Factors identified during analysis constitute exogenous variables while MTL properties constitute endogenous variables. Moderating factors also constitute exogenous variables, with the caveat of only having moderating influences on other influences.

A graphical overview over the influences identified by us can be found in Figure 3.10. The detailed structure model is depicted in Figure 3.11.

The structure model depicts which MTL properties are influenced by which of the identified factors. For each MTL property the model also illustrates which factors moderate the influence on the property. Rectangles represent factors, rounded rectangles represent MTL properties. Below each MTL property the moderating factors for the property are displayed. Arrows between a factor and a MTL property represent the factor having an influence on the MTL property. Each influence on a MTL property is moderated by its moderating factors. The graphical representation deviates from standard presentation due to its size.

The capabilities of model transformation languages based on domain specific abstractions are aimed at providing advantages over general purpose languages. Whether these advantages are realised or whether disadvantages emerge is moderated by the *Skills* of the users, the concrete *MTL chosen* as well as the *Use Case* for which transformations are applied. Depending on these organisational factors the versatility provided by general purpose languages may overshadow advantages provided by MTLs.

Tooling can aid the usage of advanced features of MTLs by supporting developers in their endeavours beyond simple syntax highlighting. As a result, tools can further promote the advantages that stem from the domain specific abstractions. The biggest problem that tools for MTLs face is their availability and quality.

In the following we will present thorough discussions of the most salient observations based on our interviews and the presented structure model. Note that, as will be thoroughly discussed in Section 3.8.2, the observations have a limited applicability for industry use cases due to the lack of interviewees that use MTLs in an industry setting.

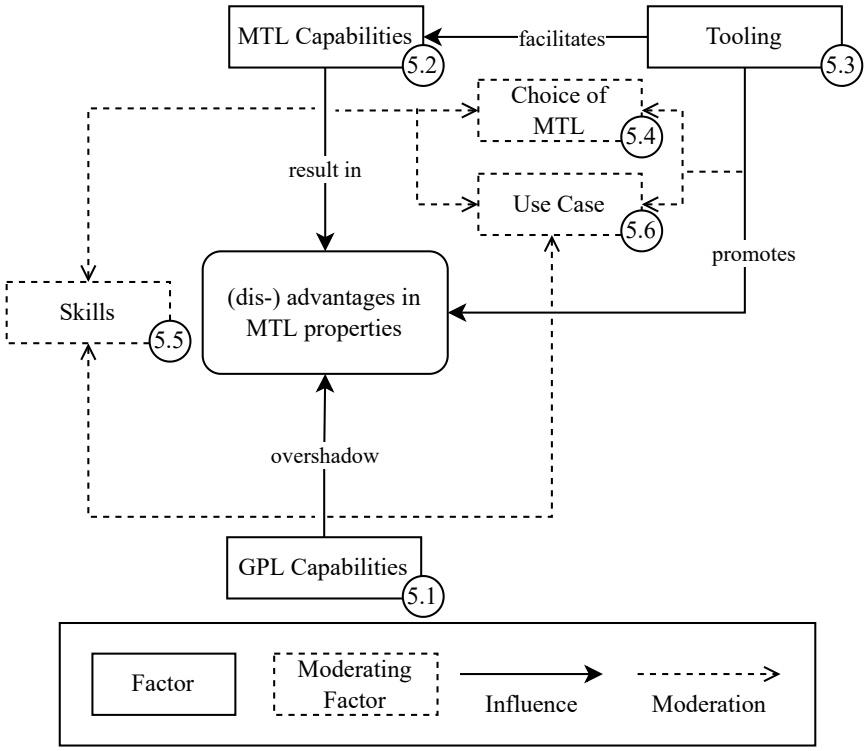


FIGURE 3.10: Graphical overview over factor influences and moderations

3.6.1 The Effects of MTL Capabilities

Capabilities of model transformation languages that go beyond what general purpose languages can offer, are regarded as opportunities for better support in development of transformations. The advantage often boils down to not having to manually implement the functionality in question when it is required. It also helps reduce clutter in transformation code, putting the mapping of input and output at the centre of attention. Moreover, they aid developers in handling problems specific to the transformation domain, such as synchronisations and the relationship of input and output values.

This does however come with its own set of limitations. Model transformation languages favour a different way of problem solving that is well suited to the problem at hand, but is unfamiliar for the common programmer. This is amplified by an education that is heavily focused on imperative programming and lacks deeper exposure to logical and functional programming. Knowledge and understanding of functional concepts would help developers when using query languages such as OCL, while logical concepts often find application in graph based transformation languages. The domain specific mechanisms in model transformation languages also make generalisations harder. This is highlighted in the discussions regarding reusability. Interviewees commonly referred to transformations as conceptionally hard to reuse because of their specificity that makes them applicable only to the use case for which they were developed.

3.6.2 Tooling Impact on Properties other than Tool Support

Tooling, or the lack thereof, is a main factor that influences how people perceive the quality and availability of usable model transformation languages. However, our interviews show that tooling also facilitates many other properties. This is because tools are not developed as an end in themselves. Tools are intended to support developers in their efforts to develop and maintain their code.

As a result, the quality of available tools is a major factor that impacts all aspects of a MTL. *"Basically all the good aids you see in a Java environment should be there even better in a MTL tool, because model transformation is so much more abstract and more relevant that you should be having tools that are again more abstract and more relevant."* (**P28**). Problems in the area of *Usability*, *Maturity* and *Interoperability* of tools have also been reported on in empirical studies on MDE in general (Mohagheghi et al. 2013a; Whittle et al. 2013).

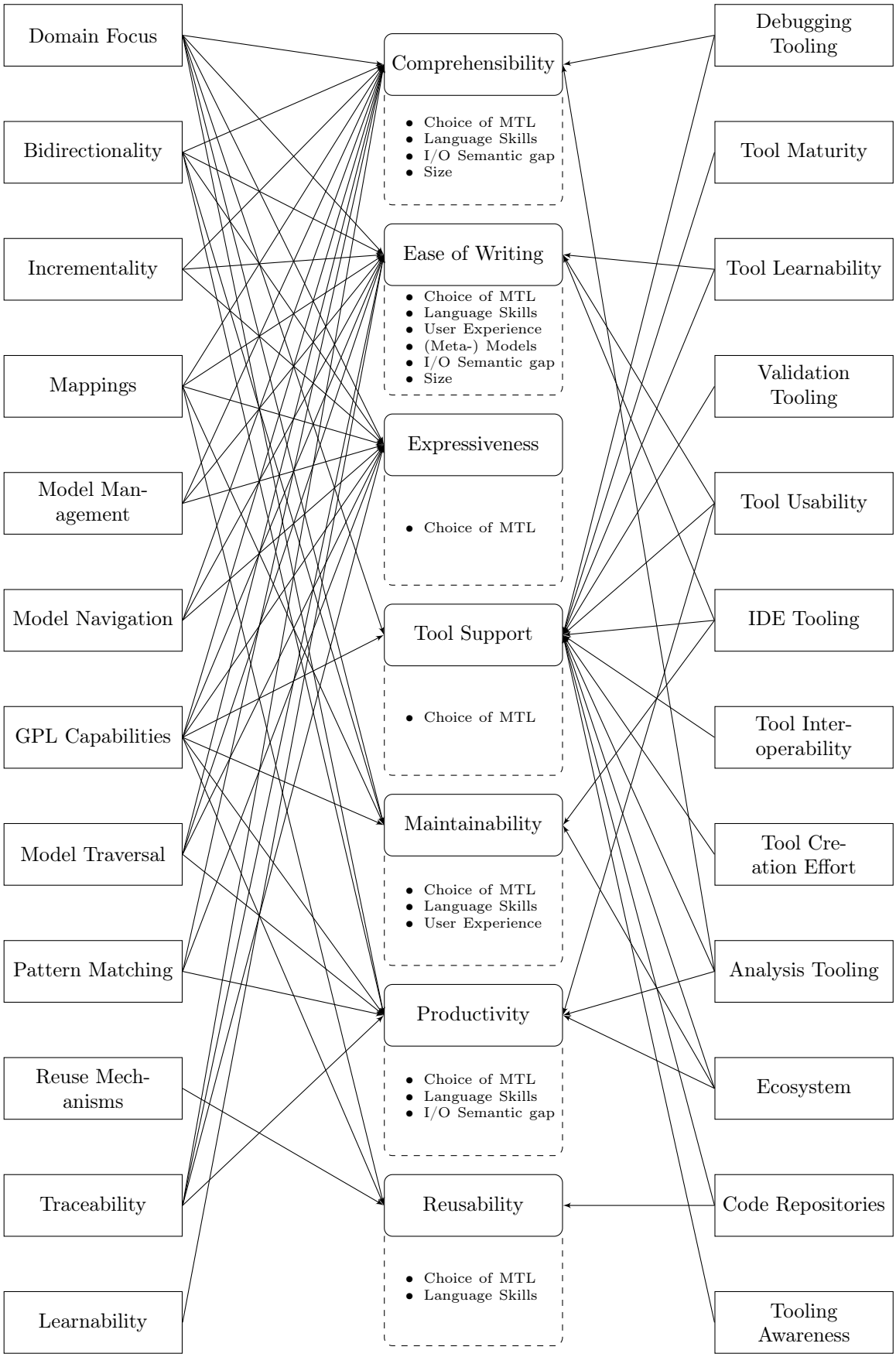


FIGURE 3.11: Structure model of influence and moderation effects of factors on MTL properties (Due to its size, the model has been made interactive using standard PDF features. Clicking a factor will show only influences of the factor. Clicking a MTL property will show only influences on the property. This view can be reset by using the reset button that appears when using the interactive features. To use the interactive features please open the PDF in *Adobe Reader* or *okular*. Other PDF viewers might work but have not been tested.)

Herein also lies the biggest problem for model transformation languages. The quality of tools is inadequate. While there do exist good and useable tools, they are far and between, only exist for certain languages and are not integrated with each other. This greatly diminishes the potential of model transformation languages because, compared to general purpose languages, developing with them can often be scattered over multiple separate workflows and tools. There do exist many tools, but most of them are prototypical in nature and only available for individual languages. This makes it hard to fully utilise the capabilities of a MTL when suitable tools only exist in theory.

The lack of good tools can be attributed mainly to the amount of work required to develop them and the comparatively small community. Moreover, there are no large commercial vendors, that could put in the required resources to develop tools of a commercially viable quality.

3.6.3 The Importance of Moderating Factors

The saying *"Use the right tool for the job"* also applies to the context of model transformations. One of the most important things to note is that depending on the context, such as *Use Case* and *developer Skills*, the right language to use can differ greatly. This was highlighted time and time again in our interviews.

Interviewees insisted that the combination of use case and the concrete implementation of a language feature significantly change how well a feature supports properties such as *Comprehensibility* or *Productivity*, i.e. the influence of factors is moderated by 'contextual' factors. For one, the implementation of a feature in a language might not fit well for the problem that needs to be solved. Or the feature is not required at all and thus could impose effort on developers that is seen as unnecessary. In our opinion, this stems from the use cases language developers intended the language for. For example, a language such as Henshin is intended for cases where patterns of model elements need to be matched and manipulated. In such cases, the features provided by Henshin can bring significant advantages over implementing the intended transformation in general purpose languages. Other use cases, where these features are not required, bring no advantage. They can even have negative effects as the language design around them might hinder users from developing a straight forward solution.

The skills and background knowledge of users is relevant, as it can greatly influence how comfortable people are in using a language. This in turn reflects on how well they can perform. This is problematic for the adoption of model transformation languages, as programmers tend to be trained in imperative, general purpose languages. As a result, a gentle learning curve is essential and the initial costs of learning need to bear fruit in adequate time. The choice of using an MTL is therefore a long term investment that is not necessarily suited for only a single project.

The considerations around *Use Case*, *Skills* and the *Choice of MTL* are not novel, but they are rarely discussed explicitly. This is concerning because almost any decision process will come back to these three factors and their sub-factors, as seen in the fact that the influence on each MTL property in Figure 3.11 is moderated by at least one of them. They provide organisational considerations that come into play before transformation development begins. Moreover, organisational concerns have already been identified as relevant factors for general MDE adoption (Hutchinson et al. 2011a,b; Whittle et al. 2013). As such they have to be at the centre of attention of researchers and language developers too.

3.7 Actionable Results

In this section we present and discuss actionable results that arise from the responses made by our interviewees and analysis thereof. Results will largely focus on actions that can be taken by researchers, because they make up the largest portion of our interview participants.

3.7.1 Evaluation and Development of MTL Capabilities

Our interviewees mentioned a large number of model transformation language capabilities and reasoned about their implications for the investigated properties of MTL. We believe the detailed results of the interviews can form a basis for further research into two key aspects:

(I) backing up the expert opinions with empirical data

(II) improving existing model transformation languages

3.7.1.1 Evaluation of MTL Capabilities and Properties

In our interviews, experts voiced many opinions on how and why factors influence the various MTL aspects examined in our study. The opinions were always based on personal experiences, experiences of colleagues and reasoning. We therefore believe, that our results provide a good insight into the communities sentiment and show that there exists consensus between the experts in many aspects. Model transformation language capabilities are considered largely beneficial, except for certain edge cases. However, empirical data to support this consensus is still missing. The lack of empirical studies into the topic of model transformation languages has already been highlighted in our preceding study (Götz et al. 2021a).

We are firmly convinced that researchers within the community need to carry out extensive empirical studies, to back up the expert opinions and to explore the exact limits that interviewees hinted at.

We envision two main types of studies, experiments and case studies. Setting up experiments that consider real-world examples with a large number of suitable participants would be optimal but is hard to achieve. Introducing large transformation examples in experiments is very time-consuming and requires that all participants are experts in the languages used. In addition, recruiting appropriate participants is a generally difficult in software engineering studies (Rainer et al. 2021). We recommend using existing transformations as the object of study in experiments instead. This enables the analysis of complex systems to generate quantitative data without involving human subjects.

If the assessments and experiences of developers are to be the central object of study, we recommend to set up case studies. This allows researchers to study effects in complex, real-world settings over a longer period of time. This is important because the exact effects of, for example, the use of a certain language feature often only become apparent to developers after a long period of use. Case studies of research projects or even industrial transformation systems can thus be used to obtain detailed information on the impact of the applied technologies.

To design such studies, our results can form an important basis.

(a) Empirical factor evaluation. How and under which circumstances the factors we have identified affect MTL properties needs to be comprehensively evaluated. Here we envision both qualitative and quantitative studies that focus on the impact of a single factor or a group of related factors. These could, for example, make comparisons between cases where a factor does or does not apply. The results of such studies can help language developers make decisions about features to include in their MTLs.

Our interviews provide extensive context that should be taken into account in the study design and interpretation of results. For example, our interviews show that the semantic gap between input and output defines a relevant context that needs to be considered. For this reason, when investigating the advantages and disadvantages of mappings, transformations involving models with different levels of semantic gaps between input and output have to be used, to be able to fully evaluate all relevant use cases. Some transformations need to contain complicated selection conditions or complex calculations for attributes while others need to have less complicated expressions. Researchers can then evaluate how well mappings in a language fit the different scenarios to aid in providing a clear picture of their advantages and disadvantages.

(b) Empirical MTL property evaluation. What advantages or disadvantages MTLs really have is still up for debate. We believe that the credibility of research efforts on MTL can be greatly improved with studies that provide empirical substantiation to the speculated properties. Advances like those made by Hebig et al. (2018) are rare and further ones, based on real world examples, must be carried out.

Our results can also make a valuable contribution to such studies. The factors we have identified as influencing a property can be taken into account in studies from the outset. They can be used to formulate null hypotheses on why a MTL is superior or inferior to a GPL when considering one specific property.

For example, a study that is interested in investigating the *Comprehensibility* of MTLs compared to GPLs can find a number of factors in our results that need to be taken into account. Such factors include tracing mechanisms, mappings or pattern matching capabilities. Researchers can consciously decide which of them are relevant for the transformations used in the study and what impact their presence or absence has on the study results. Based on these considerations hypotheses can be formed.

A recent study we conducted provides an example of how these considerations can be used to expand the body of empirical studies on this topic (Höppner et al. 2021) By focusing the investigation on *Mappings*, *Model Navigation* and *Tracing* we were able to present clear and focused results for comparing and explaining differences in the expressiveness of transformation code written in ATL and Java. We concentrated our analysis on these factors because they all influence Expressiveness according to our interviews.

Such considerations should of course be part of any proper study, but our results provide a basis that can be useful in ensuring that no relevant factors are overlooked.

(c) Influence Quantification. Lastly, the results of this study should be quantified. The design of the reported study makes quantification of the importance of factors and their influence strengths impossible. However, such quantification is necessary to prioritise which factors to focus on first, both for assessment and for improvement. We intend to design and execute such a study as future work to this study.

We can use structural equation modelling methods (Weiber et al. 2021) to quantify the factors and their influences because we already have a structure model. We plan to use an online survey to query users of MTLs from research and industry about the amount they use different language features, their perception of qualitative properties of their transformations and demographic data surrounding use-case, skills & experience and used languages. The responses are used as input for universal structure modelling (USM) (Buckler et al. 2008) based on the structural equation model developed from the interview responses.

USM is used to estimate the influence and moderation weights of all variables within the structure model. We can therefore use it to produce quantified data on the influence and moderation effects of identified factors.

We are confident that the approach of using a survey to quantify interview results, can complement the current results, because several of the authors have had positive experiences applying it (Juhnke et al. 2020; Liebel et al. 2018).

3.7.1.2 Improving MTL Capabilities

To improve current model transformation languages the criticisms articulated by interviewees can be used as starting points for enhancements and innovation. There are several aspects that are considered to be problematic by our interviewees.

(d) Improve reuse mechanism adoption. Reuse mechanisms in model transformation languages are one aspect where interviewees saw potential for improvement (see Section 3.5.2.10). Languages that do not currently possess mature reuse mechanisms can adopt them to become more usable. For the adoption of mature reuse mechanisms in MTLs we see the languages developers as responsible.

(e) Reuse mechanism innovation. Innovation towards transformation specific reuse mechanisms, as has been requested by some participants (see Section 3.5.2.10), should also be advanced. This topic was discussed at length during the interviews on the statement *"Having written several transformations, we have identified that current MTLs are too low a level of abstraction for succinctly expressing transformations between DSLs, because they demonstrate several recurring patterns that have to be reimplemented each time."* in Question Set 3.

Interviewees pointed out a need for reuse mechanisms that allow transformations to adapt to differing inputs and outputs. It would be conceivable to define transformation rules, or parts of them, independently of concrete model types, similar to generics in GPLs. This would allow development of generic transformation 'templates' of common transformation patterns. One pattern, for example, could be finding and manipulating specific model structures, like cliques, independent of the *concrete* model elements involved. Such templates can then be reused and adapted in all transformations where the pattern is required.

We believe, that innovating such new transformation specific reuse mechanisms is a community wide effort that needs to be taken on in order to make them more widely usable.

(f) Improving MDSE education. The *Learnability* of MTLs has also been a point of criticism. We believe, that more effort needs to be put into the transfer of knowledge for MDSE and its techniques like model transformations and MTLs. This believe is supported by the findings of Hutchinson et al. (2011a). They also identified the lack of MDSE knowledge as a limiting factor for the adoption of the approach.

People need to come into contact with the principles earlier so that the inhibition threshold to apply them is lower. This was also remarked by interviewees when discussing the *Learnability* (see Section 3.5.2.11). More focus needs to be given to modelling and modelling techniques in software engineering courses. This is especially important since the skill of users has been said to be a largely impactful factor upon which many of the advantages from other MTL capabilities rely. Furthermore, there exist studies such as the one by Dieste et al. (2017), which detected a connection between the experience of developers with a language and their productivity as well as the code quality of the resulting programs.

To achieve this, we believe, that the researchers from the community, in their role as higher education teachers and university staff, need to become active. They should advocate for teaching the concepts of MDSE and the advantages/disadvantages in undergraduate studies in computer science study programmes. This view is shared by Samiee et al. (2018). Particularly, it should be taught that models can be used for more than documentation purposes, e.g., code generation, simulations early in the development cycle, test case generation. These other uses are widely and successfully employed in the domain of cyber-physical systems according to Bucchiarone et al. (2021). Hence, it might be beneficial to include industrial modelling tools like Matlab/Simulink/Stateflow from this domain in addition to standard UML tools in undergraduate courses. Furthermore, we successfully used simulation frameworks for autonomous cars, like Carla (see Dosovitskiy et al. (2017)), in the past as targets for student projects when teaching courses on the development of modeling languages and model transformations. For example, the students devised a state machine language and code generator targeting the simulation framework to develop an automatic parking functionality. Model transformations were developed to flatten hierarchical state machines to non-hierarchical state machines prior to code generation.

(g) Increase knowledge retention. It is also difficult to get to grips with the subject matter in general, as information on it is much harder to obtain than on general purpose programming (see Section 3.5.3.7). This starts with the fact that, we found websites on MTLs to often be outdated or unappealing and lack good tutorials and comprehensible documentation. These points need to be fixed, by the language developers, to provide potential users with better resources to combat the perceived steepness of the learning curve. More active community involvement is also conceivable here. Users of MTLs could invest time in creating documentation and keeping it up-to-date. The possibility of this working and producing good results can be seen in examples such as the arch-linux wiki[4].

(h) Improve community outwards presentation. The model transformation community is small. In our opinion this leads to less innovation and poses the danger of entrenched practices. The problem is not limited to small communities as seen by, for example, the risk averse movie industry or low innovation automotive industry. An improved outwards presentation of the technology of model transformations can help alleviate the problem of limited human resources. The current hype surrounding low-code-platforms can be used to inspire young and aspiring researchers to contribute to its underlying concepts such as model transformations.

(i) Improve industry outreach and cooperation. We think it is also paramount to pursue industry cooperation to gauge industrial needs in order to facilitate more industrial adoption of MTLs. Here ambitious studies are required that attempt to provide the community with clear requirements specific domains of industry have for MDE and transformation languages, as well as to show for which domains application is reasonable at all. There exist some field studies by Mohagheghi et al. (2008, 2013a) and Staron (2006) but they are far and in between and do not focus on the transformation languages involved. The research community can attempt to organize solutions for these requirements based on such field study and industry research. However, for such industry cooperation to be possible, a focused community outreach is required. There are notable advancements in this direction e.g. MDENet[5], but they are still in their infancy and require more involvement by the research community.

(j) Provide *representative* model transformation languages. To provide reasonable evidence that model transformation languages can be competitive against GPLs there also needs to be heavy focus on providing less prototypical and more pragmatic and useable transformation languages (see Section 3.6.2). To that end only a few selected languages should be attempted to be made production ready, potentially through further industry cooperation. MTLs could be integrated into commercial modelling tools in order to be able to process models programmatically in the tool.

[4] `wiki.archlinux.org`
[5] `community.mde-network.org`

Alternatively, few modern standardised MTLs could be promoted by the community. Since such a decision has far-reaching effects, a central, community wide respected body is needed. The OMG could possibly take action for this as they are already deciding on community impacting standards.

The QVT standard was an ambitious push in this direction. However, we believe that the initiative needs a fresh take, given the findings of the last 20 years of research. This idea is supported by several interviewees who considered QVT to be bloated and outdated. Especially in the areas of bidirectional and incremental transformations we see huge potential. Furthermore, relying more on declarative approaches for defining uni-directional transformations should also be considered. This trend can also be observed in the field of GPLs with the introduction of more and more functional concepts into them.

Innovation in prototypical languages should then be thoroughly evaluated for its usefulness before adoption into one of the flagship languages. It is not the task of research to produce industry ready languages, but setting up the environment and using these languages should not be more complicated than for any general purpose programming language.

(k) Research legacy integration. The integration of MTLs into existing legacy systems has been remarked as a huge entry barrier for industry adoption (see Sections 3.5.2.1 and 3.5.3.4). We believe this stems from a lack of techniques that facilitate gradual integration of modelling technology into existing systems and infrastructure. This is highlighted by the fact that basic literature such as that by Brambilla et al. (2017) does not contain any suggestions to this end. To combat this, we propose a dedicated branch of MDE research focused on developing tools and processes to integrate model driven techniques into legacy systems.

We envision distinct guidelines and processes on how to integrate transformations and transformation concepts into existing systems. There should be terms of reference as to which types of system components lend themselves to the use of model transformations. Furthermore, descriptions of which transformations and which transformation languages are suitable for which type of use case are also required. Having such guides can reduce the barrier of entry, because they provide a clear course of action when trying to (gradually) adopt the paradigm.

This also includes accessible GPL bindings for applying model transformation concepts. They can be used to gradually replace system components that can benefit from the use of transformations. This can be done without the overhead of integrating a new language and intermediate models. One example for this is DresdenOCL, a OCL dialect that can be used on Java code (Demuth et al. 2009).

3.7.2 Steps Towards Solving the Tooling Problem

From our interviews, we have to conclude that the biggest weak point of model transformation languages is their *Tool Support*.

The two biggest tooling gaps that we were able to identify are:

(I) many necessary tools do not exist

(II) existing tools lack user-friendliness and are not compatible with each other

We hope that our work can be a starting point in counteracting these drawbacks.

(l) Provide essential tooling. In our view, tooling of flagship model transformation languages needs to be extended to include all the essential tools mentioned in the interviews to make MTLs production and industry ready. This includes useable *Editors*, *Debuggers* and *Validation* or *Analysis* tools. At best all such tools for a language should be useable within one IDE. One way language developers can help with this task is by implementing the *Language Server Protocol* (LSP) (Microsoft 2022) or its graphical counterpart GLSP (Eclipse Foundation 2022) for their MTL. This would greatly improve the ability of tool developers to create and distribute tooling.

(m) Develop transformation specific debugging. As mentioned by our interviewees, for debuggers there is a need for model transformation specific techniques. Troya et al. (2022) showed that there are numerous advances in this area like by Ege et al. (2019), Hibberd et al. (2007), and Wimmer et al. (2009) but none of them have led to well rounded debuggers yet. Further effort by researchers active in this area is therefore required. They should strive to develop their approaches to a point where they can be productively used to demonstrate their usefulness for a productive transformation development.

(n) Improve tool usability. Most importantly, a lot of effort needs to be put into improving the usability of MTL tools. Our interviews have shown, that unusable tools are the most off putting

factor that hampers wider adoption. To combat this, we believe usability studies to be essential. Studies to identify usability issues in the likeness of what is proposed by Pietron et al. (2018) can be used to gain insights into where problems originate from and how to improve them. Such studies have already been successfully utilised for other MDE related tooling (Stegmaier et al. 2019). We therefore need more researchers from the community to get involved in designing and conducting usability studies for tooling surrounding MTLs.

We think the results of usability studies can also provide useful lessons learned for tool developers to make tools more usable from the beginning. The overall goal must be to find out what needs to be changed or improved in MTL tools to make their adoption significant. Industrial efforts to provide proper tool support can then be based on these results and the existing, usable, tools. This adoption is necessary because, in our view, the human resources required for providing adequate long-term support for the tools can only be provided by commercially operating companies. Such long term support is necessary so that model transformation languages, and their accompanying tools, can gain a foothold in the fast-moving industrial world. The industrialisation of MTL tooling was also proposed during an open community discussion detailed by Burgueño et al. (2019).

The goal should be to provide well rounded, all-in-one solutions that integrate all necessary tooling in one place, to make development as seamless as possible. The appropriateness of this has been shown by Jonkers et al. (Jonkers et al. 2006).

(o) Limit-test internal MTLs. A different approach that should be further explored is the attempt to thoroughly embed an internal model transformation language in a main stream GPL as done by Hinkel et al. (2019b). The advantage of this approach is the ability to inherit tooling of the host language (Hinkel et al. 2019b) and it allows general purpose developers to apply their rich pool of experience. However, there are some drawbacks to this approach, as discussed in Section 3.5. The amount of tooling that can be properly integrated is limited and it is more difficult to develop transformation specific tooling for internal languages as it is hard to extract the required information from the code. For this reason, we think, the required tools should be known at design time and the language has to be designed to expose all the required information while not imposing this as an additional burden on developers. Researchers that plan to develop an internal model transformation language should therefore thoroughly asses the tool requirements for the use case for which they intend to develop their language.

3.8 Threats to validity

Our interview study was carefully designed, and followed reputable guidelines for preparation, conduction and analysis. Nonetheless there are some threats to validity that need to be discussed to provide a complete picture of our study and its results.

3.8.1 Internal Validity

Internal validity describes the extent to which a casual conclusion based on the study is warranted. The validity is threatened by manual errors and biases of the involved researchers throughout the study process.

Errors could have been introduced during the transcription phase and during the analysis of the data since both steps were conducted by a single author at a time.

To prevent transcription errors, all transcripts were re-examined after completion to ensure consistency between the transcripts and audio recordings.

To minimize possible confirmation biases introduced during analysis and categorisation of interviewee statements, random samples were checked by other authors to find possible discrepancies between the authors assessments on statements. In cases where such discrepancies were encountered, thorough discussions between all authors were conducted to find a consensus that was then applied to all transcripts containing similar considerations.

Lastly there is the potential of misinterpretation of interviewees responses during analysis. While we carefully stuck to interpret statements literally during coding, there are words and phrases that have overloaded meanings. During the interviews, it would always be necessary to ask exactly what meaning interviewees used, but this was not always possible. Therefore the threat could not be mitigated completely as contextual information was required to interpret interviewees responses in some cases.

3.8.2 External Validity

External validity describes the extent to which the results of a study can be generalised. In our interview study this validity is threatened by our interview participant assortment ,which is a result of our sampling and selection method.

We utilise convenience sampling interviewing any and all people that respond to our emails. This can limit how representative the final group of interviewees is of the target population. The issue here is that we do not know much about the makeup of the target population. It is therefore difficult to assess how much the group of participants deviates from a representative set.

Using research publications as the starting point for participant selection also introduces a bias towards users from research. This can be clearly seen in Figure 3.6. There is an apparent lack of participants from industry which limits the applicability of our results to industrial cases. This threat is somewhat mitigated by the fact that half of all participants do have at least some contact with industry, either through research projects in conjunction with industry or by having worked in industry.

Another threat to external validity relates to model to text (M2T) transformations. Only a few of our participants stated to have experience in applying M2T transformations. This is a result of how the initial set of potential participants was constructed. The search terms used in the SLR miss terms that relate to M2T such as 'code generation' or 'model to text'. This limitation was opted into to avoid having to differentiate between the two transformation approaches during analysis. Moreover, the consensus during discussions was that we were talking about model to model transformations. As such, our results can not be applied to the field of model to text languages.

Lastly, there is the threat of participation bias. Participants may disproportionately posses a trait that reduces the generalisability of their responses. People that view model transformation languages positively might be more inclined to participate than critics. We can not preclude this threat, but, the amount of critique we were able to elicit from the interviews suggests the effects from this bias to be weak. Other impacts of this bias are discussed in Section 3.8.4.

3.8.3 Construct Validity

Construct validity describes the extent to which the right method was applied to find answers for the research question. This validity is threatened by an inappropriate method that allows for errors.

Prior to conducting our research much work went into designing a proper framework to use. Here we relied on reputable existing guidelines for both the interview and analysis parts of this work. We used open ended questions to facilitate an open space for participants to bring forth any and all their opinions and considerations for the topic at hand. The statements used as guidance can however present a potential threat since their wording could introduce an unconscious bias in our interviewees. To combat this we selected broad statements as well as used both a negative and a positive statement for each discussed property. However, there is a chance that these measures were not fully sufficient.

Lastly, it can not be excluded that some relevant factors have not been raised during our interviews. We have interviewed a large number of people, but this threat cannot be overcome because of the study design and the open nature of our research question.

3.8.4 Conclusion Validity

Conclusion validity describes the extent to which our results stem from the investigated variables and are reproducible. Here, the biases of our participants represent the biggest threat.

It is safe to assume that people who do research on a subject are more likely to see it in a positive light and less likely to find anything negative about it. As such there is the possibility that too little negative impact factors were considered and presented. However, we found that the people we interviewed were also able to deal with the topic in a very critical way. We therefore conclude that the statements may have been somewhat more positively loaded, but that the results themselves are meaningful.

3.9 Related Work

To the best of our knowledge, there exists no other interview study that focuses on influence factors on the advantages and disadvantages of model transformation languages. Nonetheless there exist several works that can be related to our study. The related work is divided into empirical studies on model transformation languages, empirical studies on model transformations in general and interview studies on MDE.

3.9.1 Empirical studies on model transformation languages

A structured literature review we conducted (Götz et al. 2021a) forms the basis for the work presented in this paper. The goal of the reported literature review was to extract and categorize claims about the advantages and disadvantages of model transformation languages as well as to learn and report on the current state of evaluation thereof. The authors searched over 4000 publications to extract a total of 58 publications that directly claim properties of model transformation languages. In total the authors found 137 claims and categorized them into 15 properties. From their work the authors conclude that while many advantages and disadvantages are claimed little to no studies have been executed to verify them. They also point out a general lack of context and background information on the claimed properties that hinders evaluation and prompts scepticism.

Burgueño et al. (2019) report on a online survey as well as a subsequent open discussion at the 12th edition of the International Conference on Model Transformations (ICMT'2019) about the future of model transformation languages. Their goal for the survey was to identify reasons as to why developers decided for or against the use of model transformation languages and what their opinion on the future of these languages was. At ICMT'2019 where the results of the survey were presented they then moderated an open discussion on the same topic. The results of the study indicate that MTLs have fallen in popularity compared to at the beginning of the decade which they attribute to technical issues, tooling issues, social issues and the fact that general purpose languages have assimilated ideas from MTLs making GPLs a more viable option for defining model transformations. While their methodology differed from our interview study, the results of both studies support each other. However the results of our study are more detailed and provide a larger body of background knowledge that is relevant for future studies on the subject.

The notion of general purpose programming languages as alternatives to MTLs for writing model transformations has been explored by Hebig et al. (2018) and by us (Götz et al. 2021b). Hebig et al (Hebig et al. 2018) report on a controlled experiment where student participants had to complete three tasks involved in the development of model transformations. One task was to comprehend an existing transformation, one task involved modifying an existing transformation and one task required the participants to develop a transformation from scratch. The authors compare how the use of ATL, QVT-O and the general purpose language Xtend affect the outcome of the three tasks. Their results show no clear evidence of an advantage when using a MTL compared to a GPL but concede the narrow conditions under which the observation was made. The study provides a rare example of empirical evaluation of MTLs of which we suggest that more be made. The narrow conditions the authors struggled with could be alleviated by follow-up studies that draw from our results for defining their boundaries.

In a recent study by us (Götz et al. 2021b) we put the value of model transformation language into a historical perspective and drew from the preliminary results of the interview study for the study setup. We compare the complexity of a set of 10 model transformations written in ATL with their counterparts written in Java SE5, which was current around 2006 when ATL was first introduced, and Java SE14. The Java transformations were translated from the ATL modules based on a predefined translation schema. The findings support the assumptions from Burgueno et al. (Burgueño et al. 2019) in part. While we found that newer Java features such as Streams allow for a significant reduction in cyclomatic complexity and lines of code the relative amount of complexity of aspects that ATL can hide stays the same between the two Java versions.

Gerpheide et al. (2016) use an exploratory study with expert interviews, a literature review and introspection to formalize a quality model for the QVT-O model transformation standard by the OMG. They validate their quality model using a survey and afterwards use the quality model to identify tool support need of transformation developers. In a final step the authors design and evaluate a code test coverage tool for QVT-O. Their study is similar to ours in that they also relied on expert interviews for their goal. The end goal of the study however differs from ours as they used

the interviews to design a quality model for QvT-O while we used it to formulate influence factors on quality attributes of model transformation languages

Lastly there are two study templates for evaluating model transformation languages which have yet to be used for executing actual studies. Kramer et al. (2016) present a template for a controlled experiment to evaluate the comprehensibility of model transformation languages. Their approach suggests the use of a paper-based questionnaire to let participants prove their ability to understand what a transformation code snippet does. The influence of the language in which the code is written on comprehension speed and quality is then measured by comparing the average number of correct answers and the average time spent to fill out the questionnaires. Strüber et al. (2016) propose a controlled experiment for comparing the benefits and drawbacks of the reusability mechanisms *rule refinement* and *variability-based rules*. They suggest that the value of the reusability of an approach can be measured by looking at the comprehensibility of the two mechanisms as well their changeability, which is measured through bug-fixing and modification tasks. The results of studies executed based on both study templates could draw from our results for their final design and would provide valuable empirical data, a gap we identified in this and the preceding literature review.

3.9.2 Empirical studies on model transformations

Tehrani et al. (2016) executed an interview based study on requirements engineering for model transformation development. Their goal was to identify and understand the contexts and manner in which model transformations are applied as well as how requirements for them are established. To this end they interviewed 5 industry experts. From the interviews the authors found that out of 7 transformation projects only a single project was developed in an already existing project while all other projects were created from scratch. Their findings are relevant to our work since participants in our study agreed that it is hard to integrate MTLs in existing infrastructures. Whether the fact that MTLs are hard to integrate was an influence factor for the projects considered in the interview study by them is however not clear.

Groner et al. (2020) utilize an exploratory mixed method study consisting of a survey and subsequent interviews with a selection of the survey participants to try and evaluate how developers deal with performance issues in their model transformations. They also asses the causes and solutions that developers experienced. The survey results show that over half of all developers have experienced performance issues in their transformations. While the interviews allowed the authors to identify and categorize performance causes and solutions into 3 categories: *Engine related*, *Transformation definition related* and *Model related*. From the interviews they were also able to identify that tools such as useable profilers and static analyses would help developers in managing performance issues. The results of their study highlight that some of the factors identified by us are also relevant for other MTL properties not directly investigated in our study.

3.9.3 Interview studies on model driven software engineering

There are numerous publications and several groups of researchers that have carried out large scale, in-depth empirical studies on model driven engineering as a whole. We focus on a selection of those that have relation to our study in terms of findings.

Whittle, Hutchinson, Rouncefield et al. used questionnaires (Hutchinson et al. 2011a, 2014) and interviews (Hutchinson et al. 2011a,b, 2014; Whittle et al. 2013) to elicit positive and negative consequences of the usage of MDE in industrial settings. Apart from technical factors related to tooling they also found organisational and social factors that impact the adoption and efficacy of MDE. Several of their findings for MDE in general coincide with results from our study. Related to tooling they too found the factors of *Interoperability*, *Maturity* and *Usability* to be influential. Moreover, on the organisational side, the small amount of people that are knowledgable in MDE techniques and the problem of integrating into existing infrastructure are also results Whittle et al. found. Lastly, developers being more interested in using techniques that help build their CV was identified by them as a limiting factor too.

Staron (2006) analyse data collected from a case study of MDE adoption at two companies where one company withdrew from adopting MDE while the other was in the process of adoption. Their findings suggest that legacy code was a main influence factor on whether a cost efficient MDE adoption was possible. This observation is consistent with our findings that integrating MTLs into existing infrastructures has a negative impact on the *Productivity* that can be achieved with MTLs.

The research group surrounding Mohagheghi also carried out multiple empirical studies on MDE, focusing on factors for and consequences of adoption thereof. They use surveys and interviews at several companies (Mohagheghi et al. 2013a,b) as well as a literature review (Mohagheghi et al. 2008) for this purpose. In addition to mature tooling, factors identified by the authors are usefulness, ease of use and compatibility with existing tools. Similar to statements by our interviewees, they also found that MDE is seen as a long term investment. It is not well suited for single projects.

Lastly, Akdur et al. (2018) report on a large online survey of people from the domain of embedded systems industry. They too found tools surrounding MDE to be a major factor. Another interesting finding by them was that UML models are by far the most commonly used models. This is of relevance to our results since one of our interviewees pointed out, that the makeup of some UML models can have detrimental effects on the usefulness of MTLs.

The results of all presented research groups show, that many of the factors we identified for MTLs also apply to MDE in general which provides additional confidence in our results and shows that advancements in these areas would have a high impact.

3.10 Conclusion

There are many claims about the advantages and disadvantages of model transformation languages. In this paper, we presented and argued the detailed factors that play a role for such claims. Based on interviews with 56 participants from research and industry we present a **structure model** of relevant factors for the *Ease of writing*, *Expressiveness*, *Comprehensibility*, *Tool Support*, *Productivity*, *Reuse* and *Maintainability* of model transformation languages. For each factor we detail which properties they influence and **how** they influence them. We have identified two types of factors. There are factors that have a **direct impact** on said properties, e.g. different capabilities of model transformation languages like automatic trace handling. And there are factors that define a **context** whose characteristics *moderate* the the impact of the former factors, e.g. the *Skills* of developers.

Based on the interview results we suggest a number of tangible actions that need to be taken in order to convey the viability of model transformation languages and MDSE. For one, empirical studies need to be executed to provide proper substantiation to claimed properties. We also need to see more innovation for transformation specific reuse, legacy integration and need to improve outreach and presentation to both industry and academia. Lastly, efforts must be made to improve tool support and especially tool usability for MTLs.

For all of the suggested actions, our results can provide detailed data to draw from.

Chapter 4

Paper C

Traceability and Reuse Mechanisms, the most important Properties of Model Transformation Languages

S. Höppner, M. Tichy

under review in *Empirical Software Engineering (EMSE)*
Springer Nature

Abstract

Context
Dedicated model transformation languages are claimed to provide many benefits over the use of general purpose languages for developing model transformations. However, the actual advantages and disadvantages associated with the use of model transformation languages are poorly understood empirically. There is little knowledge and even less empirical assessment about what advantages and disadvantages hold in which cases and where they originate from. In a prior interview study, we elicited expert opinions on what advantages result from what factors surrounding model transformation languages as well as a number of moderating factors that moderate the influence.
Objective
We aim to quantitatively asses the interview results to confirm or reject the influences and moderation effects posed by different factors. We further intend to gain insights into how valuable different factors are to the discussion so that future studies can draw on these data for designing targeted and relevant studies.
Method
We gather data on the factors and quality attributes using an online survey. To analyse the data and examine the hypothesised influences and moderations, we use universal structure modelling based on a structural equation model. Universal structure modelling produces significance values and path coefficients for each hypothesised and modelled interdependence between factors and quality attributes that can be used to confirm or reject correlation and to weigh the strength of influence present.
Results
We analyzed 113 responses. The results show that the MTL capabilities Tracing and Reuse Mechanisms are most important overall. Though the observed effects were generally 10 times lower than anticipated. Additionally, we found that a more nuanced view of moderation effects is warranted. Their moderating influence differed significantly between the different influences, with the strongest effects being 1000 times higher than the weakest.
Conclusion
The empirical assessment of MTLs is a complex topic that cannot be solved by looking at a single stand-alone factor. Our results provide clear indication that evaluation should consider transformations of different sizes and use-cases that go beyond mapping one elements attributes to another. Language development on the other hand should focus on providing practical, transformation specific reuse mechanisms that allow MTLs to excel in areas such as maintainability and productivity compared to GPLs.

4.1 Introduction

Model driven engineering (MDE) envisions the use of model transformations as a main activity during development (Sendall et al. 2003). When practising MDE, model transformations are used for a wide array of tasks such as manipulating and evolving models (Metzger 2005), deriving artefacts like source code or documentation, simulating system behaviour or analysing system aspects (Schmidt 2006).

Numerous dedicated model transformation languages (MTLs) of different form, aim and syntax (Kahani et al. 2019) have been developed to aid with model transformations. Using MTLs is associated with many benefits compared to using general purpose languages (GPLs), though little evidence for this has been brought forth (Götz et al. 2021a). The number of claimed benefits is enormous and includes, but is not limited to, better *Comprehensibility*, *Productivity* and *Maintainability* as well as easier *development* in general (Götz et al. 2021a). The existence of such claims can partially be attributed to the advantages that are ascribed to domain specific languages (DSLs) (Hermans et al. 2009; Johannes et al. 2009).

In a prior systematic literature review, we have shown that it is still uncertain whether these advantages exist and where they arise from (Götz et al. 2021a). Due to this uncertainty it is hard to convincingly argue the use of MTLs over GPLs for transformation development. This problem is exacerbated when considering recent GPL advancements, like Java Streams, LINQ in C# or advanced pattern matching syntax, that help reduce boilerplate code (Höppner et al. 2021) and have put them back into the discussion for transformation development. Even a community discussion held at the 12th edition for the International Conference on Model Transformations (ICMT'19) acknowledges GPLs as suitable contenders (Burgueño et al. 2019). Moreover, the few existing empirical studies on this topic provide mixed and limited results. Hebig et al. found no direct advantage for the development of transformations, but did find an advantage for the comprehensibility of transformation code in their limited setup (Hebig et al. 2018). A study conducted by us, found that certain use cases favour the use of MTLs, while in others the versatility of GPLs prevails (Höppner et al. 2021). Overall there exists a gap in knowledge in what the exact benefits of MTLs are, how strong their impact really is and what parts of the language they originate from.

To bridge this gap, we conducted an interview study with 56 experts from research and industry to discuss the topic of advantages and disadvantages of model transformation languages (Höppner et al. 2022a). Participants were queried about their views on the advantages and disadvantages of model transformation languages and the origins thereof. The results point towards three main-areas that are relevant to the discussion, namely *General Purpose Languages Capabilities*, *Model Transformation Languages Capabilities* and *Tooling*. From the responses of the interviewees we identified which claimed MTL properties are influenced by which sub-areas and why. They also provided us with insights on moderation effects on these interdependencies caused by different *Use-Cases*, *Skill & Experience levels* of users and *Choice of Transformation Language*.

All results of the interview study are qualitative and therefore limited in their informative value as they do not provide indication on the strength of influence between the involved variables. It is also not clear whether the influence model is complete and whether the views pretended by the interview participants withstand community scrutiny. Therefore they only represent an initial data set that requires a quantitative and detailed analysis.

In this paper, we report on the results of a study to *confirm or deny* the interdependencies hypothesised from our interview results. We provide *quantification of the influence strengths* and *moderation effects*. To ensure a more complete theory of interactions, we also present the results of exploring interdependencies between factors and quality properties not hypothesised in the interviews.

Due to limited resources, this study focuses on the effects of *MTL capabilities* (namely Bidirectionality, Incrementality, Mappings, Model Management, Model Navigation, Model Traversal, Pattern Matching, Reuse Mechanisms and Traceability) on *MTL properties* (namely Comprehensibility, Ease of Writing, Expressiveness, Productivity, Maintainability and Reusability and Tool Support) in the context of their *uses-case* (namely bidirectional or unidirectional, incremental or non-incremental, meta-model sanity, meta-model, model and transformation size and semantic gap between input and output), the *skills & experience* of users and *language choice*. Further studies can follow the same approach and focus on different areas. Descriptions for all MTL capabilities and MTL properties can be found in Section 4.2 and thorough explanations can be found in our previous works (Götz et al. 2021a; Höppner et al. 2022a).

The goal of our study is to provide quantitative results on the influence strengths of interdependences between model transformation language *Capabilities* and claimed *Quality Properties* as perceived by users. Additionally we provide data on the strength of moderation expressed by *contextual properties*. The study is structured around the hypothesised interdependencies between these variables, and their more detailed breakdown, extracted from our previous interview study. Each presumed influence of a *MTL capability* on a *MTL property* forms one hypothesis which is to be examined in this study. All hypotheses are extended with an assumption of moderation by the context variables. The system of hypotheses that arises from these deliberations is visualised in a structure model, which forms the basis for our study. The structure model is depicted in Figure 4.1. The model shows exogenous variables on the left and right and endogenous variables at the centre. Exogenous variables depicted in a ellipse with a dashed outline constitute the hypothesised moderating variables.

All hypotheses investigated in our study are of the form: *"<MTL Property> is (positively or negatively) influenced by <MTL Capability>"*. They are represented by arrows from exogenous variables on the left of Figure 4.1 to endogenous variable at the centre. A moderation on the hypothesised influence is assumed from all exogenous variables on the right of the figure connected to the considered endogenous variable. In total we investigate 31 hypothesised influences, i.e. the number of outgoing arrows from the exogenous variables on the left of Figure 4.1.

Our study is guided by the following research questions:

RQ1 Which of the hypothesised interdependencies withstands a test of significance?

RQ2 How strong are the influences of model transformation language capabilities on the properties thereof?

RQ3 How strong are moderation effects expressed by the contextual factors *use-case*, *skills & experience* and *MTL choice*?

RQ4 What additional interdependencies arise from the analysis that were not initially hypothesised?

As the first study on this subject it contains confirmatory and exploratory elements. We intend to confirm which of the interdependencies between *MTL capabilities*, *MTL properties* and *contextual properties* withstand quantitative scrutiny (**RQ1**). We explore how strong the influence and moderation effects between variables are (**RQ2 & RQ3**), to gain new insights and to confirm their significance and relevance (minor influence strengths might suggest irrelevance even if goodness of fit tests confirm a correlation that is not purely accidental). Lastly, we utilise the exploratory elements of USM to identify interdependencies not hypothesised by the experts in our interviews (**RQ4**).

We use an *online survey* to gather data on language use and perceived quality of researchers and practitioners. The responses are analysed using universal structure modelling (USM) (Buckler et al. 2008) based on the structure model developed from the interview responses. This results in a quantified structure model with influence weights, significance values and effect strengths.

Based on the responses from 113 participants, the key contributions of this paper are:

- An adjusted structure model with newly discovered interdependencies;
- Quantitative data on the influence weight and effect strength of all factors as well as significant values for the influences;
- Quantitative data on the moderation strength of context factors;
- An analysis of the implications of the results for further empirical studies and language development;
- Reflections on the use of USM for investigating large hypotheses systems in software engineering research;

The method used in the reported study has been reviewed and published as part of the Registered Reports track at ESEM'22 (Höppner et al. 2022b).

The structure of this paper is as follows: Section 4.2 provides an extensive overview of model-driven engineering, domain-specific languages, model transformation languages and structural equation modelling as well as universal structure modelling. Afterwards, in Section 4.3 the methodology

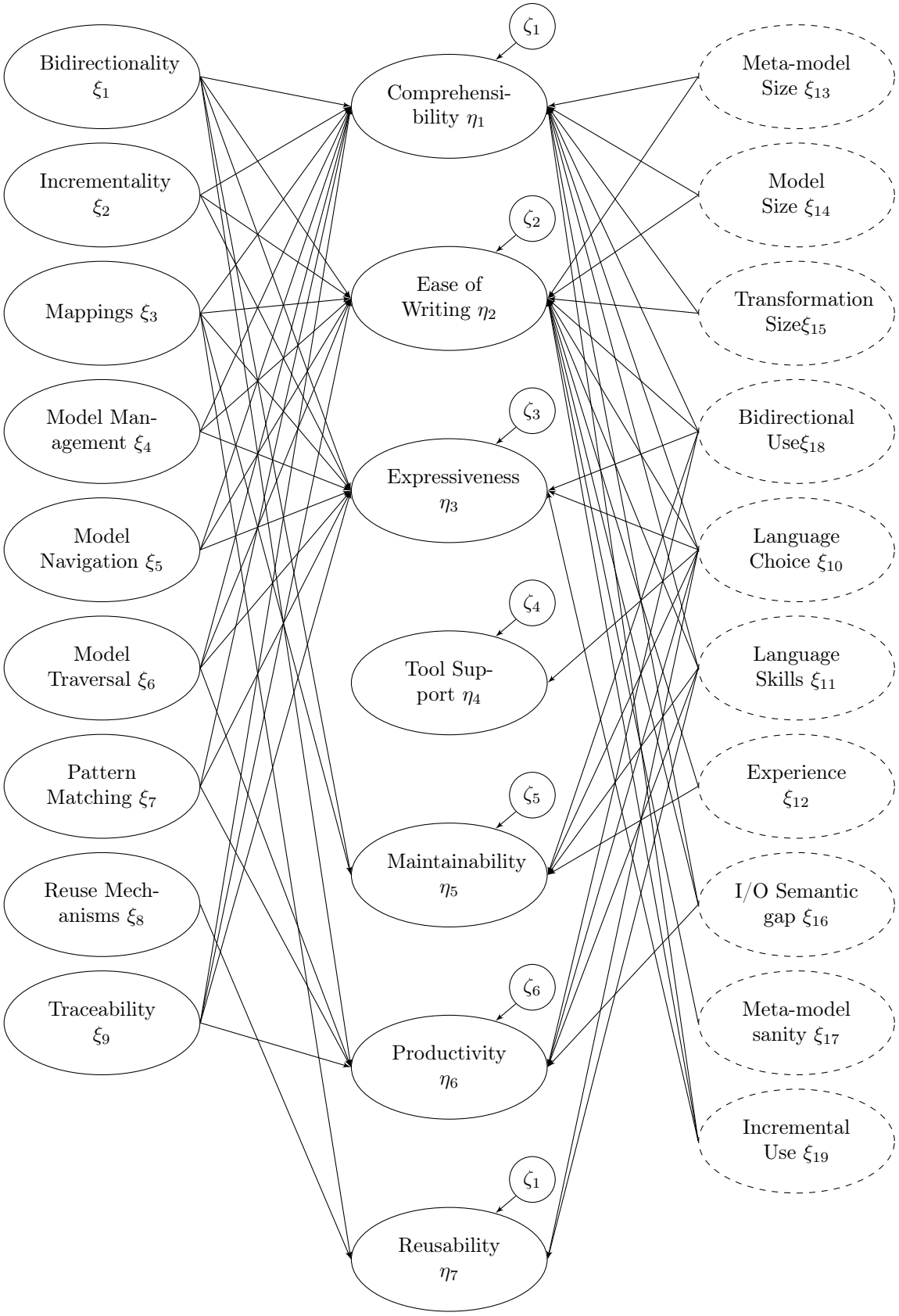


FIGURE 4.1: Structure model depicting the hypothesised influence and moderation effects of factors on MTL properties.

is outlined. Demographic data of the responses is reported in Section 4.4 and the results of analysis is presented in Section 4.5. In Section 4.6 we discuss implications of the results and report our reflections on the use of USM. Section 4.7 discusses threats to validity of our study and how we met them. Lastly, in Section 4.8 we present related work before giving concluding remarks on our study in Section 4.9.

4.2 Background

In this section we provide the necessary background for our study. Since it is a follow up study to our interview study (Höppner et al. 2022a) much of the background is the same and is therefore taken from those descriptions. To stay self contained we still provide these descriptions. This concerns Sections 4.2.1 to 4.2.3. Sections 4.2.4 and 4.2.5 contains an extension of our descriptions from the registered report (Höppner et al. 2022b).

4.2.1 Model-driven engineering

The *Model-Driven Architecture* (MDA) paradigm was first introduced by the Object Management Group in 2001 (OMG 2001). It forms the basis for an approach commonly referred to as *Model-driven development* (MDD) (Brown et al. 2005), introduced as means to cope with the ever growing complexity associated with software development. At the core of it lies the notion of using models as the central artefact for development. In essence this means, that models are used both to describe and reason about the problem domain as well as to develop solutions (Brown et al. 2005). An advantage ascribed to this approach that arises from the use of models in this way, is that they can be expressed with concepts closer to the related domain than when using regular programming languages (Selic 2003).

When fully utilized, MDD envisions automatic generation of executable solutions specialized from abstract models (Schmidt 2006; Selic 2003). To be able to achieve this, the structure of models needs to be known. This is achieved through so called meta-models which define the structure of models. The structure of meta-models themselves is then defined through meta-models of their own. For this setup, the OMG developed a modelling standard called *Meta-object Facility* (MOF) (OMG 2002) on the basis of which a number of modelling frameworks such as the *Eclipse Modelling Framework* (EMF) (Steinberg et al. 2008) and the *.NET Modelling Framework* (Hinkel 2016) have been developed.

4.2.2 Domain-specific languages

Domain-specific languages (DSLs) are languages designed with a notation that is tailored for a specific domain by focusing on relevant features of the domain (Van Deursen et al. 2002). In doing so DSLs aim to provide domain specific language constructs, that let developers feel like working directly with domain concepts thus increasing speed and ease of development (Sprinkle et al. 2009). Because of these potential advantages, a well defined DSL can provide a promising alternative to using general purpose tools for solving problems in a specific domain. Examples of this include languages such as *shell scripts* in Unix operating systems (Kernighan et al. 1984), *HTML* (Raggett et al. 1999) for designing web pages or AADL an architecture design language (SAEMobilus 2004).

4.2.3 Model transformation languages

The process of (automatically) transforming one model into another model of the same or different meta-model is called *model transformation* (MT). They are regarded as being at the heart of Model Driven Software Development (Metzger 2005; Sendall et al. 2003), thus making the process of developing them an integral part of MDD. Since the introduction of MDE at the beginning of the century, a plethora of domain specific languages for developing model transformations, so called model transformation languages (MTLs), have been developed (Arendt et al. 2010; Balogh et al. 2006; George et al. 2012; Hinkel et al. 2019a; Horn 2013; Jouault et al. 2006; Kolovos et al. 2008). Model transformation languages are DSLs designed to support developers in writing model transformations. For this purpose, they provide explicit language constructs for tasks involved in model transformations such as model matching. There are various features, such as directionality or rule organization (Czarnecki et al. 2006), by which model transformation languages can be distinguished.

Table 4.1: MTL feature overview

Feature	Characteristic	Representative Language
Embeddedness	Internal	FunnyQT (Clojure), RubyTL (Ruby), NMF Synchronizations (C#)
	External	ATL, Henshin, QVT
Rules	Explicit Syntax Construct	ATL, Henshin, QVT
	Repurposed Syntax Construct	NMF Synchronizations (Classes), FunnyQT (Macros)
Location Determination	Automatic Traversal	ATL, QVT
	Pattern Matching	Henshin
Directionality	Unidirectional	ATL, QVT-O
	Bidirectional	QVT-R, NMF Synchronisations
Incrementality	Yes	NMF Synchronizations
	No	QVT-O
Tracing	Automatic	ATL, QVT
	Manual	NMF Synchronizations
Dedicated Model Navigation Syntax	Yes	ATL (OCL), QVT (OCL), Henshin (implicit in rules)
	No	NMF Synchronizations, FunnyQT, RubyTL

For the purpose of this paper, we will only be explaining those features that are relevant to our study and discussion in Sections 4.2.3.1 to 4.2.3.7. Table 4.1 provides an overview over the presented features.

Please refer to Czarnecki et al. (2006), Kahani et al. (2019), and Mens et al. (2006) for complete classification.

4.2.3.1 External and Internal transformation languages

Domain specific languages, and MTLs by extension, can be distinguished on whether they are embedded into another language, the so called host language, or whether they are fully independent languages that come with their own compiler or virtual machine.

Languages embedded in a host language are called *internal* languages. Prominent representatives among model transformation languages are *FunnyQT* (Horn 2013) a language embedded in Clojure, *NMF Synchronizations* and the *.NET transformation language* (Hinkel et al. 2019a) embedded in C#, and *RubyTL* (Jesús Sánchez Cuadrado et al. 2006) embedded in Ruby.

Fully independent languages are called *external* languages. Examples of external model transformation languages include one of the most widely known languages such as the *Atlas transformation language* (ATL) (Jouault et al. 2006), the graphical transformation language Henshin (Arendt et al. 2010) as well as a complete model transformation framework called VIATRA (Balogh et al. 2006).

4.2.3.2 Transformation Rules

Czarnecki et al. (2006) describe rules as being *"understood as a broad term that describes the smallest units of [a] transformation [definition]"*. Examples for transformation rules are the rules that make up transformation modules in ATL, but also functions, methods or procedures that implement a transformation from input elements to output elements.

The fundamental difference between model transformation languages and general-purpose languages that originates in this definition, lies in dedicated constructs that represent rules. The difference between a transformation rule and any other function, method or procedure is not clear cut when looking at GPLs. It can only be made based on the contents thereof. An example of this

```java
public void methodExample(Member m) {
  System.out.println(m.getFirstName());
}
public void methodExample2(Member m) {
  Male target = new Male();
  target.setFullName(m.getFirstName() + " Smith");
  REGISTRY.register(target);
}
```

List. 4.1: Example Java methods

```
rule Member2Male {
  from
    s : Member (not s.isFemale())
  to
    t : Male (
      fullName <- s.firstName + ' Smith'
    )
}

rule Member2Female {
  from
    s : Member (s.isFemale())
  to
    t : Female (
      fullName = s.firstName + ' Smith'
      partner = s.companion
    )
}
```

List. 4.2: Example ATL rules

can be seen in Listing 4.1, which contains exemplary Java methods. Without detailed inspection of the two methods it is not apparent which method does some form of transformation and which does not.

In a MTL on the other hand transformation rules tend to be dedicated constructs within the language that allow a definition of a *mapping* between input and output (elements). The example rules written in the model transformation language ATL in Listing 4.2 make this apparent. They define mappings between model elements of type `Member` and model elements of type `Male` as well as between `Member` and `Female` using *rules*, a dedicated language construct for defining transformation mappings. The transformation is a modified version of the well known Families2Persons transformation case (Anjorin et al. 2017).

4.2.3.3 Rule Application Control: Location Determination

Location determination describes the strategy that is applied for determining the elements within a model onto which a transformation rule should be applied (Czarnecki et al. 2006). Most model transformation languages such as ATL, Henshin, VIATRA or QVT (OMG 2016), rely on some form of *automatic traversal* strategy to determine where to apply rules.

We differentiate two forms of location determination, based on the kind of matching that takes place during traversal. There is the basic *automatic traversal* in languages such as ATL or QVT, where single elements are matched to which transformation rules are applied. The other form of location determination, used in languages like Henshin, is based on *pattern matching*, meaning a model- or graph-*pattern* is matched to which rules are applied. This does allow developers to define sub-graphs consisting of several model elements and references between them which are then manipulated by a rule.

The *automatic traversal* of ATL applied to the example from Listing 4.2 will result in the transformation engine automatically executing the `Member2Male` on all model elements of type

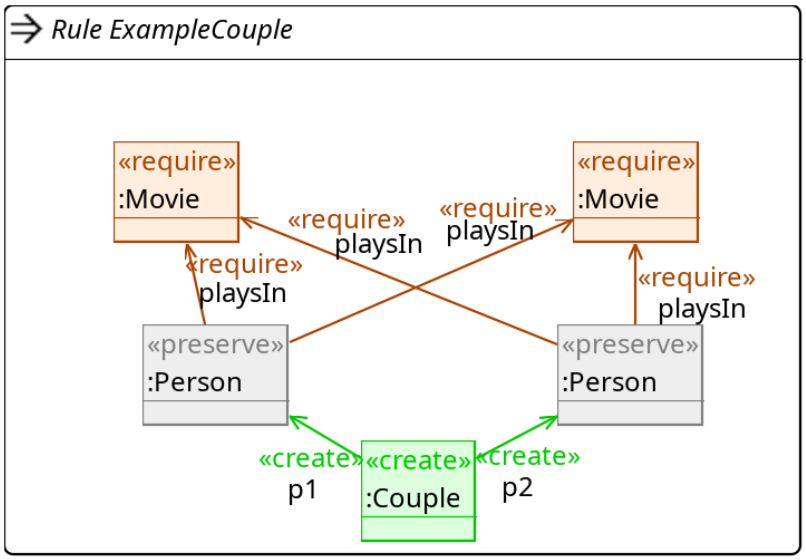


FIGURE 4.2: Example Henshin transformation

`Member` where the function `isFemale()` returns `false` and the `Member2Female` on all other model elements of type `Member`.

The *pattern matching* of Henshin can be demonstrated using Figure 4.2, a modified version of the transformation examples by Krause et al. (2014). It describes a transformation that creates a couple connection between two actors that play in two films together. When the transformation is executed the transformation engine will try and find instances of the defined graph pattern and apply the changes on the found matches.

This highlights the main difference between *automatic traversal* and *pattern matching* as the engine will search for a sub graph within the model instead of applying a rule to single elements within the model.

4.2.3.4 Directionality

The directionality of a model transformation describes whether it can be executed in one direction, called a unidirectional transformation or in multiple directions, called a multidirectional transformation (Czarnecki et al. 2006).

For the purpose of our study the distinction between unidirectional and bidirectional transformations is relevant. Some languages allow dedicated support for executing a transformation both ways based on only one transformation definition, while other require users to define transformation rules for both directions. General-purpose languages can not provide bidirectional support and also require both directions to be implemented explicitly.

The ATL transformation from Listing 4.2 defines a unidirectional transformation. Input and output are defined and the transformation can only be executed in that direction.

The QVT-R relation defined in Listing 4.3 is an example of a bidirectional transformation definition (For simplicity reasons the transformation omits the condition that males are only created from members that are not female). Instead of a declaration of input and output, it defines how two elements from different domains relate to one another. As a result given a `Member` element its corresponding `Male` elements can be inferred, and vice versa.

4.2.3.5 Incrementality

Incrementality of a transformation describes whether existing models can be updated based on changes in the source models without rerunning the complete transformation (Czarnecki et al. 2006). This feature is sometimes also called model synchronisation.

Providing incrementality for transformations requires active monitoring of input and/or output models as well as information which rules affect what parts of the models. When a change is detected the corresponding rules can then be executed. It can also require additional management tasks to be executed to keep models valid and consistent.

```
top relation Member2Male {
  n, fullName : String;
  domain Families s:Member {
    firstName = n };
  domain Persons t:Male {
    fullName = fullName};
  where {
    fullName = n + ' Smith'; };
}
```

List. 4.3: Example QVT-R relation

4.2.3.6 Tracing

According to Czarnecki et al. (2006) tracing *"is concerned with the mechanisms for recording different aspects of transformation execution, such as creating and maintaining trace links between source and target model elements"*.

Several model transformation languages, such as ATL and QVT have automated mechanisms for trace management. This means that traces are automatically created during runtime. Some of the trace information can be accessed through special syntax constructs while some of it is automatically resolved to provide seamless access to the target elements based on their sources.

An example of tracing in action can be seen in `line 16` of Listing 4.2. Here the `partner` attribute of a `Female` element that is being created, is assigned to `s.companion`. The `s.companion` reference points towards a element of type `Member` within the input model. When creating a `Female` or `Male` element from a `Member` element, the ATL engine will resolve this reference into the corresponding element, that was created from the referred `Member` element via either the `Member2Male` or `Member2Female` rule. ATL achieves this by automatically tracing which target model elements are created from which source model elements.

4.2.3.7 Dedicated Model Navigation Syntax

Languages or syntax constructs for navigating models is not part of any feature classification for model transformation languages. However, it was often discussed in our interviews and thus requires an explanation as to what interviewees refer to.

Languages such as OCL (OMG 2014), which is used in transformation languages like ATL, provide dedicated syntax for querying and navigating models. As such they provide syntactical constructs that aid users in navigation tasks. Different model transformation languages provide different syntax for this purpose. The aim is to provide specific syntax so users do not have to manually implement queries using loops or other general purpose constructs. OCL provides a functional approach for accumulating and querying data based on collections while Henshin uses graph patterns for expressing the relationship of sought-after model elements.

4.2.4 Structural equation modelling and (Universal) Structural Equation Modelling

Structural equation modelling (SEM) is an approach used for confirmatory factor analysis (Graziotin et al. 2021). It defines a set of methods used to *"investigate complex relationship structures between variables and allows for quantitative estimates of interdependencies thereof. Its goal is to map the a-priori formulated cause-effect relationships into a linear system of equations and to estimate the model parameters in such a way that the initial data, collected for the variables, are reproduced as well as possible"* (Weiber et al. 2021).

Structural equation modelling distinguishes between two sets of variables *manifest* and *latent*. *Manifest* variables are variables that are empirically measured and *latent* variables describe theoretical constructs that are hypothesised to interact with each other. Latent variables are further divided into *exogenous* or independent and endogenous or *dependent* variables.

So called structural equation models, a sample of which can be seen in Figure 4.3, comprised of manifest and latent variables, form the heart of analysis. They are made up of three connected

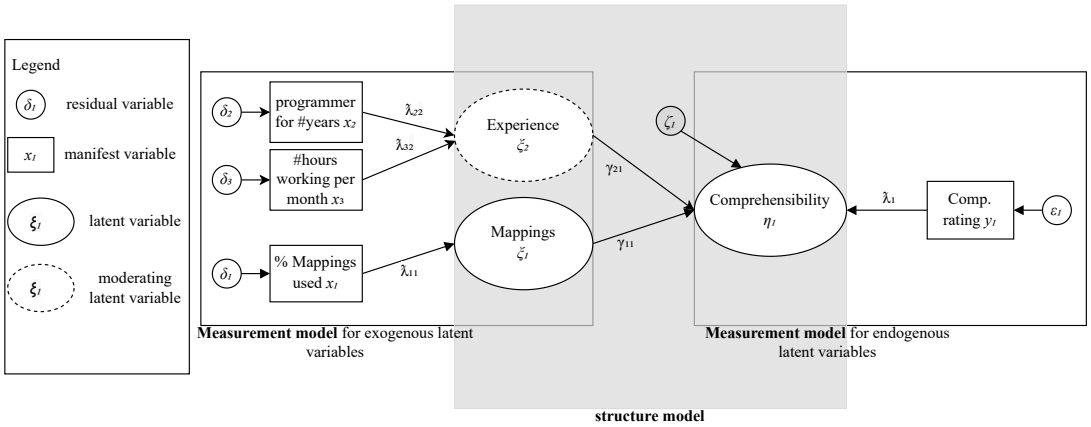


FIGURE 4.3: The makeup of a structural equation model.

sub-models. The *structure model*, the *measurement model* of the exogenous latent variables and the *measurement model* of the endogenous latent variables.

The *structure model* defines all hypothesised interactions between exogenous (ξ_{exID}) and endogenous (η_{endID}) latent variables. Each exogenous variable is linked, by arrow, to all endogenous variables that are presumed to be influenced by it. Each of these connections is given a variable (γ_{exID_endID}) that measures the influence strength. If an exogenous variable *moderates* the influences on a endogenous variable, the exogenous variable is depicted with a dashed outline and connected to all endogenous variables that are moderated by it[1]. For each moderated influence a separate variable of the form $\gamma_{exID\_endID\_modEndID}$ is assigned. In addition, an residual (or error) variable is appended to each endogenous latent variable to represent the influence of variables not represented in the model.

Figure 4.3 shows an example structure equation model model for the hypothesis that *"Mappings help with the comprehensibility of transformations, depending on the developers experience."*. The structure model seen at the centre of the figure, is comprised of the exogenous latent variable ξ_1 (*Mappings*), the moderating exogenous variable ξ_2 (*Experience*), the endogenous latent variable η_1 (*Comprehensibility*), a presumed influence of *Mappings* on *Comprehensibility* via γ_{11} and the error variable ζ_1. Lastly the model also contains a moderation of Experience on all influences of Comprehensibility. As described earlier, this moderation effect is assigned the variable γ_{11_2}. The moderation variables are not depicted in our graphical representation of the structure model because of their high number and associated visibility issues.

The *measurement model* of the exogenous latent variables reflects the relationships between all exogenous latent variables and their associated manifest variables. Each manifest variable is linked, by arrow, to all exogenous latent variables that are measured through it. Each of these connections is given a variable that measures the indication strength of the manifest variable for the latent variable. Additionally, an error variable for each manifest variable is introduced that represents measurement errors. In Figure 4.3, the measurement model for exogenous latent variables, seen at the left of the figure, is comprised of the exogenous latent variables ξ_1 (*Mappings*) and ξ_2 (*Experience*), the manifest variables x_1 (*% of code using Mappings*), x_2 (*number of years a person has been a programmer*) and x_3 (*number of hours per month spent developing transformations*) their measurement accuracy for Mapping usage λ_{11} and their measurement accuracy for Experience λ_{22} and and λ_{32} and the associated measurement error δ_1 and δ_2 and δ_3.

The *measurement model* of the endogenous latent variables reflects the relationships between all endogenous latent variables and their associated manifest variables. It is structured the same way as the *measurement model* of the exogenous latent variables. In Figure 4.3, it is shown on the right of the figure.

[1]To illustrate moderation, arrows are usually shown from the moderating exogenous variable to the arrow representing the moderated influence , i.e., an arrow between an exogenous variable and an endogenous variable. However our illustration deviates from this due to the size and makeup of our hypothesis system. Standard representations can be found in the basic literature such as Weiber et al. (2021).

Given a structural equation model and measurements for manifest variables, the SEM approach calls for estimating the influence weights and latent variables within the models. This is done in alternation for the measurement models and the structure model until a predefined quality criterion is reached. Traditional methods (covariance-based structural equation modeling & partial least squares) use different mathematical approaches such as maximum-likelihood estimation or least squares (Weiber et al. 2021) to estimate influence weights.

Universal Structure Modeling (USM) is an exploratory approach that complements the traditional confirmatory SEM methods (Buckler et al. 2008). It combines the iterative methodology of partial least squares with a Bayesian neural network approach using multilayer perceptron architecture. USM derives a starting value for latent variables in the model via principal component analysis and then applies the Bayesian neural network to discover an optimal system of linear, nonlinear and interactive paths between the variables. This enables USM to identify complex relationships that may not be detected using traditional SEM approaches including hidden structures within the data and highlights unproposed model paths, nonlinear relations among model variables, and moderation effects.

The primary measures calculated in USM are the 'Average Simulated Effect' (ASE), 'Overall Explained Absolute Deviation' (OEAD), 'interaction effect' (IE) and 'parameter significance'. ASE measures the average change in the endogenous variable resulting from a one-unit change in the exogenous variable across all simulations. OEAD assesses the degree of fit between the observed and simulated values of the endogenous variable, capturing the overall explanatory power of the model. IE evaluates the extent to which the effect of one exogenous variable on the endogenous variable depends on the level of another variable. Parameter Significance determines whether the estimated coefficients for each exogenous variable in the model are statistically significant at a predetermined level of confidence which indicated if the exogenous variable has a meaningful impact on the endogenous variable and is calculated through a bootstrapping routine (Mooney et al. 1993). These metrics together provide a comprehensive assessment of the performance and explanatory power of a USM model.

USM is recommended for use in situations where traditional SEM approaches may not be sufficient to fully explore the relationships between variables. Using USM instead of traditional structural equation modelling approaches is suggested for studies where there are still uncertainties about the completeness of the underlying hypotheses system and for exploring non-linearity in the influences (Buckler et al. 2008; Weiber et al. 2021). Moreover its use of a neural network also reduces the requirements for the scale levels of data thus allowing the introduction of categorical variables in addition to metric variables (Weiber et al. 2021).

At present, the tool NEUSREL[2] is the only tool available for conducting USM.

4.2.5 MTL Quality Properties

There exists a large body of quality properties that get associated with model transformation languages. In literature many claims are made about advantages or disadvantages of MTLs in these different properties. We categorised these properties in a previous work of ours (Götz et al. 2021a). This study focuses on a subset of all the identified quality properties of MTLs which requires them to be properly explained. In this section, we give a brief description of our definitions of each of the quality properties of MTLs relevant to the study.

Comprehensibility describes the ease of understanding the purpose and functionality of a transformation based on reading code.

Ease of Writing describes the ease at which a developer can produce a transformation for a specific purpose.

Expressiveness describes the amount of useful dedicated transformation concepts in a language.

Productivity describes the degree of effectiveness and efficiency with which transformations can be developed and used.

Maintainability describes the degree of effectiveness and efficiency with which a transformation can be modified.

Reusability describes the ease of reusing transformations or parts of transformations to create new transformations (with different purposes).

Tool Support describes the amount of quality tools that exist to support developers in their efforts.

[2] `https://www.neusrel.com`

4.3 Methodology

The methodology used in this study has been reviewed and published as part of the Registered Reports track at ESEM'22 (Höppner et al. 2022b). In the following, we provide a more detailed description and highlight all deviations from the reported method as well as justification for the changes.

The study itself is comprised of the following steps which were executed sequentially and are reported on in this section.

1. Development of survey methodology.
2. Submission to the Registered Reports track at EMSE'22.
3. Methodology revision based on feedback.
4. Development of online survey using an on premise version of the survey tool LimeSurvey[3].
5. Survey review and pilot test by co-authors.
6. Reworking survey based on pilot test.
7. Opening online survey to public.
8. Reaching out to potential survey subjects per mail and social media.
9. Closing of online survey (9 weeks after opening).
10. Data extraction.
11. Data analysis using the USM tool NEUSREL.

The steps executed differ in two ways from those reported in the registered report. First, we do not contact potential participants for a second time after two weeks. This was deemed unnecessary based on the number of participants at that point in time. Moreover we did not want to bother those that participated already and had no way of knowing their identity. Second, we kept the survey open 3 weeks longer than intended due to receiving several requests to do so.

4.3.1 Survey Design

In this section we detail the design of the used questionnaire and methodology used to develop and distribute it.

4.3.1.1 Questionnaire

The questions in the questionnaire are designed to query data for measuring the latent variables from the structure model in Figure 4.1. The complete questionnaire can be found in Appendix C.2. In the following, we describe each latent variable and explain how we measure it through questions in the questionnaire.

There are 26 latent variables relevant to our study. Variables $\xi_{1..19}$ describe exogenous variables and $\eta_{1..7}$ describe endogenous variables. Each latent variable is measured through one or more manifest variables. Extending the structure model from Figure 4.1 with the manifest variables produces the complete structural equation model evaluated in this study. Note that USM reduces the requirements for the scale levels of data thus allowing the use of categorical variables in addition to metric variables (Weiber et al. 2021).

All latent variables related to *MTL capabilities* ($\xi_{1..9}$) are associated with a single manifest variable $x_{1..9}$, which measures how frequently the participants utilized the MTL capabilities in their transformations. This measurement is represented as a ratio ranging from 0% to 100%. The higher the value of $x_{1..9}$, the more frequently the participants used the MTL capabilities in their transformations. Similarly, latent variables related to *MTL properties* ($\eta_{1..7}$) are associated with a single manifest variable $y_{1..7}$ which measures the perceived quality of the property on a 5-point likert scale (e.g., very good, good, neither good nor bad, bad, very bad).

[3] `https://www.limesurvey.org/`

The use of single-item scales is a debated topic. We justify their usage for the described latent variables on multiple grounds. First, the latent variables are of high complexity due to the abstract concepts they represent. Second, our study aims to produce first results that need to be investigated in more detail in follow up studies, more focused on single aspects of the model. And third, due to the size of our structural equation model multi-item scales for all latent variables would increase the size of the survey, potentially putting off many subjects. The validity of these deliberations for using single-item scales is supported by Fuchs et al. (2009).

The latent variable *language choice* (ξ_{10}) is measured by means of querying participants to list their 5 most recently used transformation languages. In our registered report we planned to also request participants to give an estimate on the percentage of their respective use % (x_{10}). This was discarded during pilot testing as it was seen as unnecessarily prolonging the questionnaire. Pilot testers had difficulties providing accurate data and questioned whether this data was actually used in analysis.

Language skills (ξ_{11}) is measured through x_{11} and x_{12} for which participants are asked to give the amount of years they have been using each language (x_{11}) and the amount of hours they use the language per month (x_{12}).

Similarly, *experience* (ξ_{12}) is associated with the amount of years subjects have been involved in defining model transformations (x_{13}) and the amount of hours they spend on developing transformations each month (x_{14}).

Meta-model size (ξ_{13}) and *model size* (ξ_{14}) both require participants to state the range between which their (meta-) models vary (x_{15}, x_{16}). This is measured by offering participants a number of ranges of (meta-) model objects. For each range participants should give an estimate on how much percent of the (meta-) models they work fall within that size range. For models the ranges are: #objects $\leq$ 10, 10 $\leq$ #objects $\leq$ 100, 100 $\leq$ #objects $\leq$ 1000, 1000 $\leq$ #objects $\leq$ 10000, 10000 $\leq$ #objects $\leq$ 100000, 100000 $\leq$ #objects. For meta-model the ranges are: #objects $\leq$ 10, 10 $\leq$ #objects $\leq$ 20, 20 $\leq$ #objects $\leq$ 50, 50 $\leq$ #objects $\leq$ 100, 100 $\leq$ #objects $\leq$ 1000, 1000 $\leq$ #objects. Similarly, *Transformation size* (ξ_{15}) is measured on a range of lines of code (x_{17}). The options being: LOC $\leq$ 100, 100 $\leq$ LOC $\leq$ 500, 500 $\leq$ LOC $\leq$ 1000, 1000 $\leq$ LOC $\leq$ 5000, 5000 $\leq$ LOC $\leq$ 10000, 10000 $\leq$ LOC. Querying size data in this manner and the associated ranges have been successfully applied in a prior work the authors were involved in (Groner et al. 2021).

To formulate the *semantic gap between input and output* (ξ_{16}) we elicit the similarity of the structure (x_{18}) and data types (x_{19}) on a 5-point likert scale (very similar, similar, neither similar nor dissimilar, dissimilar, very dissimilar). Participants are asked to give the percentage of all their meta-models that fall within each of the five assessments.

The *meta-model sanity* (ξ_{17}) is measured through means of how well participants perceive their structure (x_{20}) and their documentation (x_{21}) to be on a 5-point scale (very well, well, neither well nor bad, bad, very bad). Participants are asked to give the percentage of all their meta-models that fall within each of the five assessments.

Lastly, for both *bidirectional uses* (ξ_{18}) and *incremental uses* (ξ_{19}) we query participants on the ratio of bidirectional (x_{22}) and incremental (x_{23}) transformations compared to simple uni-directional transformations they have written.

4.3.1.2 Pilot Study

We pilot tested the study with three researchers from the institute. All pilot testers are researchers in the field of model driven engineering with more than 5 years of experience. Based on their feedback, we reworded some questions questions, removed the usage percentage part of the question for *language choice* and added more precise descriptions of the queried concepts. We then made the questionnaire publicly available and distributed a link to it via emails.

4.3.1.3 Target Subjects & Distribution

The target subjects are both researchers and professionals from industry that have used dedicated model transformation languages to develop model transformations in the last five years. We use voluntary and convenience sampling to select our study participants. Both authors reached out to researchers and professionals they knew personally via mail and request them to fill out the online survey. We further reach out, via mail, to all authors of publications listed in *ACM Digital Library*, *IEEE Xplore*, *Springer Link* and *Web of Science* that contain the key word *model transformation* from the last five years. A third source of subjects is drawn from social media. The authors use

their available social media channels to recruit further subjects by posting about the online-survey on the platforms. The social media platform used for distribution was MDE-Net[4], a community platform dedicated to model driven engineering.

The sampling method differs from the intended method by not including snowballing sampling as a secondary sampling method. We decided on this to have more control over the subjects receiving a link to the study as we believe secondary and tertiary contacts might be too far secluded from our target subjects.

Participation was voluntary and we did not incentivise participation through offering rewards. This decision is rooted in our experience in previous studies one other survey with 83 subjects (Groner et al. 2021) and the interview study we are basing this study on with 56 subjects (Höppner et al. 2022a).

It is suggested in literature to have between 5 to 10 times as many participants as the largest number of parameters to be estimated in each structural equation (i.e., the largest number of incoming paths for a latent model variable) (Buckler et al. 2008). Thus, the minimal number of subjects for our study to achieve stable results is 80. To gain any meaningful results a sample size of 30 must not be undercut (Buckler et al. 2008).

In total we contacted 2383 potential participants and got 113[5] responses exceeding the minimum requirement for stable results.

4.3.2 Data Analysis

We use USM to examine the hypotheses system modelled by the structure model shown in Figure 4.1. USM is chosen over its structural equation modelling alternatives due to it being able to better handle uncertainty about the completeness of the hypothesis system under investigation, it having more capabilities to analyse moderation effects and the ability to investigate non linear correlations (Weiber et al. 2021).

USM requires a declaration of an initial likelihood of an interdependence between two variables. This is used as a starting point for calculating influence weights but can change over the course of calculation. For this, Buckler et al. (2008) suggest to only assign a value of 0 to those relationships that are known to be wrong. We use the results of our interview study (Höppner et al. 2022a), shown in the structure model, to assign these values. For each path that is present in the model, we assume a likelihood of 100%. To check for interdependencies that might have been missed by interview participants, we also use a likelihood of 100% for all missing paths between $\xi_{1..19}$ and $\eta_{1..7}$. Our plan was to use a likelihood of 50% for these interdependencies but the tool available to us only allowed for either 100% or 0% to be put as input.

The tool NEUSREL is used on the extracted empirical data and the described additional input to estimate path weights and moderation weights within the extended structure model, i.e., the structure model where each exogenous latent variable is connected to all endogenous latent variables. It also runs significance tests via a bootstrapping routine (Buckler et al. 2008; Mooney et al. 1993) and produces the significance value estimates for each influence. The following procedures are then followed to answer the research questions from Section 4.1.

RQ1. We reject all hypothesised influences, i.e., those present in our structure model in Figure 4.1, that do not pass the statistical significance test. The threshold we set for this is 0.01. Moreover, we discard hypothesised influences with minimal effects strengths that are several magnitudes lower than the median influence of all coefficients. If, for example, the median of all path coefficients is 0.03 all influences with a coefficient lower or equal to 0.0009 are discarded. We do so because such low influences suggest that the influence is negligible.

RQ2 & RQ3. All path coefficients produced that were not rejected in **RQ1** will then provide direct values for the influence and moderation strengths to answer **RQ2** & **RQ3**.

RQ4. The same significance criteria we applied to all hypothesised influences for **RQ1**, we also apply to the extended influences, i.e., those not present in the structure model from Figure 4.1. Those influences that pass the significance test are added to the initial structural model as newly discovered influences.

[4] `https://mde-network.com/`

[5] This constitutes a response rate of 4.8%. We do however not know how many responses are a result of our social media posting.

4.3.3 Privacy and Ethical concerns

All participants were informed of the data collection procedure, handling of the data and their rights, prior to filling out the questionnaire. Participation was completely voluntary and not incentivised through rewards.

During selection of potential participants the following data was collected and processed.

- First & last name.
- E-Mail address.

The questionnaire did not collect any sensitive or identifiable data.

All data collected during the study was not shared with any person outside of the group of authors.

The complete information and consent form can be found in Appendix C.4.

The study design was not presented to an ethical board. The basis for this decision are the rules of the German Research Foundation (DFG) on when to use a ethical board in humanities and social sciences[6]. We refer to these guidelines because there are none specifically for software engineering research and humanities and social sciences are the closest related branch of science for our research.

4.4 Demographics

We detail the background and experience of the participants in our study in the following sections.

4.4.1 Experience in developing model transformations (ξ_{12})

Our survey captured model transformation developers with wide range of experience. The experience span (x_{13}) ranges from the least experience participant with half a year of experience up to the one with most experience of 30 years. Figure 4.4 shows a histogram of the experience stated by participants. Over half of all participants have between 1 to ten years of experience in writing model transformations. Three stated to have more than 20 years in total. On average our participants have 9 years of experience.

How much time participants spend developing transformations each month (x_{14}) also greatly varies. Some participants have not developed transformations in recent time whereas others stated to spend 70 or more hours each month on transformation development. Figure 4.5 shows an overview over the hours participants spend each month in developing transformations. The vast majority spends around 1 to 10 hours each month on transformation development. Nine stated that they did not develop any transformation in recent times. On average our participants spend about 14 hours per month developing model transformations.

4.4.2 Languages used for developing model transformations (ξ_{10}) and experience therein (ξ_{11})

To develop their transformations, participants use a wide array of languages. In total 43 languages (x_{10}) have been named 24 of which are unique languages used only by a single participant.

Surprisingly the language that has been used by the most participants is Java, a general purpose language. Java has been used by 70 of the 113 participants. The most used MTL is ATL with 58 users closely followed by another GPL, namely Xtend with 52 users. Table 4.2 shows how many participants use one of the ten most used languages for developing transformations.

Overall the prevalence of general purpose programming languages is higher than expected. This might be explained by the large number of existing MTLs which reduce the amount of total users per language while only four different GPLs are used.

[6] `https://www.dfg.de/foerderung/faq/geistes_sozialwissenschaften/`

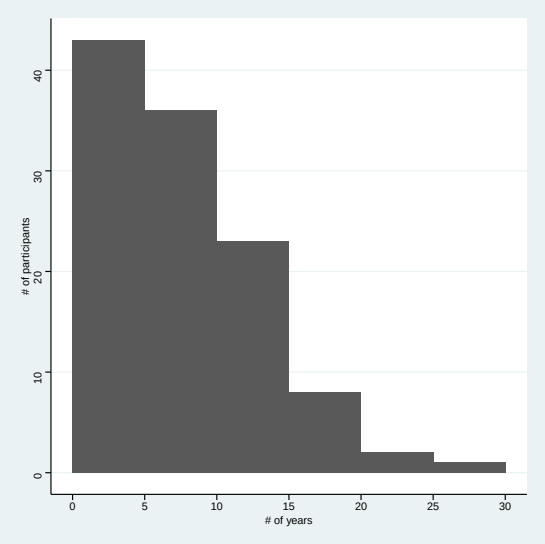


FIGURE 4.4: Histogram of participants total experience in years

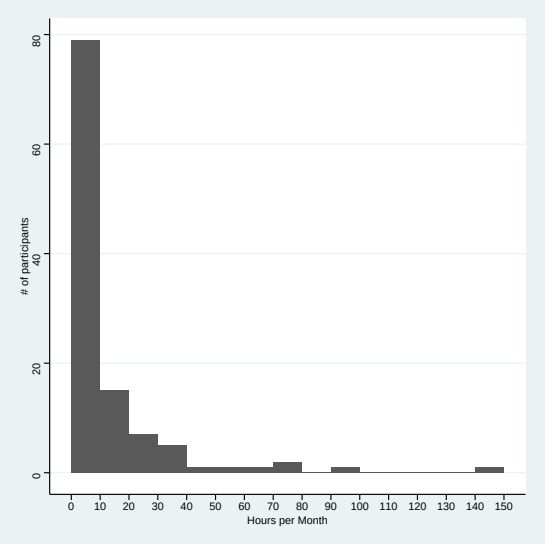


FIGURE 4.5: Histogram of participants recent experience in hours per month

TABLE 4.2: Overview of languages used by participants

Language	# number of participants
Java	70
ATL	58
Xtend	52
ETL	29
QVT-O	22
Henshin	14
JavaScript	12
eMoflon	7
Fujaba	5
Python	4

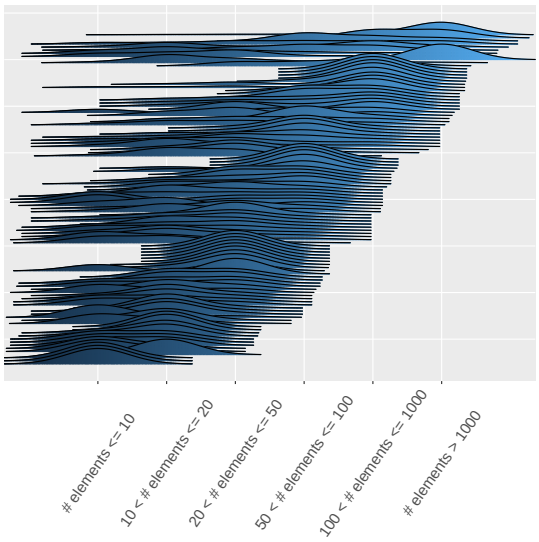


FIGURE 4.6: Distribution of meta-model sizes per participant

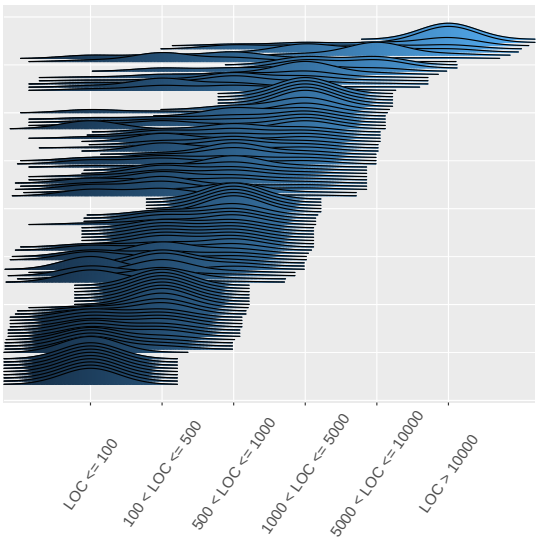


FIGURE 4.7: Distribution of transformation sizes per participant

4.4.3 Sizes (ξ_{12},ξ_{14})

The size distribution of meta-models (x_{15}) transformed by participants is shown in Figure 4.6. On the x-axis the given intervals of meta-model sizes are shown and on the y-axis the distribution for each participant is shown. For example, the first ridge line at the bottom of Figure 4.6 shows the answers of a participant who has stated that 100% of their transformations revolve around meta-models with 10 or less meta-model elements.

The figure illustrates that most transformations involve meta-models with 20 to 100 meta-model elements. Moreover, most participants have some experience with small meta-models while only a handful of them has experience with transformations involving large meta models of more than 1.000 elements.

The size distribution of model transformations (x_{17}) written by participants is shown in Figure 4.7. Similarly to the meta-model sizes, the figure illustrates that most participants have some experience with small transformations of sizes up to 100 lines of code. Most also have experience with large transformations up to 1.000 lines of code. More than 25% of all participants also have experience with large and very large transformations ranging from 5.000 up to more than 10.000 lines of transformation code.

Overall the experience of our participants includes many moderately large to large transformations. This strengthens us in the assumption that their answers are meaningful for our study.

4.4.4 Conceptual distance between meta-models (ξ_{16})

The similarity distribution of meta-models involved in the transformations of our participants is shown in Figure 4.8 for the similarity of meta-model structures (x_{18}) and Figure 4.9 for the similarity of data types (x_{19}). Both show a even mix between structurally similar and distant meta-models as well as similar and dissimilar attribute types within the elements that are transformed into each other.

4.4.5 Meta-model quality (ξ_{17})

Participants agreed that the vast majority of meta-models they transform are well structured (x_{20}). This means there is little to no additional burden put onto development solely due to unfavourably structured meta-models. The distribution of structure assessment per participant is shown in Figure 4.10.

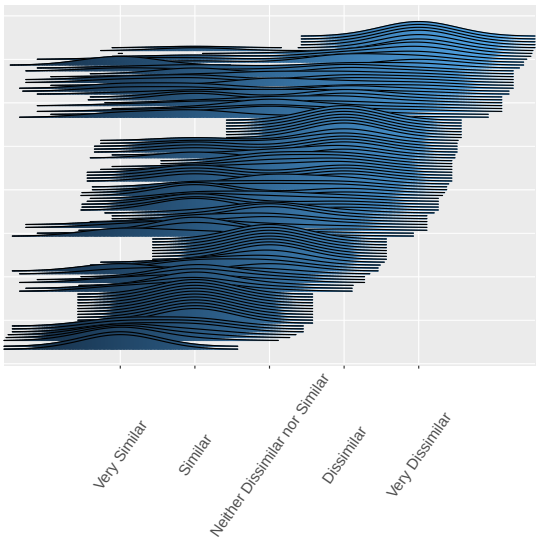


FIGURE 4.8: Distribution of input output meta-model structure similarity

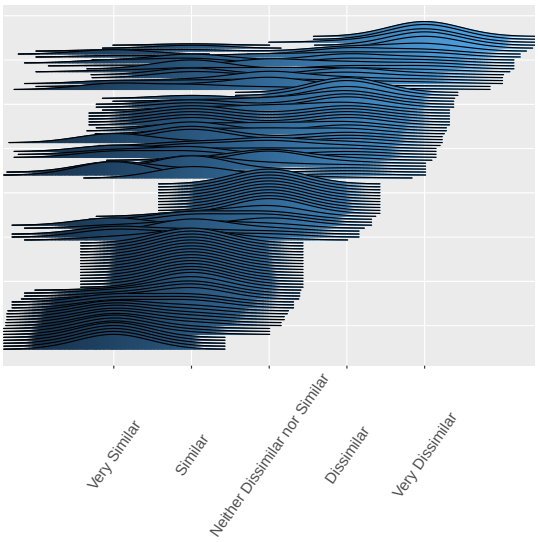


FIGURE 4.9: Distribution of input output meta-model attribute types similarity

The situation is different with documentation (x_{21}). Most participants stated that they have experience with badly or even very baldy documented meta-models Figure 4.11. For many participants, this constitutes the majority of meta-models they work with.

4.5 Results

In this section, we present the results of our analysis of the questionnaire responses using universal structure modelling structured around the research questions **RQ1-4**. The quantitative results for all influences between MTL capabilities and MTL properties are shown in Table 4.3 in Appendix C.1. On the x-axis the different *MTL Properties* are shown. On the y-axis the *MTL Capabilities* are shown. The first number in a cell describes the average simulated effect. The second number describes the overall explained absolute deviation. The third number shows the significance value. A significance value lower or equal to 0.01* (the chosen significance level) is indicated with one asterisks. The effect strengths of moderation effects can be found in Tables C.1 to C.10 in Appendix C.1. Each table describes the moderation effect of one of the moderating factors on all influences between MTL Capabilities and MTL Properties.

The rest of this section presents our results in context of the four research questions. We focus on the most salient influences that we deem interesting for the respective research question. Detailed interpretation and discussion of the implications of the presented results are done in Section 4.6.

4.5.1 RQ1: Which of the hypothesised interdependencies withstands a test of significance? & RQ4: What additional interdependencies arise from the analysis that were not initially hypothesised?

Our first research question is aimed at evaluating the accuracy of the structure model developed in the previous study (Höppner et al. 2022a). We do so by subjecting all hypothesised influences to a significance test during analysis. The significance test can also be used to directly gain insights into interdependencies missed in the initial model. Thus we discuss both the rejection of previously hypothesised influences as well as the extension of the model through newly discovered significant interdependencies in this section.

Most initially hypothesised influences withstand the test of significance but there are several exceptions. Most notably all but one(*Maintainability*) of the hypothesised influences of *Bidirectionality* functionality of MTLs have to be rejected. This means that from our results we can not conclude that the presence of *Bidirectionality* functionality in a language changes how people perceive the

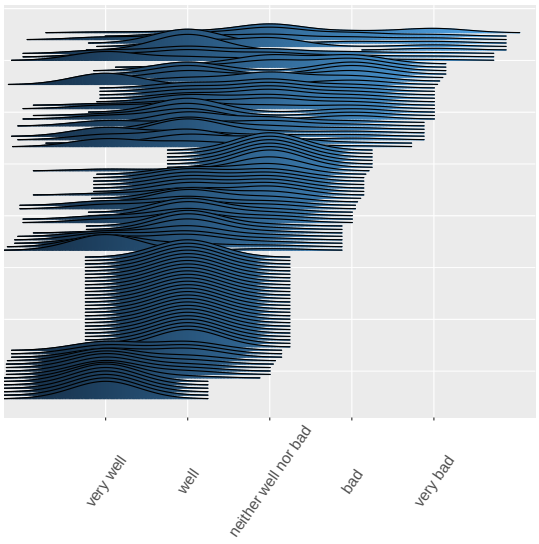


FIGURE 4.10: Distribution of structure quality of meta-models per participant

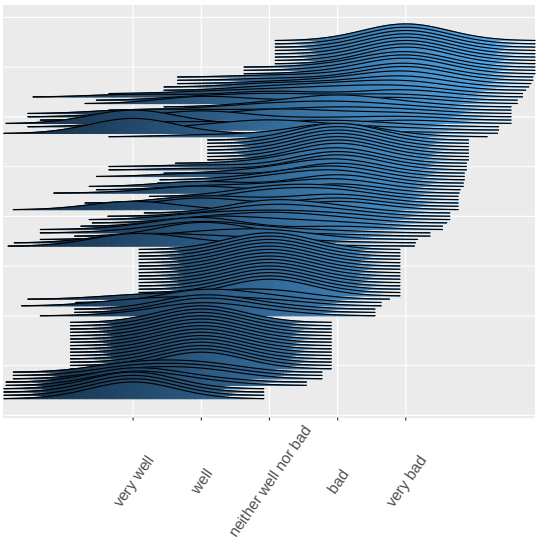


FIGURE 4.11: Distribution of meta-model documentation quality per participant

Comprehensibility, *Ease of Writing Expressiveness*, *Tool Support*, *Productivity* and *Reusability* of the language.

Similarly, half of the influences on *Ease of Writing* and *Expressiveness* are also rejected. This means that the presence of *Bidirectionality*, *Incrementality*, *Model Management* and *Model Traversal* functionality do not change how people perceive the *Ease of Writing* transformations with a language. And that the presence of *Bidirectionality*, *Incrementality*, *Model Navigation*, *Model Traversal* and *Reuse* functionality do not change how people perceive the *Expressiveness* of a language.

On a more positive note. The hypothesised influences on *Comprehensibility* are confirmed (apart from the one exerted by *Bidirectionality*). The same goes for *Productivity* and *Reusability*.

We also found that the perceived quality in *Tool Support* and *Maintainability* are influenced by most of the *MTL Properties*. A result that was not apparent from previous interview study. *Tool support* was hypothesised to be influenced only by the chosen language and *Maintainability* only by *Bidirectionality* and *Mapping* functionality. Our results however show, that the perceived *Maintainability* of transformations written in a language is influenced by all MTL functionality considered in this study with the exception of *Incrementality*.

Moreover, several additional influences on *Productivity* and *Reusability* were also discovered. The perceived *Productivity* and *Reusability* of transformations in a language are influenced by *Mappings*, *Model Management*, *Model Navigation*, *Model Traversal*, *Pattern Matching*, *Reuse Mechanisms* and *Tracability* functionality.

Regarding the moderating effects, our findings suggest that a nuanced view is warranted. The hypothesis that context moderates all influences on an MTL Property still holds but the strength of the moderation effects varies greatly.

As hypothesised, we are able to observe that *Comprehensibility* and *Ease of Writing* are the two properties moderated by the most context variables. But the moderation is only significant for a hand full of influences on these properties. This can be seen e.g. in the moderation effects of *Meta-Model Size* on influences on *Comprehensibility* depicted in Table C.2 in Appendix C.1. Changes in the *Meta-model sizes* participants worked with had next to no effect on how their usage of *Bidirectionality* functionality affected their view on the *Comprehensibility* of transformations. The impact on the influence of *Model Management* on *Comprehensibility* is orders of magnitudes higher.

Another observation that stands out is the impact of *Language Choice* and *Language Experience*. The moderation effects of both variables are negligible or even 0 for all influences. We believe this is due to the large number of languages considered in this study. It makes analysing the effects of choosing one of the languages difficult.

Overall the results for research questions **RQ1** & **RQ4** suggest that our initial structure model contains many relevant interdependencies but several more have to be considered as well. We do

have to reject several direct influences due to low significance and moderation effects have to be considered on a per influence basis instead of being generalised for each *MTL Property*.

4.5.2 RQ2: How strong are the influences of model transformation language capabilities on the properties thereof?

Our second research question is intended to provide numbers that can help to identify the most important factors to consider when evaluating the advantages and disadvantages of model transformation languages empirically. We do this by considering both the average simulated effect of influences calculated by NEUSREL as well as the overall explained absolute deviation of influences compared to each other. As explained earlier in this section all numbers can be found in Table 4.3.

Overall the effects identified in our analysis are lower than anticipated. They range from 0.29 down till 6.5e-8. We expected some effects to be low, mainly those from non significant interdependencies, but the fact that even significant effects are in the order of 0.01 is surprising. We assume this stems from the large number of variables that are involved and the overall complexity of the matter under investigation. Nonetheless we believe there are meaningful insights that can be drawn when comparing the influences for each *MTL Property* with each other.

Of the influences hypothesised from our previous interview study *Traceability* is the most impactful *MTL Capability*. Its usage exerts the highest influence on perceived *Comprehensibility* with 0.29. Similarly it has the highest influence for *Ease of Writing* though with a value of 0.0021 the effect is small. We were, however, already able to show empirical evidence that MTLs utilising automatic trace handling provide clear advantages for writing transformations compared to GPLs (Höppner et al. 2021).

For the properties *Tool Support*, *Maintainability* and *Productivity* the availability of *Reuse Mechanisms* seems to be the strongest driving factor with an average simulated effect of 0.1, 0.1, 0.1 and 0.2, respectively. No other factor has an ASE or effect strength as high as *Reuse Mechanisms* for these properties. This result is surprising as the influences were not raised even once during our interview study.

Overall, automatic tracing and reuse mechanisms appear to be the most influential factors for MTL properties. This suggests to us two main pathways for further research. First, to improve model transformation languages more research should be devoted to developing effective ways to reuse transformations or parts of transformations. From our experience, current mechanism are hard to use and are especially unsuited for different use-cases. Secondly, the first area to address for improved adoption of model transformation concepts in general purpose languages should be the development of mechanisms for automatic trace handling.

4.5.3 RQ3: How strong are moderation effects expressed by the contextual factors *use-case*, *skills & experience* and *MTL choice*?

As expressed in Section 4.5.1 the results of our analysis suggest that a more nuanced view of moderation effects is warranted. In this section we go into detail on these nuances.

As hypothesised the size of meta-models moderates the influences on *Comprehensibility*. The moderation strength differs greatly between the different causing factors though. For example, *Meta-model size* exerts the strongest moderation on the influence of *Model Management* onto *Comprehensibility* with 0.14. All other moderation effects are far lower. The second highest moderation effect, the moderation of *Meta-model size* on the influence of *Traceability* on *Comprehensibility*, is about half es strong (0.0778) and the lowest, the moderation of *Meta-model size* on the influence of *Bidirectionality* functionality on *Comprehensibility*, is only 0.0009. The moderations make sense intuitively as larger meta-models would make implementing these tasks manually more labour intensive and thus clutter the code unnecessarily.

Model size exerts similar moderation effects as meta-model size. Its strongest moderation effect is also on the influence of *Model Management* on *Comprehensibility* (0.36). Moreover, *Model size* also strongly moderates the influence of *Traceability* functionality on the *Ease of Writing* transformations (0.17). Most other moderation effects of *Model size* are far lower than 0.1.

Transformation size seems to be the most relevant moderating factors across the board. It has many noteworthy moderation effects on all influences of *MTL Capabilities* on *Tool Support*, none being less than 0.16, and *Productivity*, most being above 0.12. We assume this is because the larger

Table 4.3: Average simulated effect, overall explained absolute deviation and significance of direct influences

	Comprehensibility	Ease of Writing	Expressiveness	Tool Support	Maintainability	Productivity	Reusability
Bidirectionality	-3.2e-07/ 3.3e-07/ 1	6.6e-08/ 9.2e-05/ 1	1.5e-09/ 4.2e-06/ 1	-1.5e-05/ 2.0e-05/ 1	-0.06/ 0.05/ 0.1*	-8.3e-06/ 8.9e-07/ 1	1.4e-06/ 5.5e-06/ 1
Incrementality	0.02/ 0.01/ 0.01*	-3.5e-07/ 0.0003/ 1	7.4e-07/ 0.002/ 1	0.0002/ 0.0002/ 1	0.0004/ 1.7e-05/ 1	-0.0001/ 6.5e-07/ 1	0.02/ 0.005/ 0.05
Mappings	0.03/ 0.01/ 0.01*	-1.8e-05/ 0.01/ 0.01*	-6.8e-06/ 0.01/ 0.01*	-0.0116/ 0.01/ 0.01*	-0.000793/ 0.003/ 1	-0.005/ 0.04/ 0.01*	-0.02/ 0.01/ 0.01*
Model Management	0.03/ 0.1/ 0.01*	4.4e-05/ 0.003/ 1	0.0006/ 0.006/ 0.01*	0.02/ 0.07/ 0.01*	0.03/ 0.04/ 0.01*	-0.0005/ 0.05/ 0.01*	0.06/ 0.03/ 0.01*
Model Navigation	-0.01/ 0.03/ 0.01*	-2.3e-05/ 0.01/ 0.01*	5.9e-05/ 0.005/ 0.05	0.06/ 0.2/ 0.01*	-0.004/ 0.05/ 0.01*	0.05/ 0.07/ 0.01*	-0.08/ 0.08/ 0.01*
Model Traversal	0.008/ 0.009/ 0.01*	2.1e-07/ 0.002/ 1	-9.4e-05/ 0.0005/ 1	-0.09/ 0.2/ 0.01*	0.07/ 0.1/ 0.01*	-0.003/ 0.03/ 0.01*	0.007/ 0.01/ 0.01*
Pattern Matching	0.05/ 0.07/ 0.01*	-8.4e-05/ 0.01/ 0.01*	-0.0001/ 0.006/ 0.01*	0.05/ 0.06/ 0.01*	-0.04/ 0.08/ 0.01*	0.005/ 0.1/ 0.01*	-0.06/ 0.06/ 0.01*
Reuse Mechanisms	0.1/ 0.08/ 0.01*	-7.8e-05/ 0.008/ 0.01*	-0.0002/ 0.002/ 1	0.1/ 0.1/ 0.01*	0.1/ 0.2/ 0.01*	0.1/ 0.2/ 0.01*	0.2/ 0.1/ 0.01*
Traceability	0.29/ 0.12/ 0.01*	0.002/ 0.02/ 0.01*	-2.1e-05/ 0.007/ 0.01*	-0.05/ 0.2/ 0.01*	-0.02/ 0.1/ 0.01*	0.04/ 0.05/ 0.01*	0.08/ 0.09/ 0.01*

Please note that the significance values obtained through the NEUSREL tool may exhibit reduced accuracy compared to standard approaches due to the bootstrapping method used for their estimation.

transformations get, the more reliant developers are on tooling and abstractions that reduce the development effort.

Another interesting effect we found is, that *developer experience* moderates the influence of many of the domain specific abstractions, e.g. *Mappings* and *Model Traversal*, on *Productivity*. This makes sense because these specific features often break with how developers are used to develop programs and thus need practice to use them effectively.

The *semantic gap between input and output* meta-models exerts its moderation strongest on the influences on *Maintainability*. Most notable are the moderations on the influences of *Model Traversal* (0.194), *Pattern Matching* (0.239) and *Reuse Mechanisms* (0.237).

Lastly, there is a strong moderation effect of the *meta-model sanity* onto the influence of *Model Management* facilities and *Bidirectionality* on *Comprehensibility*. Both being about 0.2. This makes sense as badly structured or poorly documented meta-models are harder to handle and thus the tasks revolving around working with the structure are most influenced by that.

Overall the size of transformations is in our opinion the most relevant moderating variable. The assumption on the relevance of language choice could however not be confirmed. This is most likely due to the large amount of languages each participant has had experience with which weakens the ability to elicit the effect of differences of language choice between participants.

4.6 Discussion

The results of our analysis provide useful insights for research on model transformation languages. In this section, we discuss the implications of our results for evaluation and development of MTLs. Additionally, we provide a critical evaluation of our methodology with regards to the goals of this study.

4.6.1 Implications of results

The topic of influences on the quality properties of model transformation language is vastly complex, as reflected in the already large structure model which we set out to analyse. While we were able to reject some of the hypothesised influences, our analysis also identified several new influences. As a result, the structure model depicting the influences grew in complexity, further highlighting the need for comprehensive studies of the factors that influence MTL quality properties. The updated structure model can be seen in Figure 4.12. It contains 36 more interdependencies than the one we started our analysis with.

Our analysis produced a number of interesting observations that have important implications for further research. In particular, we now discuss the implications for empirical evaluations. Additionally, we highlight the implications of our results for further development of MTLs and domain-specific features thereof.

4.6.1.1 Suggestions for further empirical evaluation studies

Traceability is one of the most important factors to consider when it comes to the development of model transformations. This is because it has the strongest influence on the perceived quality of both the ease of writing and the comprehensibility of the resulting code. It is crucial to consider scenarios where tracing is involved in order to properly evaluate the value of MTL abstractions for writing and comprehending transformations. Additionally, it is important to evaluate scenarios where tracing is not necessary to understand the difference that MTL abstractions can make. To truly understand the relevance of this feature, it is also important to assess how many real-world use cases require it. By taking all of these factors into account, it is possible to gain a comprehensive understanding of the value of MTL abstractions for writing and comprehending transformations.

For evaluation of Maintainability, *Reuse Mechanisms* as well as *Model Traversal* functionality are important capabilities to consider. We therefore believe that researchers focusing on such an evaluation must make sure to use transformations that utilise these capabilities. Moreover, the most important context to consider is the semantic gap between input and output meta-models. Empirical evaluations focusing on maintainability should therefore make sure to evaluate transformation cases with varying degrees of differences between input and output meta-models. These studies should then analyse how much the effectiveness of MTLs and GPLs changes in light of the semantic gap between input and output.

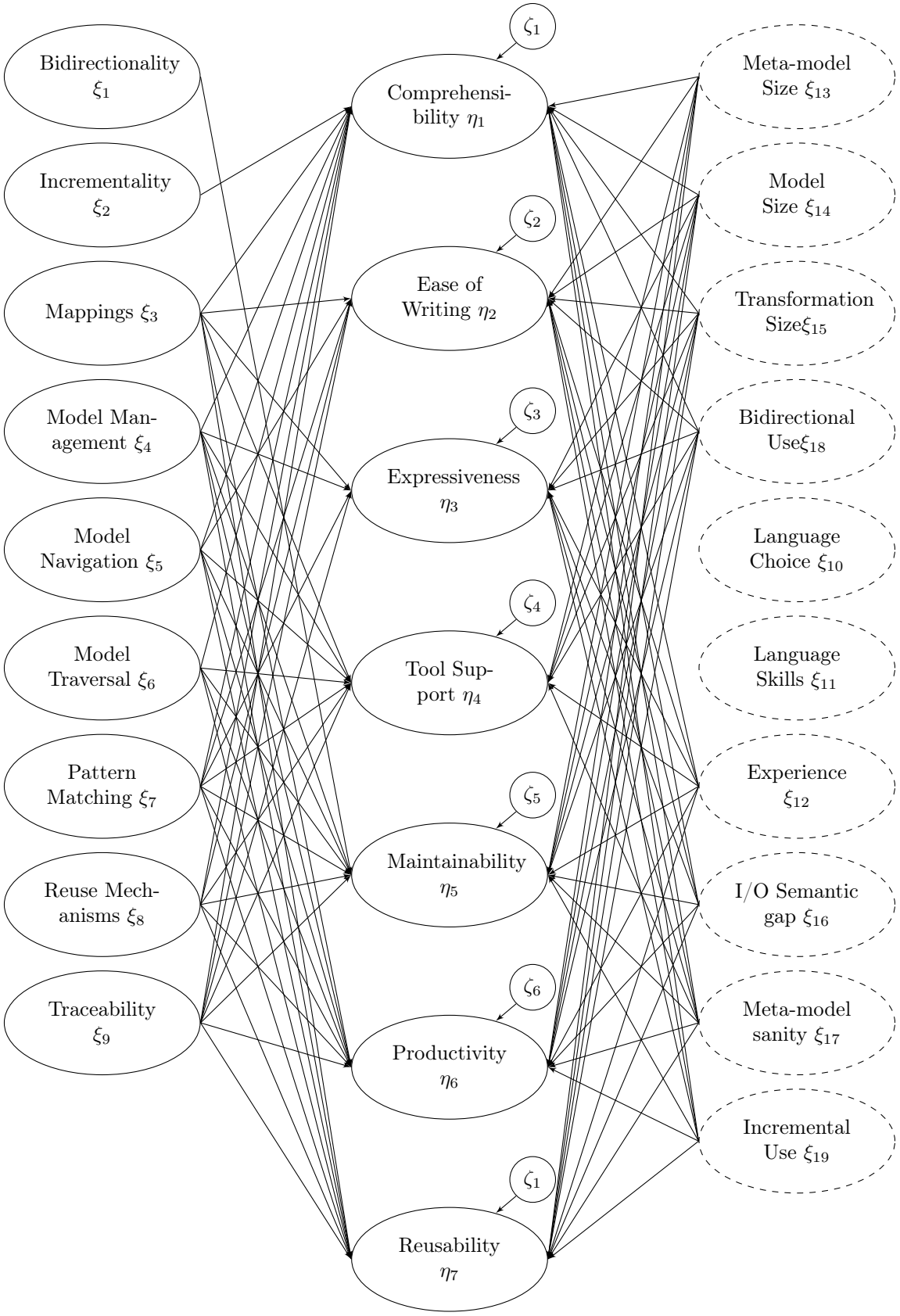


FIGURE 4.12: Structure model depicting the confirmed influence and moderation effects of factors on MTL properties.

When selecting transformations for evaluation, it is essential to consider their size. Our results have shown that size has the most significant impact on the influence of other factors on properties. Put differently, the larger the transformation, the more noticeable the effect of all capabilities will be. As such, it is imperative to focus on large transformation use-cases when designing a study to evaluate MTLs.

4.6.1.2 Suggestions on language development

For us, the most surprising finding of this study is the importance of reuse functionality. The quality attributes tool support, maintainability, productivity and reusability are all most influenced by it. This is especially surprising because there was no indication of this in our interviews (Höppner et al. 2022a). We suppose this influence stems from the fact that reuse mechanisms allow for more abstraction and thus less code that can be developed and maintained more efficiently.

As a result we believe that more focus should be put on developing transformation specific reuse mechanisms. We are aware that some languages, e.g. ATL, already provide general reuse mechanisms through concepts like inheritance. However, these concepts are limited by the fact that they rely on the object-oriented nature of the involved models. This means that they can only be used to define reusable code within transformations of a single meta-model. Defining transformation behaviour that can be reused between different meta-models is not possible. But this would be important to further reduce redundancy in transformation development.

As result, we believe, that development of reuse mechanisms tailored to MTs is important to focus on. In order to stand out compared to the reuse mechanisms of GPLs, it may be valuable to explore ways to define and reuse common transformation patterns independently of meta-models. Higher order transformations are sometimes used to allow reuse too (A. Kusel et al. 2015), but from our experience current implementations are too cumbersome to be used productively. Chechik et al. (2016) provide a number of suggestions for transformation specific reuse mechanisms but to the best of our knowledge there exist no implementations of their concepts.

4.6.2 Interesting observations outside of USM

When discussing model transformation languages, it is often stated that they are only demonstrated on 'toy examples' that have little to no real world value. This argumentation has for example been raised several times in our previous interview study (Höppner et al. 2022a). However, the demographic data collected in our study disputes this.

There are several participants that stated to have worked solely on small transformations with small meta-and input models. But this group is opposed by a similarly large group of participants that have worked with huge transformations, dissimilar and large meta-models as well as large inputs. From this we conclude, that there are large use-cases where model transformations and MTLs are applied but they rarely get described in publications. It seems likely that such examples are not used for highlighting important aspects authors want to discuss due to the space describing such cases would take up. However, we argue that it is paramount that such case-studies are published to diminish the cynicism that MTLs are only useful for small examples.

Another noteworthy observation based on the demographic data of our participants is that documentation pertaining meta-models is predominantly perceived as inadequate. We believe that this is primarily due to the fact that many of meta-models stem from research projects that prioritize expeditious prototyping over the long-term viability of the artefacts. Nonetheless, we are convinced that there is an urgent need to enhance the documentation surrounding model transformations. This issue is not limited solely to the meta-models, but also extends to the languages that are known for their challenging learning curve because of lack of tutorials (Höppner et al. 2022a).

4.6.3 Critical Assessment of the used methodology

The appeal of using structural equation modelling for analysing the responses to our survey was to have a method of analysis that can be used to investigate a complex hypothesis system in its entirety. Moreover, analysis is straight forward after an initial setup due to the sophisticated tooling for this methodology. Instead of presenting participants with a case that they should assess we also opted for querying them on their overall assessment of MTL quality attributes. These design decisions have implications and ramifications that we discuss in this section.

First, the effects observed in our study are small. We assume this stems from the intricate and large structure model and the comparatively small sample size. As explained in Section 4.3 it is suggested to have between 5 to 10 times as many participants as the largest number of parameters to be estimated in each structural equation. In light of the newly discovered paths in our structure model, the 113 total participants are close to the minimum sample size required. Moreover, because of the large number of influences we do expect the influence of a single factor to be much smaller than in structure models where only 2-3 factors are relevant. The results therefore reinforce our assessment that it is a very complex topic.

We also ran into some difficulties when using NEUSREL to analyse our data. The structure model was so large that sometimes the tool crashed during calculations. The online tooling to set everything up was also painfully inefficient leading to more problems during setup like browser crashes. It took us some trial and error to find a way to get everything set up and run the analysis without crashes.

We chose to execute a study based on our study design in hopes of producing a complete theory independent of the use case under consideration. The results exhibit less effect strength but we believe them to be more externally valid. Nonetheless, we think that several additional studies need to be conducted to confirm our results for different use-cases.

4.7 Threats to validity

Our study is carefully designed and follows standard procedures for this type of study. There are, however still threats to validity that stem from design decisions and limitations. In this section we discuss these threats.

4.7.1 Internal Validity

Internal validity is threatened by manual errors and biases of the involved researchers throughout the process.

The two activities where such errors and biases can be introduced are the subject selection and question creation. The selection criteria for study subjects is designed in such a way, that no ambiguities exist during selection. This prevents researcher bias.

The survey questions and answers to the questions pose another threat to internal validity. We used neutral questions to prevent subconsciously influencing the opinions of research subjects. We also provide explanations for ambiguous terms used in the survey. However, there are several instances where we can not fully ensure that each participant interprets terms the same way. The questions on quality properties of model transformation languages allow room for interpretation in that we do not provide a clear metric what terms such as ‘Very Comprehensible’ or ‘Very Hard to write’ mean. Similarly, the questions on meta-model quality leave room for interpretation on the side of participants. We opted for this limitation because there are no universal ways to quantify such estimates and because the subjective assessment is what we want to collect. The reason for this is, that subjective experiences are the main driving factor for all discussions on development when people are the main subject.

To ensure overall understandability and prevent errors in the setup of the survey we used a pilot study.

4.7.2 External Validity

External validity is threatened by our subject sampling strategy and the limitations on the survey questions imposed by the complexity of the subject matter.

We utilise convenience sampling. Convenience sampling can limit how representative the final group of interviewees is. Since we do not know the target populations makeup, it is difficult to asses the extend of this problem.

Using research articles as a starting point introduces a bias towards researchers. There is little potential to mitigate this problem during the study design, because there exists no systematic way to find industry users.

Due to the complexity and abstractness of the concepts under investigation, a measurement via reflective of formative indicators is not possible. Instead we use single item questions. We further assume that positive and negative effects of a feature are more prominent if the feature is used more

frequently. This can have a negative effect on the external validity of our results. However, we consciously decided for these limitations to be able to create a study that concerns itself with all factors and influences at once.

4.7.3 Construct Validity

Construct validity is threatened by inappropriate methods used for the study.

Using the results of online surveys as input for structural equation modelling techniques is common practice in market research (Weiber et al. 2021). It is less common in computer science. However, we argue that for the purpose of our study it is an appropriate methodology. This is because the goal of extracting influence strengths and moderation effects of factors on different properties aligns with the goals of market research studies that employ structural equation modelling.

4.7.4 Conclusion Validity

Conclusion validity is mainly threatened by biases of our survey participants.

It is possible that people who do research on model transformation languages or use them for a long time are more likely to see them in a positive light. As such there is the risk that too little experiences will be reported on in our survey. However, this problem did not present itself in a previous study by us on the subject matter (Höppner et al. 2022a). In fact researchers were far more critical in dealing with the subject. As a result, there might be a slight positive bias in the survey responses, but we believe this to be negligible.

4.8 Related Work

There are numerous works that explore the possibilities gained through the usage of MTLs such as automatic parallelisation (Benelallam et al. 2015; Biermann et al. 2010; Sanchez Cuadrado et al. 2020), verification (Ko et al. 2015; Lano et al. 2015) or simply the application of difficult transformations (Anastasakis et al. 2007). There is, however, only a small amount of works trying to evaluate the languages to gain insights into where specific advantages or disadvantages associated with the use of MTLs originate from. Several other works that can be related to our study also exist. The related work is divided into studies focused on the investigation of properties of model transformation languages and empirical studies on model transformation languages.

4.8.1 Studies on the Properties of Model Transformation Languages

Burgueño et al. (2019) conducted on a online survey and open discussion at the 12th edition of the International Conference on Model Transformations (ICMT'2019). The goal of the survey was to identify reasons why developers decided to use or dismiss MTLs for writing transformations. They also tried to gauge the communities sentiment on the future of model transformation languages. At ICMT'2019, where the results of the survey were presented, they then held an open discussion on this topic and collected the responses of participants. Their results show that MTLs have fallen in popularity. They attribute this to 3 types of issues, technical issues, tooling issues and social issues, as well as the fact that GPLs have assimilated many ideas from MTLs. The results of their study are a major driver in the motivation of our work. While they identified issues and potential avenues for future research, their results are qualitative and broad which we try to improve upon with our study.

In a prior study of ours (Götz et al. 2021a), we conducted a structured literature review which forms the basis of much of our work since then. The literature review aimed at extracting and categorising claims about the advantages and disadvantages of model transformation languages as well as the state of empirical evaluation thereof. We searched over 4000 publication for this purpose and extracted 58 that directly claim properties of MTLs. In total 137 claims were found and categorised into 15 quality properties of model transformation languages. The results of the study show that little to no empirical studies to evaluate MTLs exist and that there is a severe lack of context and background information that further hinders their evaluation.

Lastly, there is our interview study (Höppner et al. 2022a) the data of which forms the basis for the reported study. We interviewed 56 people on what they believe the most relevant factors are that facilitate or hamper their advantages for different quality properties identified in the prior

literature review. The interviews brought forth insights into factors from which the advantages and disadvantages of MTLs originate from as well as suggested a number of moderation effects on the effects of these factors. These results for the data basis for this study.

4.8.2 Empirical Studies on Model Transformation Languages

Hebig et al. (2018) report on a controlled experiment to evaluate how the use of different languages, namely ATL, QVT-O and Xtend affects the outcome of students solving several transformation tasks. During the study student participants had to complete a series of three model transformation tasks. One task was focused on comprehension, one task focused on modifying an existing transformation and one task required participants to develop a transformation from scratch. The authors compared how the use of ATL, QVT-O and Xtend affected the outcome of each of the tasks. Unfortunately their results show no clear evidence of an advantage when using a model transformation language compared to Xtend. However, they concede that the conditions under which the observations are made, were narrow.

We published a study on how much complexity stems from what parts of ATL transformations (Götz et al. 2020) and compared these results with data for transformations written in Java (Höppner et al. 2021) to elicit advantageous features in ATL and to explore what use-cases justify the use of a general purpose language over a model transformation language. In the study, the complexity of transformations written in ATL were compared to the same transformations written in Java SE5 and Java SE14 allowing for a comparison and historical perspective. The Java transformations were translated from the ATL transformations using a predefined translation schema. The results show that new language features in Java, like the Streams API, allow for significant improvement over older Java code, the relative amount of complexity aspects that ATL can hide stays the same between the two versions.

Gerpheide et al. (2016) use a mixed method study consisting of expert interviews, a literature review and introspection, to formalize a quality model for the QVT-O model transformation standard. The quality model is validated using a survey and used to identify the necessity of quality tool support for developers.

We know of two study templates for evaluating model transformation languages that have been proposed but not yet used. Kramer et al. (2016) propose a template for a controlled experiment to evaluate comprehensibility of MTLs. The template envisages using a questionnaire to evaluate the ability of participants to understand what presented transformation code does. The influence of the language used for the transformation should then be measured by comparing the average number of correct answers and average time spent to fill out the questionnaire. Strüber et al. (2016) also propose a template for a controlled experiment. The aim of the study is to evaluate the benefits and drawbacks of *rule refinement* and *variability-based rules* for reuse. The quality of reusability is measured through measuring the comprehensibility as well as the changeability collected in bug-fixing and modification tasks.

4.9 Conclusion

Our study provides the first quantification of the importance of model transformation language capabilities for the perception of quality attributes by developers. It once again highlight the complexity of the subject matter as the effect sizes of the influences are small and the final structure model grew in size.

As demonstrated by the amount of influences contained in the structure model many language capabilities need to be considered when designing empirical studies on MTLs. The results however point towards Traceability and Reuse Mechanisms as the two most important MTL capabilities. Moreover, the size of the transformations provides the strongest moderation effects to many of influences and is thus the most important context factor to consider.

Apart from implications for further empirical studies our results also point a clear picture for further language development. Transformation specific reuse mechanisms should be the main focus as shown by their relevance for many development lifecycle focused quality attributes such as Maintainability and Productivity.

Chapter 5

Paper D

Investigating the Origins of Complexity and Expressiveness in ATL Transformations

S. Götz, M. Tichy

Journal of Object Technology (JoT), volume 19, article number 2, July 2020
AITO — Association Internationale pour les Technologies Objets

Abstract

Model transformations provide an essential element to the model driven engineering approach. Over the years, many languages tailored to this special task, so-called model transformation languages, have been developed. A multitude of advantages have been proclaimed as reasons to why these dedicated languages are better suited to the task of transforming models than general purpose programming languages. However, little work has been done to confirm many of these claims. In this paper, we analyse ATL transformation scripts from various sources to investigate three common claims about the expressiveness of model transformation languages. The claims we are interested in assert that automatic trace handling and implicit rule ordering are huge advantages for model transformation languages and that model transformation languages are able to hide complex semantics behind simple syntax. We use complexity measures to analyse the distribution of complexity over transformation modules and to gain insights about what this means for the abstractions used by ATL. We found that a large portion of the complexity of transformations stem from simple attribute assignments. We also found indications for the usefulness of conditioning on types, implicit rule ordering and automatic trace resolution.

5.1 Introduction

Model transformations are a pivotal part of model-driven engineering (MDE) (Schmidt 2006; Sendall et al. 2003). This is also evident from the amount of transformation languages that have been proposed, i.e. ATL (Jouault et al. 2006), Henshin (Arendt et al. 2010), ETL (Kolovos et al. 2008), Viatra (Balogh et al. 2006) and QVT (Kurtev 2007) just to name a few.

While the number of transformation languages and their features is ever increasing, little time is spent on empirical studies on the use of said languages (Selim et al. 2017). A fact that is true not only for model transformation languages but any kind of DSL as evident from the results of Tomaž Kosar et al. (2016).

Selim et al. (2017), have shown that studying the use of transformation languages on code repositories such as the ATL Zoo[1] can provide insights into how a transformation language is used which can help developers with language evolution.

Such studies are also necessary because there is continual debate about whether dedicated model transformation languages are necessary at all (Burgueño et al. 2019; Hebig et al. 2018) since GPLs like Java can also be used for writing transformations and have been discussed as an alternative since the introduction of model transformations (Sendall et al. 2003).

In the study described here, we apply these goals to the Atlas model Transformation Language (ATL) (Jouault et al. 2006). We are particularly interested in investigating transformation scripts to gather data concerning the following claims which have been made multiple times in literature:

***H1*:** *Model transformation languages hide complex semantics behind simple syntax (Gray et al. 2003; Jouault et al. 2008; Krikava et al. 2014; Sendall et al. 2003).*

***H2*:** *Automatic handling and resolution of trace information by the transformation engines is a huge advantage of model transformation languages (Hinkel et al. 2019b; Jouault et al. 2008; Lawley et al. 2007).*

***H3*:** *Model transformation languages allow for implicit rule ordering which can lessen the load on developers (Jouault et al. 2008; Lawley et al. 2007).*

One thing that immediately stands out from the three claims is that they are intertwined. Automatic handling of traces and implicit rule ordering are both concepts that can hide certain semantics within the transformation engine. So to investigate their impact and provide insights into the complexity within model transformations as a whole we devised 5 research questions to focus our research on:

RQ1: *How is the complexity of ATL transformations distributed over multiple transformations and transformation components?* This question forms a basis data set for the following investigations. Its results can provide useful insights into where the complexity in ATL transformations originates from to provide starting points for more focused investigations. It can also help to uncover potential strengths and weaknesses of the abstractions used by ATL (*H1*).

RQ2: *When looking at the complexity distributions of individual transformation components, are there any salient characteristics?* ATL components such as the out-pattern consist of a set of bindings that assign values to the attributes of the output model. The question that arises from such structures is, whether the complexity of out-patterns stems largely from single complex bindings or a number of simpler bindings. With this research question we aim to investigate such effects which can indicate points where ATL does a good job of hiding complexity (*H1*)

RQ3: *How does the usage of refining mode impact the complexities of ATL modules?* ATLs refining mode was introduced to ease refinement transformations by allowing developers to only focus on the code generating modified elements while leaving all other elements unchanged. Accordingly, the complexity of refining mode transformations should originate to large parts in refining activities. Otherwise it would indicate that the refining mode fails in supporting developers with model refinements. This in turn would be a counterexample to the claims made in *H1*.

[1] `https://www.eclipse.org/atl/atlTransformations/`

RQ4: *How large is the percentage of bindings that require trace-based binding resolution?* Before being able to argue about the usefulness of trace information (*H2*) for model transformations it should be investigated to what extent their existence influences a model transformation script. If only a small proportion of transformations utilize traces then maybe the development effort for implicit trace handling is not worth it.

RQ5: *What portion of ATL transformations use implicit rule ordering?* The amount of implicitly ordered rules compared to manual rule ordering can be a good indication into whether the feature is well liked by developers hinting at an advantage over manual ordering.

To answer the research questions we selected a total of 33 ATL transformations from various sources to analyse. We use two sets of complexity measures based on Lano et al. (2018) to measure the complexity of ATL transformations. A meta-model representing the basic components of ATL modules is used to compile the complexity values together. Information about trace usage and rule ordering is taken directly from the models representing the ATL transformations.

The remainder of this paper is structured as follows: First in Section 5.2 an introduction into relevant aspects of ATL is given. Section 5.3 defines the used complexity measures. Afterwards in Section 5.4 we present our extraction and analysis procedures. The results of our analysis are then presented in Section 5.5. Section 5.7 discusses potential threats to the validity of the described proceedings while Section 5.6 places the approach in the context of existing work. Lastly Section 5.8 concludes and proposes potential future work.

5.2 The Atlas Transformation Language (ATL)

Specifications in ATL are organized in one of three kinds of so called *Units*. A unit is either a *module*, a *library* or a *query*. Depending on their type, units can consist of *rules*, *helpers* and *attributes*, which are a special kind of helper.

ATL uses the Object Constraint language (OCL) (OMG 2014) for both data types and expressions.

5.2.1 Modules

Modules are used to define transformations. ATL modules are made up of three segments (see Listing 5.1): the *module header* which defines the modules name as well as the types of the input and output meta-models, a number of optional *imports* and a set of *helper* and *rule* definitions.

```
1 module NAME
2   create OUT1:OUTTYPE1, ...
3   [from|refining] IN1:INTYPE1, ...
4
5   [uses LIBRARY]*
6   [RULEDEF|HELPERDEF]*
```

List. 5.1: Structure of an ATL module

Libraries consist of a set of helper definitions. Libraries can be imported into modules.

Lastly, **Queries** are comprised of an import section, a *query* element and a set of *helper* definitions. Queries are used to define transformations from models to simple OCL types rather than output models.

5.2.2 Helpers and Attributes

Helpers allow developers to define outsourced expressions that can be called from within rules. Helper definitions can define a data type for which the helper is specified, called *context*. ATL also allows developers to define so called *Attribute* helpers. The main difference between *attributes* and *helpers* is that *attributes* do not accept parameters. *Attributes* serve as constants that are defined for a specific context.

The definition of both traditional helpers and attribute helpers follow the same syntax patterns (see Listing 5.2). The only difference lies in whether input parameters are defined.

```
helper [context CONTEXTTYPE]? def : NAME[(PARAMETERS)]? : TYPE = EXPR;
```

List. 5.2: Syntax to define Helpers

5.2.3 Rules

In ATL, *rules* are used to specify the transformation of input models into output models. There exist two main types of rules: *called rules* and *matched rules*. Matched rules enable a declarative way to define how a model element of a specific type is transformed into output model elements, while called Rules enable generation of target model elements from imperative code. Matched rules are executed automatically on all matching input model elements by the ATL engine.

Matched rules are comprised of four main sections (see Listing 5.3):

An *In-Pattern* which defines source model elements that are being transformed. In-Patterns can contain a filter expression which defines a condition that must be met for the rule to be applied.

An optional *Using-Block* that allows to define local variables.

The *Out-Pattern* which defines a number of output model elements that are created for the model element defined in the in-pattern when the rule is applied. Each output model element is defined by an *Out-Pattern element* which contains so called *bindings* that assign values to attributes of the model element.

And lastly an optional *Action-Block* which allows the specification of imperative code that is executed once the target elements have been created.

```
[lazy| unique lazy]? rule NAME {
  from
    INVAR : INTYPE [(CONDITION)]*
  [using {
    [VAR : VARTYPE = EXPR;]+
  }]?
  to
    [OUTVAR : OUTTYPE {
      [ATR <- EXPR,]+
    },]+
  [do {
    [STATEMENT;]*
  }]?
}
```

List. 5.3: Syntax to define matched rules

Apart from regular **matched rules** there are also **lazy** rules. They are defined by adding the key word *lazy* in front of a matched rule definition. Lazy rules are executed only when explicitly called for a specific model element that matches the rules type and filter expression. Lazy rules can be called multiple times on the same model element to produce multiple distinct output elements.

Unique lazy rules, defined through the *unique lazy* key words, change this behaviour. Instead of producing a new model element for each call, unique lazy rules always return the same output element when called on the same input model element.

Lastly, **called rules** are defined in a similar fashion to **matched rules** (see Listing 5.4). The main difference between the two being that **called rules** do not contain an *In-Pattern* and allow the definition of required parameters.

5.2.4 Refining mode

The refining mode is a special execution mode for ATL rules which is intended to assist developers with refactoring models, i.e., endogenous transformations.

Normally, the ATL engine only produces output model elements for input elements on which rules are executed on. When using the refining mode however, the ATL engine executes all rules on matching input elements and produces a copy of all unmatched elements. This way developers are able to focus solely on the refining part of their refactoring efforts according to the language developers.

```
rule NAME([PARAMETER,]*) {
  [using {
    [VAR : VARTYPE = EXPR;]+
  }]?
  to
    [OUTVAR : OUTTYPE {
      [ATR <- EXPR,]+
    },]+
  [do {
    [STATEMENT;]*
  }]?
}
```

List. 5.4: Syntax to define called rules

5.3 Complexity Measures

There exist several approaches for measuring complexity of model transformation languages and ATL in particular (Di Rocco et al. 2015; Kapová et al. 2010; Lano et al. 2018; Tolosa et al. 2011; Vignaga 2009). Most of these approaches use a simple metric that relates the number of transformation components such as rules or helpers to the complexity of a transformation module. In our opinion, however, the number of rules or helpers alone does not capture the complexity of model transformations well enough. For that reason, we opted to adopt the complexity measure proposed by Lano et al. (2018) which includes not only the number of transformation components but also the complexity of expressions used within the transformation.

In the following, the complexity measures will be explained.

5.3.1 Syntactic complexity

The syntactic complexity $c(\tau)$ of a transformation specification τ is defined based on the complexity of expressions and activities within the defined transformation (Lano et al. 2018). The general idea behind it being that the complexity of each construct is comprised of a static value for the construct itself plus the sum of the complexities of its contained elements.

The complexity of a module as defined in Listing 5.1 would be comprised of the sum of the complexities for its contained helper definitions and rule definitions. The complexity of rules, defined as shown in Listings 5.3 and 5.4, is then comprised of the complexity of their contained `from`-block (In-Pattern), the `to`-block (Out-Pattern), the `using`-block and `do`-block plus a static value of 1 for the rule itself.

The complexity of In-Patterns is defined by their contained filter expression and a static value for the construct itself, while the complexity of Out-Patterns is defined by a static value for the construct as well as the sum of the complexities of all contained Out-Pattern elements and their contained bindings. An overview over the most important complexity measure definitions can be found in Table 5.1 for expressions and Table 5.2 for activities/structural elements[2].

We adopted the complexity measure with slight modifications since we disagreed with certain defined values. The following adjustments were made to the definition from Lano et al. (2018):

First, the complexity of helpers was adapted to also include the complexity of their context. The reason for this change being the fact that the context of a helper has to be considered when trying to understand its function. Furthermore, in our opinion there is no difference between attribute and operation helpers, the additional, static complexity attributed to both types of helper definitions was aligned at 1. For this the static complexity of attribute helpers was reduced from 3 to 1 and that of operation helpers was increased from 0 to 1.

Action blocks were given an additional static complexity value of 1 which was missing from the definitions of Lano et al. (2018). This aligns it with the static complexity that is attributed to all elements contained within rules, i.e. In-Patterns, Out-Patterns and Using-blocks.

The complexity attributed to operation calls was increased to 1 to align it with that of attribute and navigation calls. In our opinion calling an operation on an object is just as complex as accessing

[2]Full definitions can be found in `https://spgit.informatik.uni-ulm.de/stefan.hoeppner/mtl-complexities/-/blob/master/ATL/transformations/qvt/transforms/complexity.qvto`

one of its attributes. For the same reason the complexity for collection operation calls was also increased to 1 as well.

Table 5.1: Definitions of expression complexity measure based on Lano et al. (2018).

Expression e	Complexity $c(e)$
Numeric, boolean or String value	0
Identifier *iden*	1
Attribute call *source.attr*	$c(source) + 2$
Operation call $source.op(p1, ..)$	$2 + c(source) + \sum_i c(p_i)$
Operator call $e1$ *op* $e2$	$c(e1) + c(e2)$
CollectionOperation call $source- > op(p1, ..)$	$2 + c(source) + \sum_i c(p_i)$
$if\ e1\ then\ e2\ else\ e3\ endif$	$c(e1) + c(e2) + c(e3) + 1$
$let\ v\ :\ t\ =\ e1\ in\ e2$	$c(t) + c(e1) + c(e2) + 4$
CollectionExpression $Col\{e1, ..\}$	$1 + \sum_i c(e_i)$
Primitive Type (Integer,String,...)	1
Collection Type $Col(t)$	$1 + c(t)$

Table 5.2: Definitions of complexity measure for ATL elements/activities based on Lano et al. (2018). ATL elements are capitalized while expression elements are written in lower case.

ATL element A	Complexity $c(A)$
Module $H_1, .., R_1, ..$	$\sum_i c(H_i) + \sum_i c(R_i)$
Helper $helper\ context\ c\ def\ :\ n\ :\ t\ =\ e$	$c(c) + c(t) + c(e)$
MatchedRule $rule\ N\ \{From\ Using\ To\ Do\}$	$c(From) + c(To) + c(Do) + c(Using)$
CalledRule $rule\ N(p)\ \{Using\ To\ Do\}$	$c(To) + c(Do) + c(Using)$
VariableDefinition $n\ :\ t\ =\ e$	$c(t) + c(e) + 3$
InPattern $from\ s\ :\ t\ (f)$	$c(f) + c(t) + 3$
OutPattern $o : t\ \{B_1, ..\}$	$c(t) + \sum_i c(B_i) + 2$
Binding n <- e	$c(e) + 2$
ActionBlock $do\ \{S\}$	$c(S)$
$S1; S2$	c(S1) + c(S2)
if e then $S1$ else $S2$	$c(e) + c(S1) + c(S2) + 1$
for v : e do S	c(e) + c(S) + 1
Binding Statement v <- e	c(v) + c(e) + 1

5.3.2 Computational complexity

The computational complexity is an extension of the syntactic complexity. Its goal is to more closely capture the underlying complexity of transformation definition with respect to outsourced expressions and called transformation rules. To achieve this, the complexity of `Operation Calls` is calculated by taking the complexity of the called operation into account instead of adding a static value regardless of the called operation. For example given a helper `sample` of syntactic complexity 12, the call `sample()` has a syntactic complexity of 2 whereas its computational complexity amounts to 12.

Moreover the complexity of used variables is also resolved by taking the definition expression of the variable into account instead of using a static value of 1.

5.4 Methodology

Apart from the selection of the ATL transformation modules to analyse, we strongly oriented our proceedings along the research questions from Section 5.1.

Table 5.3: Meta-data about the analysed transformation modules.

Data	minimum	average	maximum	total
LOC	39	408	1364	13455
Rules	1	14	55	460
Helpers	0	11	74	376
Bindings	2	112	487	3695

5.4.1 Module Selection

The selection of ATL modules was aimed to achieve a wide spread of transformations based on their source, purpose and size in terms of lines of code. We also aimed to achieve an even distribution of modules that use the refining mode and modules that do not.

For this purpose, we searched GitHub for ATL projects by using the search string 'ATL transformation' and included all novel (meaning not present in the ATL zoo) transformations for which we were also able to find the input and output meta-models since those were required for parts of our analysis (see Section 5.4.4). This resulted in a total of 16 transformation modules. Additionally we included the R2ML2XML transformation from Marcel F van Amstel et al. (2011a) and the Families2Persons transformation from the ATL zoo because it is a widely used example for model transformations. We then supplemented the set of transformations with transformations from the ATL zoo to try and achieve an even distribution between modules that use the refining mode and modules that do not.

The result was a set of 33 ATL transformations (some meta-data about the transformations can be found in Table 5.3). Of those 33 transformations, 15 use the refining mode of ATL while 18 are exogenous transformations. A complete overview over the selected transformations, including names and sources can be found under `https://spgit.informatik.uni-ulm.de/stefan.hoeppner/mtl-complexities/blob/master/ATL/resources/input/cases/justifications`.

5.4.2 RQ1,2: How is the complexity of ATL transformations distributed over multiple transformations and transformation components and are there any salient characteristics?

To be able to collect and analyse complexity data of ATL transformations and relevant elements thereof a meta-model was constructed[3]. Its structure was designed to be able to break down the full representation of an ATL transformation into the basic components that make up ATL transformations as described in Section 5.2. With this structure it is also possible to investigate where the complexity of entire ATL modules and rules originate from, e.g. whether a rule is complex because of its size or due to a few complex contents like filter expressions. The design of the meta-model followed the principles of abstraction and pragmatics. Compared to the ATL meta-model our developed meta-model focuses solely on those parts of the ATL transformations we are interested in and provides an easy way to track their complexity and the origin thereof.

To transform transformation modules into a model of the presented meta-model and to calculate the complexities of its components along the way, a QVT-O transformation[4] was defined. Its correctness was evaluated using unit tests: A test module containing at least one of each activities/expressions for which a complexity value can be calculated was defined. The complexity values for each element was calculated manually based on the previously introduced complexity definition. Afterwards the results of the transformation were manually compared with the manually calculated complexity values. Discrepancies between the complexity values were investigated and corrected.

In order to collect data for analysis, the tested transformation was applied to the 33 ATL transformations.

Apart from the raw complexity data, we resorted to using several diagrams such as histograms, violin and alluvial plots as well as code snippets to investigate the complexity distribution, both syntactical and computational, of ATL transformations.

[3] the meta-model can be found under `https://spgit.informatik.uni-ulm.de/stefan.hoeppner/mtl-complexities/-/tree/master/ATL/metamodels/ATLComplexity/model`

[4] `https://spgit.informatik.uni-ulm.de/stefan.hoeppner/mtl-complexities/blob/master/ATL/transformations/qvt/transforms/complexity.qvto`

In order to better understand the meaning behind the complexity values example code snippets for each component were extracted from the 33 selected transformation modules. The code snippets were selected so that their complexity values correspond to the components median complexity within the data set. One such code snippet can be seen in Listing 5.5. All used snippets can be found under `https://spgit.informatik.uni-ulm.de/stefan.hoeppner/mtl-complexities/tree/master/ATL/resources/input/medians`.

```
1 helper context SimpleClass!Class def: associations: Sequence(SimpleClass!Association)=
2   SimpleClass!Association.allInstances() -> select(asso | asso.value = 1);
```

List. 5.5: Helper with a syntactic complexity corresponding to the median of all helper complexities.

5.4.3 RQ3: How does the usage of refining mode impact the complexities of ATL modules?

As explained in Section 5.1, we also intended to analysed ATL modules using the refining mode as a example of how transformation languages hide semantics.

To do so, we used the 15 selected transformation modules that use refining mode and analysed their complexities separately and in comparison to those modules not using the refining mode.

5.4.4 RQ4: How large is the percentage of bindings that require trace-based binding resolution?

To investigate the usefulness of trace-based binding resolution (and thus to an extent that of implicit trace management) we resorted to analysing how often it is used in transformation modules. A high proportion of trace-based resolutions used would then indicate their usefulness. Since trace-based binding resolution only happens along reference types of the input and output elements we extracted all reference types per module element for all output meta-models. For this we used a simple Java-program that given an Ecore-file would produce a list of reference types for each contained `EClass`.

Afterwards the bindings within all selected transformation modules were analysed for usage of the extracted reference types. The amount of bindings that use traces compared to simple assignments was then analysed on the basis of these results.

5.4.5 RQ5: What portion of ATL transformations use implicit rule ordering?

Similar to the trace usage, the usefulness of implicit rule ordering can be indicated by the distribution of implicitly ordered transformation elements compared to explicitly ordered ones.

Called and lazy matched rules all get explicitly ordered by developers when calling them while matched rules enable the ATL transformation engine to traverse the source model and implicitly order their execution. Thus the ratio of matched rules to called and lazy rules gives an indication into how relevant implicit rule ordering is for model transformations.

Data for this analysis can be gathered from both the complexity distributions from *RQ1* as well as directly from the number of definitions.

5.5 Result Summary and Analysis

We present the results of our analysis in this section in accordance with the research questions posed in Section 5.1.

5.5.1 RQ1: How is the complexity of ATL transformations distributed over multiple transformations and transformation components?

Figures 5.1 and 5.2 show alluvial plots over the distribution of syntactic complexity and computational complexity respectively of module elements within ATL transformation modules. They

display how much of the complexity of all investigated transformations originate in which components beginning with the modules themselves following the definitions down to the contained expressions and the static value associated with each component.

Interestingly, while making up nearly 45% of all top level definitions, **Helpers** only contribute to roughly 18% of the total complexity of a transformation module[5]. The largest portion of complexity is attributed to **matched rules** which contribute to over 3/4 of the total complexity of transformation rules while accounting for 53% of all top level definitions. And lastly **called Rules**, which are not widely represented in our data sets, while making up about 1% contribute to 5% of the overall complexity of modules.

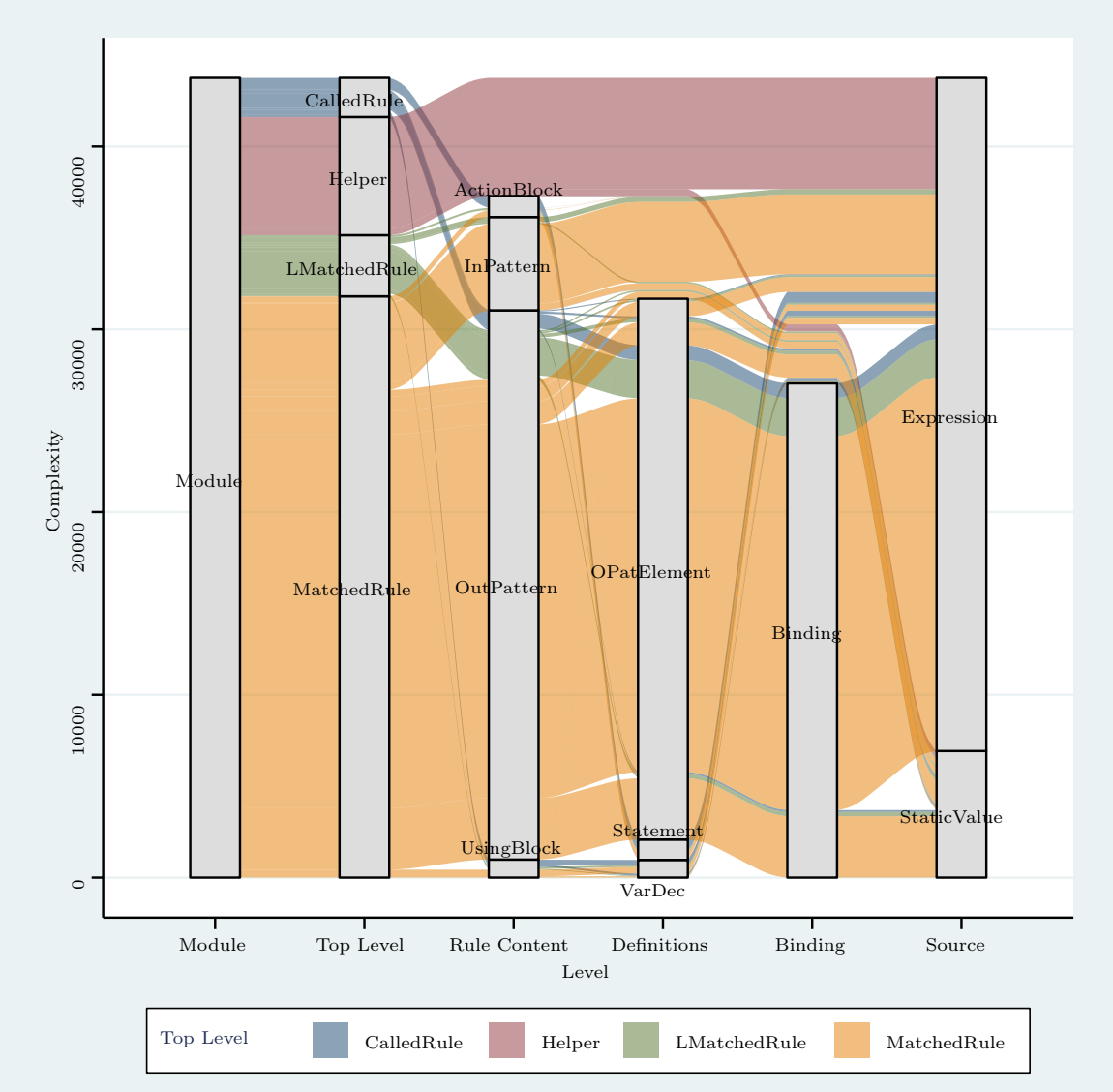


FIGURE 5.1: Distribution of syntactic complexity over all ATL modules.

Another observation that can be made from Figure 5.1, is that about 80% of the syntactic complexity of (lazy) **Matched-rules** stems from their **Out-Patterns** while only 15% come from **In-Patterns** and a nearly negligible 5% originate in *action-* and **using blocks**. Following this trend downwards 73% of the complexity of these rules stems from their contained bindings, i.e assigning a value to attributes of the output model element. Meaning most effort in transformations is spent not in selecting the correct model elements to transform but simply assigning the output values (see Section 5.5.2 for a more detailed discussion). This effect is still present when looking at the

[5]the raw data can be found under `https://spgit.informatik.uni-ulm.de/stefan.hoeppner/mtl-complexities/tree/master/ATL/data`

computational complexity distribution (as shown in Figure 5.2) which rules out the possibility that the effect is created by outsourcing of filter conditions in In-Patterns through helpers. This leads us to:

Observation 1: Over half of the effort spent in writing ATL transformations is spent assigning values to the output model.

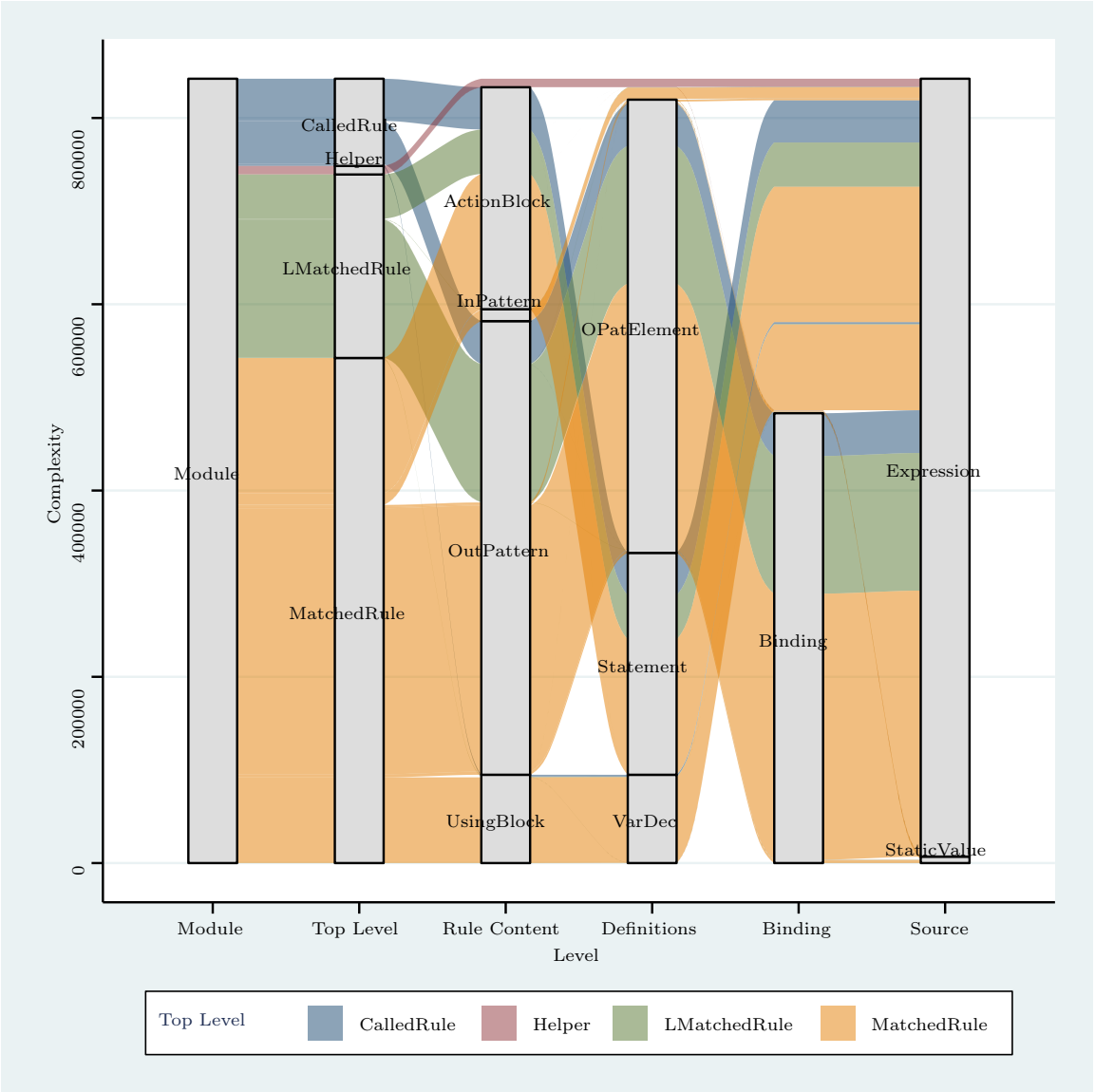


FIGURE 5.2: Distribution of computational complexity over all ATL modules.

Furthermore the 15% of matched rules syntactic complexity that comes from In-Patterns shows that conditioning the application of transformation is a relevant task for model transformations. The low proportion especially when considering the computational complexity rather suggests that the conditioning on types (as opposed to filter conditions without pre conditioning on types) which ATL does for all matched rules by default alone already provides a useful abstraction for model transformations. This assumption is supported by the fact that about 25% of all matched rules get by with only using the standard type conditioning without any additional filter expression in the **In-Pattern**. Of those 25% only 12% (which therefore constitute only 3% of all matched rules) are trivial transformation rules. Simple transformations in the context of this paper mean transformations that contain no filter condition and only assign attributes from the input model element to the output model element without doing any additional operations. The fact that a large portion of transformations get by with only the default conditioning on types in ATL leads us to:

Observation 2: Conditioning on types provides an abstraction well suited for model transformations.

5.5.2 RQ2: When looking at the complexity distributions of individual transformation components, are there any salient characteristics?

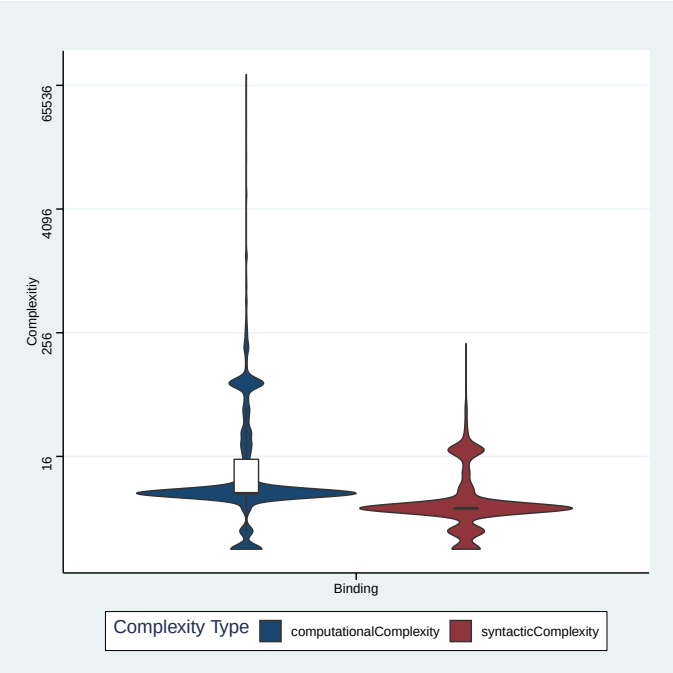


FIGURE 5.3: Syntactic complexity distribution of Bindings.

As previously mentioned, the proportion of the complexity of bindings within transformations also stands out in Figure 5.1. Bindings alone make up over half of the complexity of transformations, a trend, that persists even when looking at the computational complexity of transformation modules. Interestingly, the complexity within bindings is very unevenly distributed. Figure 5.3, which shows a violin and box plot for the syntactic complexity distribution of bindings (note the logarithmic scale of the y axis), illustrates this. The majority of all bindings have a syntactic complexity 5 ($b_5 = 60\%$). This corresponds to directly accessing an attribute of an object as shown in Listing 5.6, calling a helper on said object or accessing a global attribute (`thisModule.attribute`).

Further analysis shows that a total of 93% of all bindings with a syntactic complexity of 5 do indeed stem from direct accesses of attributes of the input model element ($b_{5a} = 93\%$). Only 2% are global attribute accesses and the last 5% originate from helper calls on the source model element. This is also indicated by the fact that a majority of bindings have a computational complexity of 7 which can only correspond to accessing attributes on input elements. In summary, this leads us to:

Observation 3: Over half ($b_5 * b_{5a} = 56\%$) of all bindings are used to map one attribute of an input model element to one attribute of an output model element.

Adding to this point, only 5% of bindings with a syntactic complexity of 5 stem from trivial transformations, i.e., transformations that simply map attributes from an input element to an output element without doing any meaningful filtering or modification of the content. This reinforces *Observation 3* since we can rule out that the majority of bindings with complexity of 5 stem from trivial transformations which do by definition only contain bindings of this or lower complexity .

Looking at ATL modules as a whole *Observation 3* means that 33% of their total syntactic complexity comes from the activity of copying input model attributes to output model attributes.

That much of the complexity of transformation modules comes from bindings means that the main effort when writing model transformations in ATL consists of defining how the output should

```
rule MedianBinding {
  from s : Families!Member
  to t : Persons!Female (
    fullName <- s.firstName
  )
}
```

List. 5.6: Rule containing a binding with a syntactic (computational) complexity of 5 (7)

look which is actually one of the main goals of model transformation languages. This in turn suggests that ATL does a good job in abstracting away other tasks in model transformation such as model traversal, conditioning on types as shown in Section 5.5.1, tracing and rule ordering to which we will come in Sections 5.5.4 and 5.5.5.

5.5.3 RQ3: How does the usage of refining mode impact the complexities of ATL modules?

Given the observations from the previous sections we would expect that the syntactic complexity distribution of bindings to deviate away from 5 (and the computational complexity form 7) since the refining mode is designed to enable developers in focusing only on the refining part of the transformation.

In the transformations investigated for this paper this is however not the case as can be seen in Figure 5.4, the median syntactic complexity of bindings remains 5 and that of computational complexity remains 7.

This indicates the usefulness of the changes made to the refining mode with the introduction of the 2010 ATL compiler. Since 2010 refining mode allows real in-place transformations which means that rules only need to specify changes to elements while all the other elements remain untouched. And because the main effort in the investigated transformations, which were all defined for ATL compilers prior to 2010, is spent on copying attributes from the input model element (98% of all bindings) to its output counterpart, newer versions of the ATL compiler would heavily reduce this necessary overhead, allowing developers to focus solely on actually refining models. To us this suggests that the current versions of ATLs refining mode can significantly reduce unnecessary overhead for refining transformations. There is also an observation to be made from this:

Observation 4: GitHub and especially the ATL Zoo lack samples of ATL transformations using the refining mode with compiler versions at least as current as 2010.

Furthermore, In-Patterns in refining mode are, on average about twice as complex as in non refining modules. Moreover only a small portion ($\sim$ 7%) of In-Patterns do not contain a filter expression at all compared to 1/3 of In-Patterns in non refining mode.

This leads us to two additional observations:

Observation 5: When refining models, filters are more heavily used than when transforming between different meta-models.

Observation 6: Filter expressions are more complex in refining mode due to having to select elements with more specific properties.

5.5.4 RQ4: How large is the percentage of bindings that require trace-based binding resolution?

About 15% of all bindings in the analysed transformation modules require traces. While this makes it apparent that traces are less frequently required than one would expect, it still demonstrates their necessity since 15% is not a negligible proportion. This leads us to:

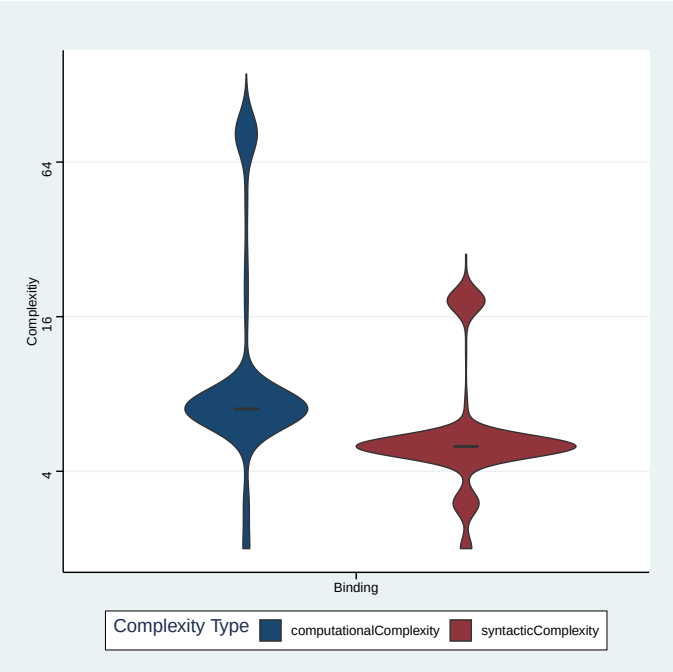


FIGURE 5.4: Complexity distribution of Bindings in refining mode.

Observation 7: Bindings that require traces constitute a significant part of the model transformations considered.

It is also worth mentioning that while such trace resolution can save developers substantial amounts of time they can also be a source of errors.

Considering the complexity of the bindings that require traces also reveals something interesting. About half of all bindings that require traces have a syntactic complexity of 5 and computational complexity of 7. This shows how well automatic trace handling can hide complexity. The developer can simply access the input model element that is supposed to be transformed into the correct output model element and the transformation engine handles resolving and referencing. Would the developer have to take care of this process manually both syntactic and computational complexity would be significantly higher since this would require identifying and accessing the corresponding output model element through additional code.

5.5.5 RQ5: What portion of ATL transformations use implicit rule ordering?

In the ATL modules analysed for this study, a total of 460 rules are defined. Of those 364 or 79% are matched rules, 84 or 18% are lazy matched rules which need to be invoked to transform model elements and only 12 or 2% are called rules.

Our results, deviate slightly from the results found by Selim et al. (2017) but still reveal the same preference trend of ATL developers:

Observation 8: Developers strongly prefer matched rules over lazy matched rules and called rules.

Since matched rules allow for implicit model traversal and rule ordering this can indicate that these concepts provide good support for transformation developers. This is also evident from the fact that the proportion of Out Pattern complexity (both syntactic and computational) to Action Block complexity is far more balanced in called rules than in (lazy) matched rules (see Figures 5.1 and 5.2) again indicating that called rules require more structural code such as calling other rules and conditioning.

5.6 Related Work

Di Rocco et al. (2015) analyse the impact of input and output meta-models on, amongst other things, the complexity of ATL transformations. For this purpose they use a number of meta-model metrics and correlate these with metrics for ATL transformations using Spearman's rank correlation coefficient. Their findings include a high correlation between the number of structural features of the output meta-models and the number of used bindings in an ATL transformation module. An insight which can be reflected upon in our results. In contrast to the complexity measures applied in this work however, the measures proposed for complexity in their work is confined to the number of structural features such as bindings or helpers of ATL transformations rather than the complexity of their structure and contained expressions. Which is not to say that the applied measure is not indicative of the complexity of transformations rather that it is only part of what makes a transformation complex in our opinion.

Marcel F van Amstel et al. (2011a) propose the usage of *cyclomatic complexity* to measure the complexity of helpers. They also envision incorporating the complexity of the contained OCL expression into its complexity measure. Similarly to Di Rocco et al. (2015), they also use the number of different transformation components as metrics for measuring ATL transformations. The described metrics are applied to seven transformations. And the resulting values are then related to quality attributes, based on the assessment of nineteen experts, such as *understandability*, *maintainability* and *conciseness* using Kendall's τ_b correlations. Notable results include a significant correlation between the number of transformation rules and *conciseness* and the number of out-patterns and *understandability*. In comparison to this, we try to draw direct conclusions about the structure and structure of transformations from our gathered data instead of about quality features.

Vignaga (2009) relate the complexity of ATL transformations to a variety of introduced metrics. Most of the related metrics are once again quantifications of different components within ATL modules such as the *number of matched rules* or *average number of filters* used in rules. They also relate the *cyclomatic complexity* to complexity much like Marcel F van Amstel et al. (2011a). As previously mentioned we believe that the number of components are only part of what makes transformations complex which is why the used complexity measure in this paper also incorporates the complexity of expressions.

The numbers of ATL transformation components have also been used by Tolosa et al. (2011) to make comparisons between several transformation modules to investigate the feasibility of applying transformations to transformation modules. The authors concede that the applied metrics need further research and development and predict that such measures could assist with identifying aspects of ATL transformations to optimize.

Similarly Angelika Kusel et al. (2013) analysed the ATL Zoo with the goal to gain insights about the frequency of use of reuse mechanisms. For this the authors devised a semi-automated process to extract and analyse projects from the ATL zoo. They found that reuse mechanisms are exclusively used within a transformation and that helpers are the most frequently used reuse mechanism while only little rule inheritance is used. In contrast to their work, our focus does not directly relate to reuse mechanisms although the computational complexity was introduced in part to account for the outsourcing of complexity due to reuse mechanisms.

5.7 Threats to validity

This section addresses the potential threats to validity identified for the performed study.

The transformations evaluated for the purpose of this study were chosen from various sources to reduce the influence of programming habits of individual transformation engineers. Consequently the purposes and characteristics of the transformations vary immensely. To be able to compare transformation modules using refining mode with modules that do not use refining mode we also aimed to use a similar amount of respective transformation modules. While this strengthens the external validity of our comparison it can potentially lead to a reduction in the external validity of our other findings since an even distribution of refining and non refining modules is potentially less representative of the overall ATL ecosystem. Given the selection of transformation modules it is also not possible to draw representative conclusions about model transformation languages in general but rather for ATL specifically.

There is of course a discussion to be held about the complexity measure used. As discussed in Section 5.6 most research uses the number of elements as basis for complexity measures. We

and Lano et al. (2018) argue that this alone does not fully cover the complexity of transformations. The syntactic complexity measure used in this study uses the complexity of expressions and activities as defined in Tables 5.1 and 5.2. The number of elements are also taken into account in these definitions but do not constitute the majority of the complexity value of an ATL transformation this is reserved to the complexity of expressions used within the transformation modules as evident from Figure 5.1. While we are missing a formal validation of the measures used we believe that this indicates their overall usefulness. The computational complexity is then a natural extension of the syntactic complexity to more closely resemble the actual complexity that is hidden in operation calls in expressions.

5.8 Conclusion and Future Work

In this work we presented our results of analysing ATL modules to provide insights into three common claims about the advantages of model transformation languages, namely that transformation languages hide complex semantics behind simple syntax, that automatic trace handling in transformation languages is advantageous and that implicit rule ordering supports developers in defining transformations.

For this purpose we used two complexity measures to investigate how complexity is distributed over ATL transformation modules which we applied to a total of 33 modules. We also analysed the proportions of matched rules compared to other types of rules and the proportion of bindings that require trace information to be resolved.

We found, that while transformations can get complex, the complexity originates mainly in definitions of how the output models should be populated rather than how the transformation should be executed. To us this provides an indication for how well ATL abstracts away from certain tasks necessary for model transformation such as model traversal, rule ordering and trace handling.

We have also shown that conditioning on types is well suited for model transformations since a total of 22% of all non-trivial matched rules get by with only filtering on types. This also provides a clear example why implicit rule ordering can be beneficial for model transformation definitions since developers can simply define to which kind of input model element a transformation should apply and the transformation engine handles execution.

This is further supported by the fact that we found that nearly 80% of all defined rules are matched rules which make use of exactly this mechanism.

Next we analysed required trace information in bindings. We came to the conclusion that while bindings that do require trace information are outweighed by those that do not, they still constitute a significant portion of model transformations. And while this suggests that automatic trace handling is advantageous further research is necessary to more precisely capture its impact.

Lastly we compared the complexities of transformation modules using the refining mode with those that do not. We found that while the complexity of matched rules defined in a refining module is much higher, the increase in complexity can be attributed to an increase in simple bindings. A fact we were able to attribute to the use of older ATL compilers which did not allow in-place refinements.

For future work, we are interested in repeating the described proceedings on transformations written in general purpose programming languages. While the resulting values can not be compared directly, the complexity distributions can be used to gain insights into where the complexity in these transformation definitions lie. Which we believe can produce further contributions to the discussion of GPLs vs MTLs for defining model transformations.

Chapter 6

Paper E

Contrasting Dedicated Model Transformation Languages Versus General Purpose Languages: A Historical Perspective on ATL Versus Java Based on Complexity and Size

S. Höppner, M. Tichy, T. Kehrer

International Journal on Software and Systems Modeling (SoSyM), volume 21, pages 805–837, 2022
Springer Nature

Abstract

Model transformations are among the key concepts of model-driven engineering (MDE), and dedicated model transformation languages (MTLs) emerged with the popularity of the MDE pssaradigm about 15 to 20 years ago. MTLs claim to increase the ease of development of model transformations by abstracting from recurring transformation aspects and hiding complex semantics behind a simple and intuitive syntax. Nonetheless, MTLs are rarely adopted in practice, there is still no empirical evidence for the claim of easier development, and the argument of abstraction deserves a fresh look in the light of modern general purpose languages (GPLs) which have undergone a significant evolution in the last two decades. In this paper, we report about a study in which we compare the complexity and size of model transformations written in three different languages, namely (i) the Atlas Transformation Language (ATL), (ii) Java SE5 (2004–2009), and (iii) Java SE14 (2020); the Java transformations are derived from an ATL specification using a translation schema we developed for our study. In a nutshell, we found that some of the new features in Java SE14 compared to Java SE5 help to significantly reduce the complexity of transformations written in Java by as much as 45%. At the same time, however, the relative amount of complexity that stems from aspects that ATL can hide from the developer, which is about 40% of the total complexity, stays about the same. Furthermore we discovered that while transformation code in Java SE14 requires up to 25% less lines of code, the number of words written in both versions stays about the same. And while the written number of words stays about the same their distribution throughout the code changes significantly. Based on these results, we discuss the concrete advancements in newer Java versions. We also discuss to which extent new language advancements justify writing transformations in a general purpose language rather than a dedicated transformation language. We further indicate potential avenues for future research on the comparison of MTLs and GPLs in a model transformation context.

6.1 Introduction

Model transformations are among the key concepts of the model-driven engineering (MDE) paradigm (Sendall et al. 2003). They are a particular kind of software which needs to be developed along with an MDE tool chain or development environment. With the aim of supporting the development of model transformations, dedicated model transformation languages (MTLs) have been proposed and implemented shortly after the MDE paradigm gained a foothold in software engineering.

6.1.1 Context & Motivation

In the literature, many advantages are ascribed to model transformation languages, such as better analysability, comprehensibility or expressiveness (Götz et al. 2021a). Moreover, model transformation languages aim at abstracting from certain recurring aspects of a model transformation such as traversing the input model or creating and managing trace information, claiming to hide complex semantics behind a simple and intuitive syntax (Gray et al. 2003; Jouault et al. 2008; Krikava et al. 2014; Sendall et al. 2003).

Nowadays, however, such claims have two main flaws. First, as discussed by Götz et al., there is a lack of actual evidence to have confidence in their genuineness (Götz et al. 2021a). Second, we argue that most of these claims emerged together with the first MTLs around 15 years ago. The Atlas Transformation Language (ATL) (Jouault et al. 2006), for example, was first introduced in 2006, at a time when third generation general-purpose languages (GPLs) were still in their infancy. Arguably, these flaws are underpinned by the observation that MTLs have been rarely adopted in practical MDE (Burgueño et al. 2019).

Within our research group as well as in conversations with other researchers, the presumption that transformations can just as well be written in a GPL such as Java has been discussed frequently. In fact, in our own research, we have implemented various model transformations using a GPL; examples of this include the meta-tooling facilities of established research tools like SiLift (Kehrer et al. 2012) and SERGe (Kehrer et al. 2016; Rindt et al. 2014), or the implementation of model refactorings and model mutations in experimental setups of more recent empirical evaluations (Schultheiß et al. 2020a,b). The presumption that model transformations can just as well be written in a GPL has been confirmed by a community discussion on the future of model transformation languages (Burgueño et al. 2019), and, at least partially, by an empirical study conducted by Hebig et al. (2018). Our argumentation for specifying model transformations using a modern GPL is mainly rooted in the idea that new language features allow developers to heavily reduce the boilerplate code that MTLs claim to abstract away from. There are also other features that certain model transformation languages can provide such as graph pattern matching, incrementality, bidirectionality or advanced analysis but for now our study focuses solely on the abstraction and ease of writing argument.

6.1.2 Research Goals and Questions

To validate and better understand this argumentation, we elected to compare ATL, one of the most widely known MTLs, with Java, a widespread GPL. More specifically, we compare ATL with Java in one of its recent iterations (Java SE14) as well as at the level of 2006 (Java SE5) when ATL was introduced[1]. The goal of this approach is twofold. First, we intend to investigate how transformation code written in Java SE14 can be improved compared to the Java code using the Java version SE5 that was timely when ATL was released. Second, we want to contextualize these improvements by relating them to transformation code written in ATL. We opted to use both size and complexity measures for this purpose because both can provide useful insights for this discussion.

In order to achieve these goals, we developed four research questions to guide our research efforts:

RQ1: How much can the complexity and size of transformations written in Java SE14 be improved compared to Java SE5?

RQ2: How is the complexity of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to each other and ATL?

RQ3: How is the size of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to each other and ATL?

[1]Interestingly, there was no significant evolution of the ATL language since its initial introduction in 2006 (Burgueño et al. 2019).

Table 6.1: Meta-data about the selected transformation modules.

Transformation Name	LOC	# rules	# helpers
ATL2BindingDebugger	41	2	0
ATL2Tracer	96	2	0
DDSM2TOSCA	582	19	2
ExtendedPN2ClassicalPN	86	7	0
Families2Persons	49	2	2
istar2archi	99	6	1
Modelodatos2FormHTML	127	9	3
Palladio2UML	189	19	0
R2ML2XML	1125	60	1
ResourcePN2ResourceM	44	3	1
SimpleClass2RDBMS	63	4	3
UML22Measure	371	27	11
Average	236.25	13.3	2

RQ4: How does the size of query aspects of transformations written in Java SE5 & SE14 compare to each other and ATL?

RQ1 aims to provide a general overview of how both size and complexity of transformations in Java might be improved using language features provided in newer Java versions. For a more detailed discussion and comparison it is then necessary to inspect and compare how the transformation code based is associated to the different aspects of a transformation, e.g. *model traversal*, *tracing* or the actual *transformation* of elements. This is the goal of **RQ2** and **RQ3** for complexity and size, respectively. With these two research questions we aim to investigate for which aspects new language features of Java help to reduce size and complexity of the associated code segments and what this means compared to ATL. Lastly, it is often assumed that querying aided by language constructs in MTLs is one key factor for their suitability over GPLs (Rentschler et al. 2014). With **RQ4** we aim to investigate this assumptions via an explicit comparison between queries written in Java and ATL.

6.1.3 Research Methodology

The process to answer the discussed research questions was structured around four consecutive steps. First, we selected a total of 12 existing ATL transformations taken from the ATL Zoo[2] and several projects from GitHub[3] to the basis for our study. References to all included transformations can be found in our supplementary material by Höppner et al. (2021). The selection of ATL modules was done with several goals in mind. First, we wanted to include transformations of different size and purpose. We also aimed to include both transformations using ATLs' refining mode and normal transformations. Lastly, due to the fact that our translations would be done manually, we decided to limit the total number of transformations to 12 and the maximum size of a single transformation to around 1000 LOC. Since our work is, in part, based on the work presented in (Götz et al. 2020) and their selection criteria align with ours, we opted to make the selection of modules from the set of transformations analysed by them. The module selection process resulted in a total of 12 ATL transformations, from a variety of sources including the ATL Zoo. Basic meta-data about the transformations can be found in Table 6.1, while further details can be found in the supplementary materials.

Next, we devised, and tested, a schema to translate the selected ATL transformations to Java. To develop the translation schema, we followed the design science research methodology (Wieringa 2014) using an iterative pattern for designing and enhancing the schema until it fit our purpose. To validate the correctness of the translated transformations, we used the input and output models that were provided within the ATL transformation projects. The input models were used as input for the Java transformations and the output models were then compared with the output of the ATL transformations.

[2]https://www.eclipse.org/atl/atlTransformations
[3]https://www.github.com

Afterwards, we developed a classification schema to divide Java code into its components and relate each component to the different aspects of the transformation process, i.e., transforming, tracing and traversing. All Java code was then labelled based on the classification schema. For ATL, a similar schema from Götz et al. (Götz et al. 2020) already exists which we adopted and applied to the selected ATL transformations.

Lastly, we decided on and applied several code measures to allow us to compare the transformations. For comparing transformations specified in Java SE5 and SE14, we use a combination of four metrics for measuring size and complexity, namely lines of code (LOC), word count (# words) (Anjorin et al. 2019), McCabe's cyclomatic complexity (McCabe 1976) and weighted method count (WMC). We use WMC based on McCabe complexity, i.e., the sum of the McCabe complexities of all elements, as the complexity measure in cases where the complexities of several elements need to be grouped together. Word count is used to supplement the standard code size measure LOC as a measure that is less influenced by code style and independent from keyword and method name size (Anjorin et al. 2019). Furthermore, word count allows a direct size comparison between ATL and Java, which is hardly possible with LOC due to the languages' significantly different structure.

Our comparison of complexity and size distributions is thus based on LOC, word count, McCabe's cyclomatic complexity and WMC, and we incorporate the findings of Götz et al. on how code is distributed within ATL transformations (Götz et al. 2020).

6.1.4 Results

Our analysis for **RQ1** shows that newer Java versions allow for a significant reduction in code complexity and lines of code, while the number of required words stays about the same. We attribute this to a more information dense style of writing single statements in the more functional programming style enabled by Java SE8 (2014) and newer.

The results for **RQ2** reflect the reduction in complexity overhead mainly in the methods involving model traversal. We also conclude that in newer language versions the most prominent remaining complexity overhead stems from manual trace management in Java compared to ATL.

The more detailed investigation done for **RQ3** support these observations. We show that tracing is not only a prominent part in the methods dedicated to trace management but also in the methods that are dedicated to actually transforming input into output elements.

Overall the results for **RQ2** and **RQ3** suggest that still, a lot of complexity and size overhead for traversal, tracing and supplementary code is required in Java even though newer Java versions improve the overall process of writing transformations. Of these, tracing is the biggest obstacle for efficiently developing transformations in a general-purpose language. The overhead associated with this transformation aspect is the most significant and, arguably, most error-prone one. A large portion of the advancements of Java SE14 over Java SE5 stem from the inclusion of more recent language aspects such as streams and functional interfaces. This fact is highlighted in our results from **RQ4** where those two aspects are the main factors for improvements in the size of OCL expressions written in Java.

6.1.5 Contributions and Paper Structure

This paper extends prior work on comparing Java and ATL transformations (Götz et al. 2021b). The extension consists of (i) a more detailed description of the applied translation schema from ATL to Java, (ii) the inclusion of an additional measure, namely number of words, for comparison, and (iii) the consideration of a larger set of transformations. Furthermore, we (iv) greatly expanded our discussion of overhead introduced by using Java for transformations based on the results from the newly included measure. This includes a more detailed inspection of Java code as well as a direct comparison between Java and ATL. Additionally, based on all the results and our own experiences, we (v) are now able to discuss more explicitly what newer Java versions improve over older ones and where the language is still lacking compared to ATL. Finally, we (vi) present a description of scenarios where these advancements are enough to justify Java over ATL and (vii) consider other features of model transformation languages not present in ATL and their impact on the suitability of general purpose languages.

The remainder of this paper is structured as follows: First, Section 6.2 introduces the relevant aspects of ATL as well as the relevant differences between Java 5 and Java 14. Afterwards, in Section 6.3, we give an overview of how we translate ATL transformations to Java. Because the

```
1 module NAME
2 create OUT1:MetaModelB, ...
3 [from|refining] IN1:MetaModelA, ...
4
5 [uses LIBRARY]*
6 [RULEDEF|HELPERDEF]*
```

List. 6.1: Structure of an ATL module.

discussions for **RQ2&3** require a precise classification of how code segments in Java are associated to the different transformation aspects, we provide an explanation for this in Section 6.4. In Section 6.5, we present our detailed method for analysing the size and complexity of the translated transformations. The results of our analysis and extensive comparison between the different transformation approaches are then presented in Section 6.6. Based on these results, Section 6.7 discusses our take-aways for what newer Java versions improve over older ones, where the language did not advance, and when these advancements are enough to justify Java over ATL. Section 6.8 then discusses potential threats to the validity of our work, while related work is considered in Section 6.9. Lastly, Section 6.10 concludes the paper and presents potential avenues for future research.

6.2 Background

In this section, we briefly introduce the relevant background knowledge required for this paper. First, since model transformations can only be specified precisely based on some concrete model representation, we introduce the structural representation of models in MDE which is typically assumed by all mainstream model transformation languages, including ATL. Afterwards, since our work builds on ATL as well as the technological advancement of Java, it is necessary to introduce the relevant background knowledge on ATL and to present the important differences between Java SE5 and Java SE14, respectively.

6.2.1 Models in MDE

In MDE, the conceptual model elements of a modelling language are typically defined by a meta-model. The Eclipse Modeling Framework (EMF) (Steinberg et al. 2008), a Java-based reference implementation of OMG's Essential Meta Object Facility (EMOF) (OMG 2002), has evolved into a de-facto standard technology to define meta-models that prescribe the valid structures that instance models of the defined modeling language may exhibit. It follows an object-oriented approach in which model elements and their structural relationships are represented by objects (EObjects) and references whose types are defined by classes (EClasses) and associations (EReferences), respectively. Local properties of model elements are represented and defined by object attributes (EAttributes). A specific kind of references are containments. In a valid EMF model, each object must not have more than one container and cycles of containments must not occur. Typically, an EMF model has a dedicated root object that contains all other objects of the model directly or transitively.

6.2.2 ATL

ATL distinguishes among three kinds of so-called *Units*, being either a *module*, a *library* or a *query*. Depending on the type of unit, they consist of *rules*, *helpers* and *attributes*. For data types and expressions, ATL uses the Object Constraint Language (OCL) (OMG 2014).

6.2.2.1 Units

As illustrated in Listing 6.1, transformations are defined in *Modules*, taking a set of input models (line 3) which are transformed to a set of output models (line 2) by *rule* and *helper* definitions which make up the transformation (line 6).

Libraries do not define transformations but only consist of a set of helper definitions. Libraries can be imported into modules to enhance their functionality (line 5).

```
1 helper [context MODELTYPE]? def : NAME[(PARAMETERS)]? :TYPE  = EXPR;
```

List. 6.2: Syntax to define Helpers.

Queries are special types of libraries, that are used to define transformations from model elements to simple OCL types. They are comprised of a *query* element and a set of *helper* definitions.

6.2.2.2 Helpers and Attributes

Helpers allow outsourcing of expressions that can be called from within rules, similar to simple functions in general purpose languages. Helper definitions can specify a so-called *context* which defines the data type for which the helper is defined as well as parameters passed to the helper. ATL also allows the definition of *attribute* helpers. Attribute helpers differ from helpers in that they do not accept any parameter and always require a context data type. They serve as constants for the specified context. Listing 6.2 shows the syntax to define helpers and attribute helpers.

6.2.2.3 Rules

In ATL, transformations of input models into output models are defined using *rules*. There are two main types of rules: *matched rules* and *called rules*.

Matched rules: The declarative part of an ATL transformation is comprised by matched rules which are automatically executed on all matching input model elements, thus allowing to define type-specific transformations into output model elements. For this, the ATL engine traverses the input model in an optimized order. Furthermore, matched rules generate *traceability* links (trace links for short) between the source and target elements. These links can be navigated throughout the transformation specification to access references to elements created from a source element. Matched rules are comprised of four sections (see Listing 6.3):

- The *In-Pattern* (lines 2 to 3) defines the type of source model elements that are to be matched and transformed. An optional filter expression allows the definition of a condition that must be met for the rule to be applied.
- An optional *Using-Block* (lines 4 to 6) allows to define local variables based on the input element.
- The *Out-Pattern* (lines 7 to 10) then defines a number of output model elements that are to be created from the input element when the rule is applied. Each output model element is defined using a set of so-called *bindings* for assigning values to attributes of the output model element.
- Lastly, an optional *Action-Block* (lines 11 to 13) can be defined which allows the specification of imperative code that is executed once the target elements have been created.

Matched rules can also be defined as *lazy* rules by adding the keyword *lazy* to the rule definition (line 1). In contrast to regular matched rules, lazy rules are only executed when explicitly called for a specific model element that matches both the rule's type and its filter expression. They can be called multiple times on the same model element to produce multiple distinct output elements. To change the behaviour of lazy rules to always produce one and the same output element for the same source model element, lazy rules can be declared as *unique* (line 1).

Called rules: As opposed to matched rules, called rules enable an explicit generation of target model elements in an imperative way. Called rules can be called from within the imperative code defined in the *Action-Block* of rules. They are defined similarly to matched rules. The main difference is that they do not contain an *In-Pattern* but instead allow the definition of required parameters. These parameters can then be used in the *Out-Pattern* and *Action-Block* to produce output model elements.

6.2.2.4 Refining Mode

The refining mode is a special execution mode for ATL modules which aims at supporting an easy definition of in-place transformations (Czarnecki et al. 2006; Strüber et al. 2017). Normally, the

```
[lazy| unique lazy]? rule NAME {
  from
  INVAR : MODELATYPE [(CONDITION)]*
  [using {
    [VAR : VARTYPE = EXPR;]+
  }]?
  to
  [OUTVAR : MODELBTYPE {
    [ATR <- EXPR,]+
  },]+
  [do {
    [STATEMENT;]*
  }]?
}
```

List. 6.3: Syntax to define matched rules.

```
public interface Function<T,R> {
  public R apply(T par);
}
```

List. 6.4: Definition of the Function interface.

ATL engine only creates new output model elements from input model elements matched by the rules defined in a module. However, in the refining mode, the ATL engine instead executes all rules on matching input elements and produces a copy of all unmatched input elements automatically. This aims to allow developers to focus solely on local modifications such as model refactorings rather than also having to manually produce copies of all other model elements.

6.2.3 Technological advancements in Java SE14 compared to Java SE5

Since the release of J2SE 5 in September of 2004, there have been a lot of improvements made to the Java language. In this section, however, we will only cover the ones relevant in the context of this paper. All the relevant features relate to a more functional programming style as they allow developers to express some key aspects of a transformation specification more concisely.

6.2.3.1 Functional Interfaces

With the introduction of the *functional interfaces* in Java SE8, Java made an important step towards embracing the functional programming paradigm, paving the way to define lambda expressions in arbitrary Java code. Lambda expressions, also called anonymous functions, are functions that are defined without being bound to an identifier. This allows developers to pass them as arguments.

In essence, a *functional interface* is an interface containing only a single abstract method. One example of this is the interface called `Function<T,R>` (see Listing 6.4). It represents a function which takes a single parameter and returns a value. This abstract method can then be implemented by means of a Java lambda expression (see Listing 6.5).

Lambdas defined with the interface `Function<T,R>` as their type are then nothing more than objects with their definition as the implementation of the apply method wrapped in a more functional syntax (see Listing 6.5).

Java provides a number of predefined functional interfaces, such as the aforementioned `Function<T,R>`, or `Consumer<T>` which takes one argument and has void as its return value.

6.2.3.2 Streams

Streams represent a sequence of elements and allow a number of different operations to be performed on the elements within the sequence. Stream operations can either be intermediate or terminal. This means that the operations can either produce another stream as their result or a non-stream result which therefore terminates the computation on the stream. This also means that intermediate

```
Function<Integer, Integer> doubleIt = (value) -> value * 2;
```

List. 6.5: Lambda expression definition based on Function.

```
List<String> myList = Arrays.asList(1,2,3,4,5,6);
myList.stream().filter(i -> i % 2 == 0).forEach( System.out::println);
```

List. 6.6: Finding and printing all even numbers in a list.

operations work with all elements within the stream without the developer having to define a loop over it.

The example in Listing 6.6 shows how one can find and print all even numbers in a list using streams.

6.3 Translation Schema

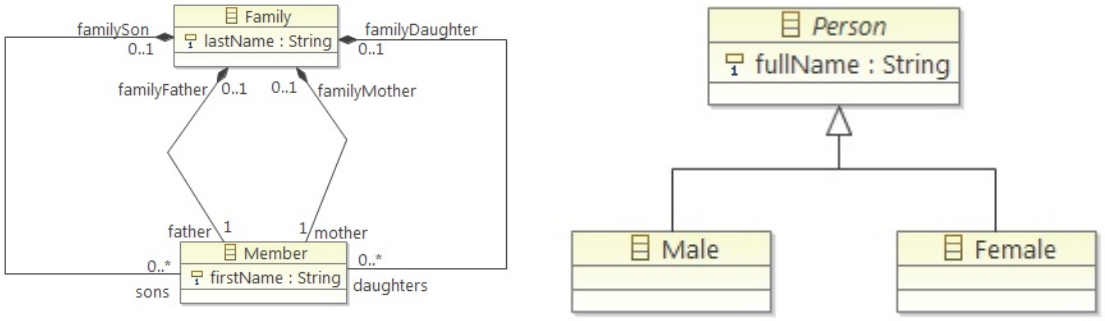
In the following, we will present a detailed description of, first, how the translation schema was developed (see Section 6.3.1), before then describing the translation schema itself (Sections 6.3.2 to 6.3.6).

The description of the translation schema is split into five parts. In Section 6.3.2, we describe the general setup used to emulate ATL semantics in Java and the basic structure that all translated modules follow. Then, in Section 6.3.3, we introduce and describe three libraries to reduce repetitive code between translated modules, one for trace handling, one for model traversal, and one for model loading and persisting. Sections 6.3.4 and 6.3.5 describe how the essential building blocks, namely matched rules and called rules, of ATL transformations are translated into Java. And lastly, in Section 6.3.6 we explain how helpers and general OCL expressions are translated.

All descriptions are illustrated by the use of a running example. For this, we use an ATL solution found in the ATL Zoo for the Families2Persons case from the TTC'17 (Anjorin et al. 2017) the code of which can be found in Listing 6.7 while its Java SE14 counterpart can be found in Listing 6.8. The meta-models for the transformation case are shown in Figure 6.1. The example illustrates how different ATL elements are translated into their corresponding Java code based on the described schemata. Our descriptions will focus on the Java SE14 translation schemata. Notable differences between the Java SE14 and Java SE5 translation schemata are highlighted as such.

6.3.1 Schema Development

To develop the translation schema, we followed the design science research methodology (Wieringa 2014). We used the ATL solution found in the ATL Zoo for the Families2Persons case from the



(A) The Families meta-model. (B) The Persons meta-model.

Figure 6.1: The Families and Persons meta-models from the Families2Persons case taken from Jouault (2013).

TTC'17 (Anjorin et al. 2017) as our initial test input for the translation scheme and focused on developing the schema for Java SE14.

The development process followed a simple, iterative pattern. A translation schema was developed by the main author and applied to the Families2Persons case. The resulting transformation was then reviewed by one co-author, focusing on completeness and meaningfulness. Afterwards, the results of the review were used as input for reiterating the process.

In a final evolution step, the preliminary transformation schema was applied to all 12 selected ATL transformations. Afterwards, both co-authors reviewed the resulting transformations separately based on a predefined code review protocol. In a joint meeting, the results of the reviews were discussed and final adjustments to the transformation schema were decided. These were then used to create a final translation of all 12 ATL transformations.

Lastly we ported the developed transformations to Java SE5 by forking the project, reducing the compiler compliance level and re-implementing the parts that were not compatible with older compiler versions.

To validate the correctness of the translated transformations, we used the input and output models that were provided within the ATL transformation projects. The input models were used as input for the Java transformations and the output models were then compared with the output of the transformations. Since neither an input nor an output model was available for the R2ML2XML transformation, we had to rely solely on the results of our code reviews for its validation. This validation approach is similar to how Sanchez Cuadrado et al. (2020) validate their generated code.

Our translation schema allows us to translate any ATL module into corresponding Java code. The only assumption we make is that all the meta-models of input and output models are explicitly available. The reason for this is that we work with EMF models in so-called static mode, which means that all model element types defined by a meta-model are translated into corresponding Java classes using the EMF built-in code generator.

6.3.2 General Setup and Module Translation

In our translation scheme, we generally assume that each model contains a single root element. This is standard for EMF but could be easily extended by using lists as input and output.

An ATL module is represented by a Java class which contains a single point of entry method that takes the root element of the input model as its input and returns the root element of the output model. The `transform` method in line *13* of Listing 6.8 represents this entry point for the Families2Persons transformation. It takes the *root* model element of type `Family` from the input model and returns a List of type `Person` which serves as the root element for the `Persons` meta-model[4].

Additionally, some setup code is needed for extracting a model and its root element from a given source file, calling the entry point of the actual transformation class, and serializing the resulting output model. The code required for our running example is shown in Listing 6.9. We utilize one of our developed libraries, namely `IO`, for reading an xmi-file containing a `Families` model, extracting the root object of type `Family` and passing it to the `transform` method of the `Families2Persons` class to initiate the actual transformation. The resulting output of type `List<Person>` is then written to an xmi-file, again, utilizing our `IO` library.

Because traceability links need to be created before they can be used, we split the transformation process into two separate runs. The first run creates all target elements as well as all traceability links between them and their source elements, while the second run can safely traverse over model references and populate the created elements by utilizing the traceability links when needed. Consequently, the corresponding Java transformation class comprises two separate methods, dedicated to each run and being called by the entry point method. In our example in Listing 6.8, the methods `preTransform(Family root)` and `actualTransform(Family root)` in lines *18* and *25* represent these two runs. Their implementation will be explained later throughout Section 6.3.4.

[4]In reality the `Persons` meta-model does not have a root element and the list is used as a substitute for the transformation to conform with the translation schema as well as general EMF standards. To produce this list from the transformed elements the `Family2List` method in lines *36-42* is introduced which does not have a counterpart in ATL.

```
module Families2Persons;
create OUT : Persons from IN : Families;

helper context Families!Member def: familyName : String =
  if not self.familyFather.oclIsUndefined() then
    self.familyFather.lastName
  else
    if not self.familyMother.oclIsUndefined() then
      self.familyMother.lastName
    else
      if not self.familySon.oclIsUndefined() then
        self.familySon.lastName
      else
        self.familyDaughter.lastName
      endif
    endif
  endif;

helper context Families!Member def: isFemale() : Boolean =
  if not self.familyMother.oclIsUndefined() then
    true
  else
    if not self.familyDaughter.oclIsUndefined() then
      true
    else
      false
    endif
  endif;

rule Member2Male {
  from
    s : Families!Member (not s.isFemale())
  to
    t : Persons!Male (
  fullName <- s.firstName + ' ' + s.familyName
  )
}

rule Member2Female {
  from
    s : Families!Member (s.isFemale())
  to
    t : Persons!Female (
  fullName <- s.firstName + ' ' + s.familyName
  )
}
```

List. 6.7: Families2Persons ATL solution.

```java
public class Families2Persons {
  private static final PersonsFactory PERSONSFACTORY = PersonsFactory.eINSTANCE;
  private static final Tracer TRACER = new Tracer();
  private static boolean isFemale(Member member) {
    return member.getFamilyDaughter() != null || member.getFamilyMother() != null;
  }
  private static String familyName(Member member) {
    return ((Family) member.eContainer()).getLastName();
  }
  public static List<Person> transform(Family family) {
    preTransform(family);
    return actualTransform(family);
  }
  private static void preTransform(Family root) {
    var iterator = root.eAllContents();
    var traverser = new Traverser(TRACER);
    traverser.addFunction(Member.class, x -> {Member2MalePre((Member)
    x);Member2FemalePre((Member) x);});
    traverser.traverseAndAcceptPre(iterator);
  }
  private static List<Person> actualTransform(Family root) {
    var newRoot = Family2List(root);

    var iterator = root.eAllContents();
    var traverser = new Traverser(TRACER);
    traverser.addFunction(Member.class, x -> {Member2Male((Member)
    x);Member2Female((Member) x);});
    traverser.traverseAndAccept(iterator);

    return newRoot;
  }
  private static List<Person> Family2List(Family root) {
    var persons = new LinkedList<Person>();
    persons.add(TRACER.resolve(root.getFather(), Male.class));
    persons.add(TRACER.resolve(root.getMother(), Female.class));
    persons.addAll(root.getDaughters().stream().map($ -> TRACER.resolve($,
    Female.class)).collect(Collectors.toList()));
    persons.addAll(root.getSons().stream().map($ -> TRACER.resolve($,
    Male.class)).collect(Collectors.toList()));
    return persons;
  }
  private static void Member2MalePre(Member m) {
    if (!isFemale(m)) {
      TRACER.addTrace(m, PERSONSFACTORY.createMale());
    }
  }
  private static void Member2Male(Member m) {
    var t = TRACER.resolve(m, PERSONSFACTORY.createMale());
    t.setFullName(m.getFirstName() + " " + familyName(m));
  }
  private static void Member2FemalePre(Member m) {
    if (isFemale(m)) {
      TRACER.addTrace(m, PERSONSFACTORY.createFemale());
    }
  }
  private static void Member2Female(Member m) {
    var t = TRACER.resolve(m, PERSONSFACTORY.createFemale());
    t.setFullName(m.getFirstName() + " " + familyName(m));
  }
}
```

List. 6.8: The Families2Persons solution translated in Java SE14.

```
1 List<EObject> ins  = IO.readModel("Family.xmi");
2 Family family = (Family) ins.get(0);
3 List<Person> persons = Families2Persons.transform(family);
4 IO.persistModel(persons, "persons.xmi");
```

List. 6.9: Setup code for the Families2Persons transformation.

6.3.3 Libraries

For both model traversal as well as trace generation and resolving, we developed generic libraries which can be reused across all transformation classes. Additionally, we also required a library to outsource the reading and writing of models from and into files. The remainder of this section will describe these libraries in more detail.

6.3.3.1 IO Library

The IO library contains methods used for reading and writing models from and to files. The library exposes two methods, namely `readModel(String uri)` and `persistModel(EObject root, String uri)` which both bundle together several EMF and file-IO methods to achieve the desired effects. To do so the library utilizes the `Resource`[5] type which represents a "persisted document" in EMF and allows to read and write `EObjects` from and to it. To be able to read and write different file types such as *xmi* or *ecore*, a corresponding `ResourceFactory` needs to be registered in the *ExtensionToFactoryMap* of the ResourceFacotry registry. For this reason, we opted to only support *xmi*, *ecore* and *uml* files since EMF provides default ResourceFactory implementations for all three.

The `persistModel` method takes a root element of a model as well as a desired output path, and creates a resource containing the root element (and all its children) which is then saved to the specified path. The `readModels` method reverses this approach by extracting the resource pointed to by the passed path and returning all contents of the referenced resource to the caller. Due to the makeup of EMF compliant files such as *xmi*, *ecore* or *uml* the first element within the contents will then always contain the root element of the model within the file which can then be used as seen in Listing 6.9.

6.3.3.2 Traversal Library

The traversal library allows us to outsource the traversal of the source model and thus reduce the amount of boilerplate code written for each translated transformation. It builds upon a `HashMap` that maps a `Class<?>` to a `Consumer<EObject>`. The `Consumer<EObject>` interface represents a function that takes an input object of type `EObject` and has a return type of `void`. During traversal, which is encapsulated within the library, the Consumer function that corresponds to an `EObject` can be retrieved from the `HashMap` by using the class of the EObject as key. To achieve this, the library exposes the methods `addFunction` and `traverseAndAccept`.

The `addFunction` method allows us to add a key-value-pair to the encapsulated hashmap. The `traverseAndAccept` method then takes an `Iterable` collection containing `EObjects`, iterates over all contained objects, fetches the function that corresponds to the concrete class of the `EObject` and executes it. This way, we only have to write code that adds the required key-value-pairs to the traverser, while the code for traversing the input model as well as resolving the correct function which is to be called is completely outsourced. Note that adding such function calls is only necessary for *matched rules* since *lazy* and *called rules* are called within the transformation code and not automatically executed based on element type matching. An example of how the traversal library is used can be found in lines *19-22* and *28-31* of Listing 6.8 and will be explained in more detail in Section 6.3.4.

For the Java SE5 solution we decided on an alternative solution using the conditional dispatcher pattern instead of outsourcing the traversal. The reason for this was a weighing of alternatives. Outsourcing the traversal in Java SE5 would require the utilisation of anonymous classes. This in turn would offer a similar work flow and an equal McCabe complexity for defining model traversal as with the functional interface solution in Java SE14. It would however significantly increase the

[5] `https://download.eclipse.org/modeling/emf/emf/javadoc/2.4.3/org/eclipse/emf/ecore/resource/Resource.html`

required number of words and lines of code compared to the conditional dispatcher solution. Only with the improved syntax provided through the functional interfaces in Java SE8 could a decrease of the McCabe complexity be accompanied with an uniform word count and lines of code. Overall, the decision leads to an increase in the McCabe complexity of the traversal code in Java SE5 but allows for word count and LOC to remain stagnant. We will come back and discuss the impact of this decision (in the relevant parts of our results discussion| in Section 6.7.1) later on.

This design decision affects the methods `preTransform` and `actualTransform`. Their implementation in Java SE5 is shown in Listing 6.10. Instead of populating the traverser objects we instead manually iterate over the whole model and decide which methods to call based on the type of the currently visited object.

```
1  //...
2  private static void preTransform(Family root) {
3    TreeIterator<EObject> iterator = root.eAllContents();
4    while (iterator.hasNext()) {
5      EObject next = iterator.next();
6      if (next instanceof Member) {
7        Member m = (Member) next;
8        Member2MalePre(m);
9        Member2FemalePre(m);
10     }
11   }
12 }
13
14 private static List<Person> actualTransform(Family root) {
15   List<Person> newRoot = Family2List(root);
16
17   TreeIterator<EObject> iterator = root.eAllContents();
18   while (iterator.hasNext()) {
19     EObject next = iterator.next();
20     if (next instanceof Member) {
21       Member m = (Member) next;
22       Member2Male(m);
23       Member2Female(m);
24     }
25   }
26   return newRoot;
27 }
28 //...
```

List. 6.10: Translated model traversal in Java SE5.

6.3.3.3 Trace Library

The trace library emulates the management of traceability links of ATL. Similar to the traversal library, the trace library is built based on a `HashMap`. In this case, however, the `HashMap` maps source `EObject`s to target `EObject`s and thus can be used both in Java SE5 and Java SE14.

In essence, the trace library exposes two methods. First, for adding a trace (`addTrace`), thus requiring the source and target objects to be passed as parameters. Second, for resolving a trace based on a source object named `resolve`. To achieve type consistency `resolve` also requires the class of the intended target object to be passed as parameter. An example of how the trace library is used can be found in line *51* of Listing 6.8 and will be explained in more detail in Section 6.3.4.

For more advanced trace management, additional methods exist that take an additional String parameter to be able to add and distinguish multiple target objects for a single source object. This functionality is sometimes required to access not the direct target object but another object that was created during the translation of a source object.

6.3.4 Matched Rule Translation

Matched rules are translated into two methods within the transformation class. One method is responsible for creating a target object and its corresponding trace link, and one method is responsible

for populating the created target object in accordance with the bindings in its corresponding ATL rule. The second method will also incorporate all code corresponding to the imperative code written in the *Action-Block* of the translated rule. As already indicated in Section 6.3.2 when introducing our two-step transformation process, the main idea behind this separation is that all traces and referenced objects can be safely resolved by the second method (called during the second traversal) because they are created by the first method (called during the first traversal). That is, calls for the object and trace creation are put by the `preTransform` method, while calls for the second method are put into the body of the `actualTransform` method.

For the rules `Member2Male` and `Member2Female`, this is illustrated in lines *45* and *50* of Listing 6.8. The rule `Member2Male` from Listing 6.7 is translated into the methods `Member2MalePre` (in line *45* of Listing 6.8) and `Member2Male` (in line *50* of Listing 6.8). `Member2MalePre` creates an empty `Male` object as well as a trace from the input `Member`, and method `Member2Male` fills the corresponding `Male` object with data as defined through the bindings from the `ATL` rule. To actually perform the transformation on all `Member` objects, the methods `preTransform` and `actualTransform` define for which type of object which method should be executed. This is done using methods from the traversal library to add the corresponding function calls for the `Member` class as shown in lines *21* and *30* of Listing 6.8.

A special feature that comes from using our traversal library is that we only need to translate the condition whether a rule should be applied in the pre method that is translated from it. This is because the `traverseAndAccept` method only executes the corresponding function for an object after it verified that an associated target object can be found via a trace. If no target object can be found, the function is not executed. An example of this can be found in the translation of the `Member2Male` rule. Line *32* of Listing 6.7 states that `Member2Male` is only executed under the condition that `not s.isFemale()`. In the Java code in Listing 6.8, this is only translated into the `Member2MalePre` method in line *46*, whereas `Member2Male` in line *50* does not contain this condition.

Lazy rules and unique lazy rules do not require as much overhead as matched rules since they are called directly from within other rules/methods and thus do not need to be integrated into the traversal order. However, they do require traces to be created and added to the global tracer. Additionally, methods translated from these types of rules have the target object as their return value rather than the return type being `void`. Suppose `Member2Female` was a lazy matched rule. In that case, instead of the code in lines *21*, *30* and *59-63* for `Member2Female`, only the code shown in Listing 6.11 would be added to the `Families2Persons` class. The method `lazyMember2Female` returns an object of type `Female` while also creating a trace from the passed `Member` to the returned `Female`. In case `Member2Female` was a unique lazy matched rule, a precondition using trace links is added to the translated Java code that ensures that the method always returns the same object when called for the same input object. This is illustrated in Listing 6.12.

```
1 private static Female lazyMember2Female(Member m) {
2   if (isFemale(m)) {
3     Female t = TRACER.add(m, PERSONSFACTORY.createFemale());
4     t.setFullName(m.getFirstName() + " " + familyName(m));
5     return t;
6   }
7   return null;
8 }
```

List. 6.11: Example translated lazy rule.

6.3.5 Called Rule Translation

Called rules, much like lazy rules, can be translated into a single method that creates the output object, populates it in accordance with the bindings of the ATL rule and then returns it. Other than the methods created for matched rules, the methods for called rules can take more than one parameter as input since called rules in ATL can define an arbitrary amount of parameters of varying types. Moreover, called rules do not create or use trace links. A sample called rule translated into Java can be found in Listings 6.13 and 6.14.

```
1 rule calledMember2Female(Member m, String name) {
2   to
3     t : Female (
```

```
private static B uniqueLazyMember2Female(A a) {
  Female t = TRACER.resolve(m, PERSONSFACTORY.createFemale());
  if (t == null) {
    if (isFemale(m)) {
      t.setFullName(m.getFirstName() + " " + familyName(m));
      return t;
    }
    return null;
  }
  return t;
}
```

List. 6.12: Example translated unique lazy rule.

```
      fullName <- name
    )
}
```

List. 6.13: Example ATL called rule.

```
private static Female calledMember2Female(Member m, String name) {
  Female t = PERSONSFACTORY.createFemale();
  t.setFullName(name);
  return t;
}
```

List. 6.14: Example translated called rule.

6.3.6 Helper and OCL Expression Translation

Helpers can be translated into methods much like called rules. The contained OCL expressions can easily be translated into semantically equivalent Java code. Examples of such semantically equivalent translations can be found in lines *9-11* of Listing 6.8 which correspond to the OCL code in lines *4-17* of Listing 6.7. One distinction that can be made here is again between the different Java versions used in terms of our study. Streams can be used to simulate the syntax of OCL, in particular the arrow symbol for implicitly navigating over collections, while older Java versions need to use loops instead. Table 6.2 shows a number of OCL expressions and their Java SE14 counterpart using streams. Note that in contrast to OCL, Java requires all collections to be converted to streams and back to be able to manage them in a functional programming style. The same expressions written in Java SE5 without streams can be found in Listings E.1 to E.6 in Appendix E.1.

Table 6.2: A selection of OCL expressions translated to Java SE14.

OCL	Java SE14
`collection->select(e)`	`collection.stream().filter(e).collect(Collectors.toCollection())`
`collection->collect(e)`	`collection.stream().map(x -> e.apply(x)).collect( Collectors.toCollection())`
`collection->includes(x)`	`collection.stream().anyMatch(a -> x == a).collect( Collectors.toCollection())`
`element.attribute`	`element.getAttribute()`
`collection.attribute`	`collection.stream().map(x -> x.getAttribute()).collect( Collectors.toCollection())`
`i | i > 5`	`i -> i > 5`

6.4 Code Classification Schema

In this section we introduce the classifications of Java and ATL code used throughout **RQ2** and **RQ3**. The ATL classification described in Section 6.4.1 is taken from Götz et al. (2020) and is based

on the hierarchical structure of ATL. The classification of Java code described in Section 6.4.2 was developed specifically for the analysis of this research. It is based in the structure of Java code and its components as well as the relation thereof to general transformation aspects and ATL. We will again use the Families2Persons example to illustrate how the classification schemas are applied.

6.4.1 ATL

The hierarchy for the ATL classification was already established by Götz et al. (2020) and consists of the following levels and their corresponding categories:

1. Module Level
2. Rule Type & Helper Level
3. Rule Blocks Level
4. Content Level
5. Binding Level

The aim of this classification system is to differentiate the different components and their contained subcomponents within an ATL module. As such, this classification represents a way to indicate how a syntax element is contained within the complete structure of the ATL code. This allows us to make precise observations on the structure of ATL modules based on their components and, for example, the distribution of number of words required to write each component. An overview of the classification hierarchy can be found in Figure 6.2. And the complete labelling for the ATL solution of Families2Persons can be found in Figure 6.3.

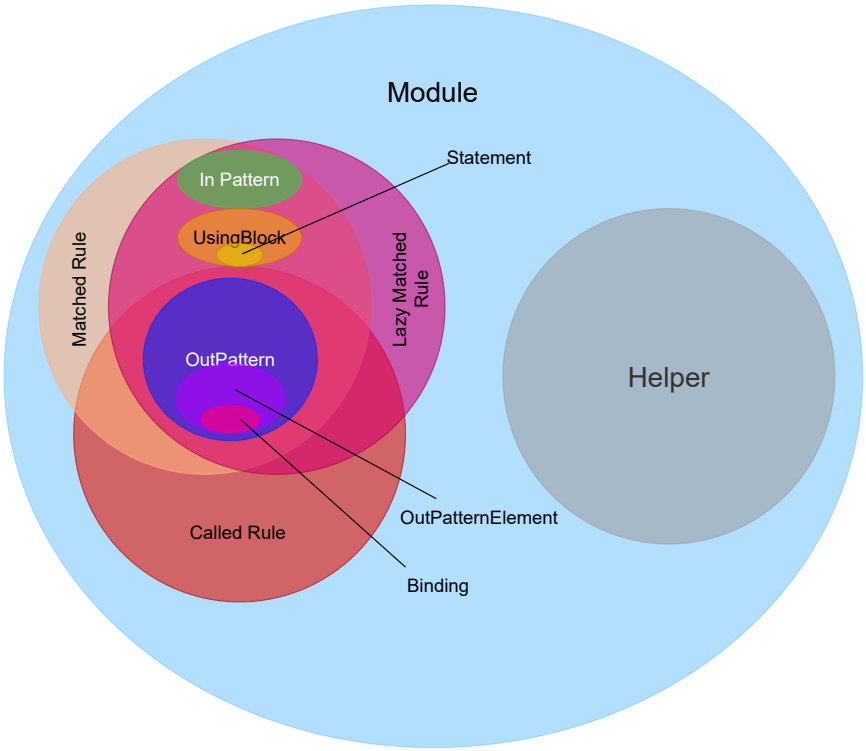


Figure 6.2: Overview of the ATL classification from Götz et al. (2020).

The **Module Level** defines the belonging of all elements within a module to said module. Below it on the **Rule Type & Helper Level** a distinction between helpers and the different types of rules is made. In the Families2Persons example from Listing 6.7 and Fig. 6.3 the helpers in lines *4* & *19* are labelled as *Helper* while both rules `Member2Male` and `Member2Female` in lines *30* & *39* are labelled as *Matched Rule*. All elements within the rules and helpers again inherit the respective classification for this level from their parent elements.

The **Rule Blocks Level** distinguishes between the different types of blocks that make up rules, i.e., *Using Block*, *OutPattern*, *InPattern* and *Action Block*. A more specific distinction of helper contents is not done due to them only containing OCL expressions. The rules in the Families2Persons example only contain InPatterns (lines *31-32, 40-41*) and OutPatterns (lines *33-36, 42-45*).

Below the Rule Blocks Level the **Content Level** then allows a more precise description of the elements contained within the rule blocks. The potential classifications on this level are: *OutPatternElement*, *Statement* and *Variable Declaration*. Lines *43-45* for example are labelled as an *OutPatternElement*.

Lastly, the **Binding Level** again only contains one characteristic and allows to label bindings as exactly that. Lines *35* and *44* are bindings and thus labelled as such as seen in Figure 6.3.

6.4.2 Java

In order to draw parallels between transformation code written in Java and ATL, it is necessary to relate all code components in the Java code to the transformation aspects they implement. For this purpose, we developed a hierarchical classification for Java code. The hierarchy follows the natural structure of Java code much like the classification for ATL. However, contrary to ATL, the code structure of Java does not allow us to directly break it down into transformation-related components. This is due to the fact that Java is focused around object-oriented and imperative components rather than transformation-specific ones. As a result, the classification schema breaks Java code down into its OO and imperative components and then relates those components to transformation aspects. The hierarchy levels of the classification are as follows:

1. Class Level
2. Attribute & Method Level
3. Statement-Type Level
4. ATL Counterpart Level

An overview of the classification levels and the characteristics attributed to each level can be found in Figure 6.4. A sample labelling for the Java solution of Families2Persons can be found in Figure 6.5.

The **Class Level** stands on top of the hierarchy. The class level itself is made up of only one type of characteristic, the *Class* itself. In the Families2Persons example from Listing 6.8 the class definition and all elements contained within the class body is thus labelled as belonging to the class characteristic of the Class Level (as seen in Figure 6.5). This also indirectly represents a relation between the class and the transformation module from which it was translated from, indicating that the class and all its components relate to the transformation module and its components. More specific relation between the contained components is then described through the lower levels within the classification system.

Below the Class Level lies the **Attribute & Method Level** in which we classify to which transformation aspect an attribute or method is related. The characteristics that can be attributed on this level are: *Traversal* when a method is used for the traversal of the input model. *Transformation* when a method contains code for the actual transformation of one model element to another. *Tracing* for all methods that are related to the creation or resolution of traces. *Helper* when a method corresponds to a helper and lastly *Setup* for all attributes that are required to exist for access throughout the transformation. The `isFemale` method in lines *5-7* from Figure 6.5 is thus assigned the label *Helper* for the **Attribute & Method Level** in addition to its *Class* label on the **Class Level**. The `transform`, `preTransform` and `actualTransform` methods all get assigned the *Traversal* label while `Family2List`, `Member2Male` and `Member2Female` are all labelled as *Transformation* related on the Attribute & Method Level. Lastly, `Member2MalePre` and `Member2FemalePre` both relate to *Tracing* and are thus characterized as such. All statements within the methods again inherit the classification of the Class Level and the Attribute & Method level from their respective parents in which they are contained in and get more specialized again through the lower levels within the system.

Below the Attribute & Method Level then lies the **Statement-Type Level** in which all statements within methods are characterized based on whether they are *Control Flow* statements (i.e., conditions or loops), *Variable Declaration*s or any other type of *Statement*. The categorization on

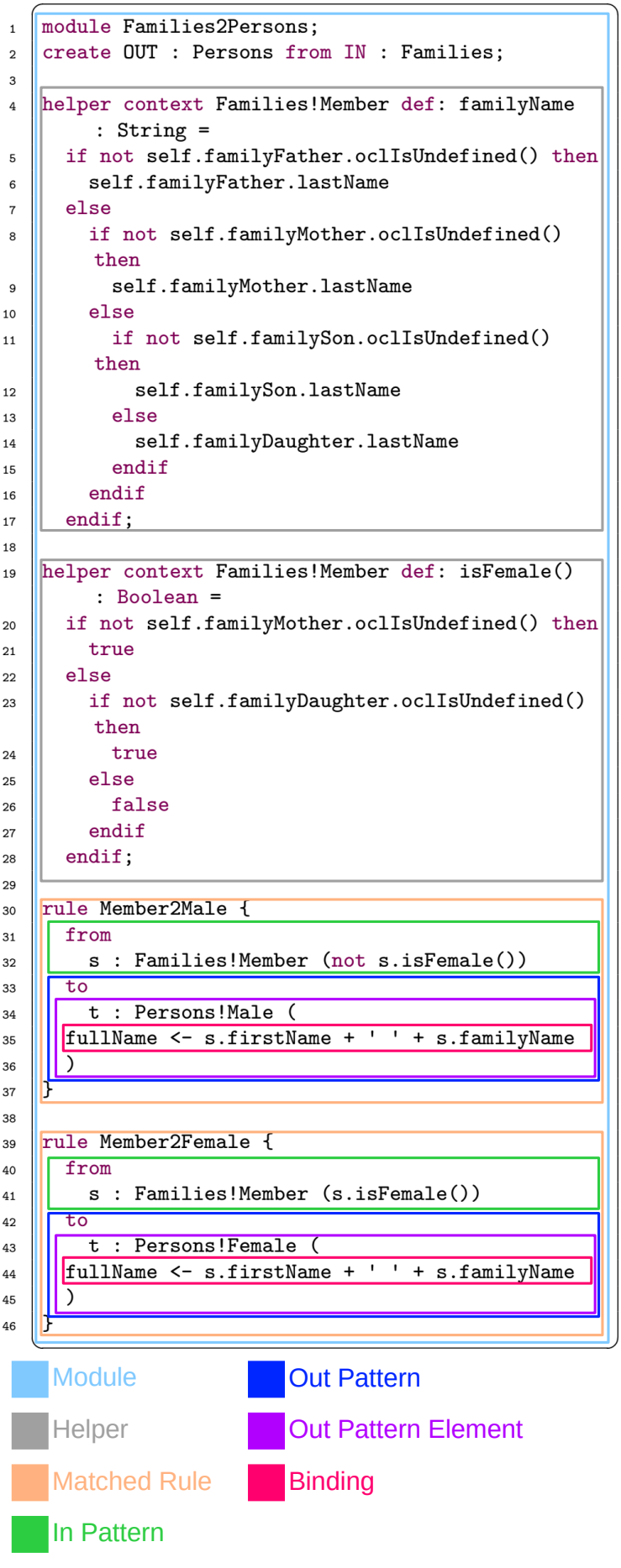


FIGURE 6.3: Labelled ATL solution for the Families2Persons case.

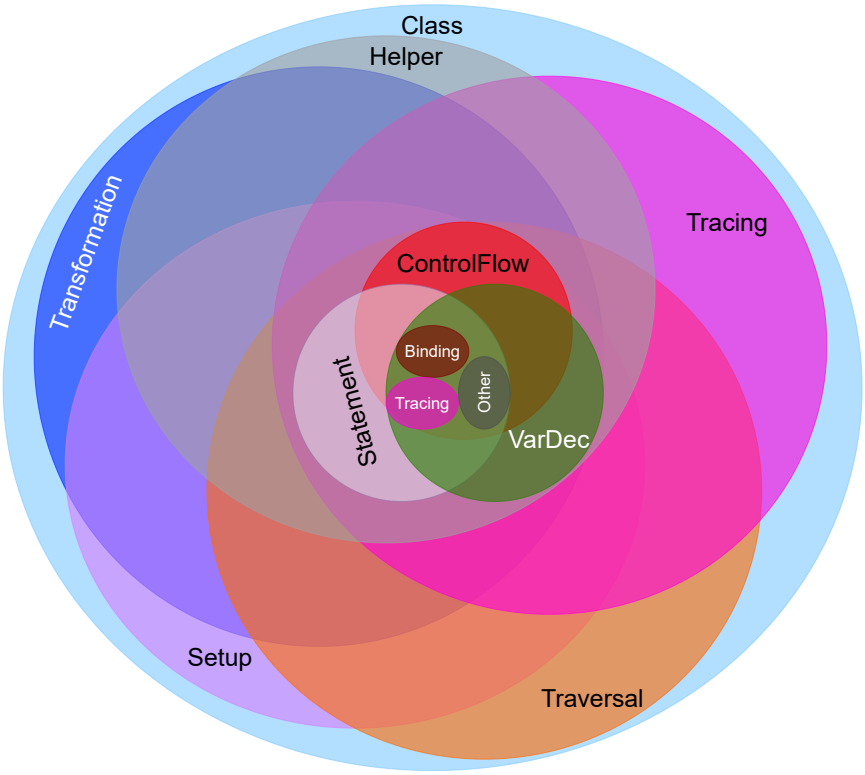


FIGURE 6.4: Overview of the makeup of our Java classification.

this level does not directly relate to any transformation aspect but rather allows us to differentiate between different types of statements in Java that are relevant for highlighting differences between the structure of Java and ATL code. The condition defined in line *46* of Figure 6.5 is labelled as belonging to *Control Flow* on this level while again inheriting its Class Level and Attribute & Method Level from its container Method `Member2MalePre`.

The next lower level is the **ATL counterpart Level**. On this level, we categorize whether a statement fulfills the role of a *Binding* in ATL or if it contains code to create or resolve *Traces* or if it is any *Other* type of Java code that does not directly relate to transformation aspects. At this level, one would expect that the categorization of statements is dependent on the categorization of the **Attribute & Method Level** of the methods they are contained in, i.e., a statement within a *Transformation* method should either be categorized as *Binding* or *Other*. However, in Java transformations these boundaries become somewhat blurred due to the fact that traces need to be explicitly resolved to access the corresponding output model elements when assigning them to output attributes. This can for example be seen for line *51* of Figure 6.5. The classification comes from it being a variable declaration that assigns the result of a trace resolution call within a method that performs the transformation of a `Member` into a `Male`.

Lastly, we can also label different parts of a single line with different labels based on their functionality. Line *38* of Figure 6.5, for example, has elements that perform assignments, i.e., bindings translated to Java, and elements that perform additional tracing operations. The labelling of this line reflects these different functionalities within the line by labelling sub-statements within the line instead of the whole line.

6.5 Size and Complexity Analysis Methodology

Our analysis of the transformation specifications is guided by the research questions introduced in Section 6.1.2.

FIGURE 6.5: Partially labelled Java solution for the Families2Persons case.

6.5.1 RQ1: How much can the complexity and size of transformations written in Java SE14 be improved compared to Java SE5?

To compare the transformations written in Java SE14 and Java SE5, we decided to use code measures focused on code complexity and size. For this reason, we chose McCabe's cyclomatic complexity, and LOC which are shown to correlate with the complexity and size of software (Jabangwe et al. 2015). To keep the LOC count as fair as possible, all Java code was developed by the same researcher and we used the same standard code formatter for all Java code. Furthermore, we supplement LOC with an additional measure for code size based on word count, the combination of these two measures also allowed additional insights. Word count means the number of words that are separated either by whitespaces or other delimiters used in the languages, such as a dot (`.`) and different kinds of parentheses (`()[]{}`). This measure supplements LOC because it is less influenced by code style and independent from keyword and method name size (Anjorin et al. 2019). This method for calculating transformation code size has already been successfully used by Anjorin et al. (2019) to compare several (bidirectional) transformation languages including eMoflon (Weidmann et al. 2019), JTL (Cicchetti et al. 2011), NMF Synchronizations (Hinkel 2016) and their own language BXtend (Buchmann 2018). Their argument for using word count is that because it approximates the number of lexical units it more accurately measures the size of a solution than lines of code.

We applied the Java code metrics calculator (CK) (Aniche 2015) on all 24 transformations (12 Java SE5 + 12 Java SE14) to calculate both metrics and used a program developed by us to calculate the word count measure. For a basic overview we then compare the total size between Java SE5 and Java SE14 based on both LOC and word count and discuss observations as well as possible discrepancies between the two measures. The same is done for McCabe complexity as well. Because CK calculates metrics on the level of *classes*, *methods*, *fields* and *variables* we opted to additionally use the values calculated on the level of *methods*, i.e., the LOC, word count and McCabe complexity of the method bodies, to gain a more detailed understanding of where differences in size and complexity arise from. Since neither the *fields* level nor the *variables* level contained values for McCabe complexity and no interesting values for LOC and word count we decided to omit data from those in our analysis. The metric values calculated by CK were then analysed and compared based on maximum, minimum, median and average values.

RQ1 serves the purpose of providing a general overview of the differences between the code size and complexity between Java SE5 and Java SE14. The results from this research question are analysed and discussed in more detail in **RQ2&3**.

6.5.2 RQ2: How is the complexity of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to each other and ATL?

To answer **RQ2**, we compare the distribution of complexity within the Java code with regards to the different steps within the transformation process. In particular, we want to see how much effort needs to be put into writing those aspects that ATL can abstract away from. To be able to analyse the complexity distribution in Java transformations, it is necessary to differentiate the different steps within the Java code, i.e., *model traversal*, *transformation*, *tracing*, *setup* and *helper*. Since cyclomatic complexity can not be calculated for each line but only for set of instructions we decided to fall back on the granularity of methods and use the classification and labelling given to each method in Section 6.4.

Based on the classification introduced in Section 6.4.2, all Java transformations were labelled by one author. The labelling was verified by the other two authors with one of them cross checking 2 transformations and the other one checking 4. The checked transformations were *istar2archi*, *Palladio2UML* and *R2ML2XML* all in both Java SE5 and Java SE14 which in total meant that about 51% of the total Java code lines were reviewed.

We then used the measures calculated for **RQ1** to create plots of the complexity distribution. The distribution shown in the resulting plots was then analysed taking into account the results of Götz et al. (2020) regarding the distribution of different transformation aspects in ATL. The goal in this step was to see how the complexity in Java transformations is distributed onto transformation aspects, such as tracing and input model traversal, that are abstracted or hidden away in ATL as well as to see the evolution of this distribution between the two different Java versions.

```
rule SimpleBinding {
  from s : Member
  to t : Female (
    name <- s.firstName
  )
}
```

List. 6.15: A rule with a simple binding.

6.5.3 RQ3: How is the size of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to each other and ATL?

The approach for this research question is twofold and follows a top down methodology. First, we compare the distribution of code size within the Java code over the different transformation aspects using the classification from Section 6.4. Afterwards, we focus on the actual code. Here, we compare how code written in ATL compares to the Java code that represents the same aspect within a transformation.

We opted to use word count as a measure for the detailed discussion of code size. The reason why we use word count and not lines of code lies in their granularity. For some parts of our analysis, it is necessary to split the value of single statements up into that of their components. This is much easier to do when using word count as a measure and does not require code to be rewritten in an unintuitive way. Moreover, the finer granularity also allows a more detailed look into the structure of methods that was not possible in **RQ2** due to the limitation of cyclomatic complexity.

The idea behind our approach is to calculate the word count for all transformations written in Java and ATL and then compare both the total count of words as well as the number of words required for specific aspects within the transformation process. While the word count for Java transformations is calculated specifically for this study, the data for the ATL transformations is taken from the results of Götz et al. (2020).

Based on the introduced categorizations, we then create Sankey diagrams for the distribution of word count in both Java and ATL. These graphs then form the basis for our comparison. Here, we compare both the distributions of the individual transformation aspects in Java with ATL as well as the concrete sizes on the basis of the numbers. When comparing the size distribution, we analyse how the distribution of the transformation aspects in Java differs from ATL, i.e., which aspects are disproportionally large or small compared to ATL. We also explicitly look at how much code is required for tracing in Java. For this, we look at the proportion of the transformations that require traces and how that compares to the total size of Java code related to traces. Lastly, the total number of words between Java and ATL are also directly compared to see which language allows for shorter transformation code based on this measure.

To illustrate where the observed effects originate from, we use a selection of three ATL fragments representing code which is often written in ATL transformations. The first fragment (see Listing 6.15) represents code that copies the value of an input attribute to an attribute of the resulting output model element, an action which constitutes 56% of all bindings in the set analysed by (Götz et al. 2020). The second fragment (see Listing 6.16) represents code that requires ATL to use traceability links, which (Götz et al. 2020) found to constitute 15% of all bindings. Because the attribute `s.familyFather` does not contain a primitive data type but a reference to another element within the source model, the contained value cannot simply be copied to the output element. Instead, ATL needs to follow the traceability link created for the referenced input element to find its corresponding output element which can then be referenced in the model element created from `s`. The last code fragment (see Listing 6.17) is a helper definition of average size and complexity.

We use those code fragments and compare them with the Java code that they are translated to in order to highlight differences between the languages.

```
rule Trace {
  from s : Member
  to t : Male (
    father <- s.familyFather
  )
}
```

List. 6.16: A rule with a binding using traces.

```
helper context Class def: associations: Sequence(Association) =
    Association.allInstances() -> select(asso | asso.value = 1);
```

List. 6.17: A typical helper in ATL.

6.5.4 RQ4: How does the size of query aspects of transformations written in Java SE5 & SE14 compare to each other and ATL?

As previously discussed, the goal of this research question is to investigate the claim that writing queries for models was improved with the introduction of model transformation languages such as ATL and to check if this is still the case when utilizing new languages features in general purpose languages today. This discussion of Java vs OCL has already been raised approaches to replace OCL with Java (D. S. Batory et al. 2020). The data basis for this analysis is formed by all helpers and their corresponding Java translations in form of methods within the 12 transformation modules subject in this study. Because this set only contains a total of 15 helpers, we complement it with a large collection of helpers and their translations from a set of supplemental libraries used in the *UML2Measure* transformation.

In our analysis, we compare Java and ATL helpers first based on their total word count and then by contrasting each ATL helper with its Java counterpart using regression analysis. All observations in this analysis are supplemented with code segments that highlight them. The regression analysis uses a linear regression model to predict the word count of Java methods ($J5WC$, $J14WC$) based on the word count of ATL Helpers ($HelperWC$). This was chosen based on an hypothesis that Java code entails an additional fix cost compared to OCL expressions as well as an increase by some factor due to the more verbose syntax of Java. This approach allows us to both verify the hypothesis and identify an approximation of the interrelationship between the code sizes.

6.6 Results

In this section, we present the results of our analysis in accordance with the research questions from Section 6.1.

6.6.1 RQ1: How much can the complexity and size of transformations written in Java SE14 be improved compared to Java SE5?

Table 6.3 presents an overview of lines of code (LOC), word count (# words) and the sum of McCabe complexities of all methods contained in the transformation classes (WMC). Looking at the total lines of code and WMC, the numbers display an expected decrease in both size and complexity. Our transformations written in Java SE5 total 3252 lines of code and have a WMC of 792. The same transformations written in Java SE14 require only 2425 lines of code and have a WMC of 411. Based on these measures, the size reduces by about 25%, while the cyclomatic complexity is cut in half to about 52% of its Java SE5 counterpart. This decreased WMC can be attributed to the improvements made through utilizing streams for handling collections. The traversal library also contributes to this by removing all control flow branching for the `transform` methods and thus reducing the McCabe complexity of these methods.

The word count measure, however, shows a different picture. While the Java SE5 implementation uses 13007 words the Java SE14 implementations use nearly the same amount of words, 13118 to be exact. When combining this with the reduced number of code lines provides and interesting

observation. Transformation code written in Java SE14 for our transformation set is more dense, i.e a single line of code contains a lot more words and thus more information about the transformation.

Table 6.3: Measurement data on the translated transformation modules.

	LOC		# words		WMC	
Transformation Name \ Java Version	SE5	SE14	SE5	SE14	SE5	SE14
ATL2BindingDebugger	22	19	93	88	4	2
ATL2Tracer	74	17	285	283	7	5
DDSM2TOSCA	509	339	2137	2036	103	44
ExtendedPN2ClassicalPN	147	107	569	553	37	19
Families2Persons	72	62	273	297	22	14
istart2archi	184	115	689	714	57	24
Modelodatos2FormHTML	215	178	761	750	58	40
Palladio2UML	303	253	1066	1100	70	47
R2ML2XML	1181	855	4720	4966	303	139
ResourcePN2ResourceM	99	67	380	389	29	13
SimpleClass2RDBMS	163	111	629	581	50	26
UML22Measure	283	249	1405	1356	52	38
Total	3252	2425	13007	13118	792	411
Median	173.5	113	599	647	51	25
Average	271	202.1	1088.9	1092.75	66	34.25

Overall, both the total number of lines of code as well as the WMC of transformations in the newer Java version are greatly reduced. However, there is no notable change in the number of required words, which hints at a more information-dense code rather than simply less code.

Table 6.4 summarises the calculated size (LOC and word count) and complexity (McCabe) measurements on the method level for both the Java SE5 and Java SE14 transformation code.

Table 6.4: Measurement data on the methods in the translated transformation modules.

	Minimum		Median		Average		Maximum	
Measure \ Java Version	SE5	SE14	SE5	SE14	SE5	SE14	SE5	SE14
LOC	3	3	7	6	12.5	9.4	135	105
# words	1	2	5	6	5.2	6.4	64	37
McCabe complexity	1	1	2	1	3	1.6	44	11

As expected from the total numbers, the average and median length, measured in LoC, of methods in Java SE14 is reduced by about 30%. The already low minimum of 3 lines has not been further reduced in the newer version but the longest method is now 51 lines shorter.

Contrasting the numbers for lines of code with word count, we see a small increase in both the average and median method sizes in Java SE14 compared to Java SE5. However, the maximum number of words for a method is about 43% shorter in Java SE14 than in Java SE5. This means that while on average (or median) the number of words required to implement transformation-related methods in Java SE14 increased compared to Java SE5, newer Java versions help to reduce the size of methods that required large number of words in older Java versions.

The reduction in cyclomatic complexity seen in the total numbers is also reflected for the more detailed consideration on method level. The average transformations written in Java SE14 are 45% less complex than in Java SE5. A result also reflected in the median. Furthermore, the maximum McCabe complexity is reduced from 44 to 11, which is a significant decrease as this suggests that even highly complex methods within the transformations can be expressed a lot less complex in newer Java versions. This, again, can be attributed to the utilization of streams and functional

interfaces which help to remove the requirement to manually implement large amounts of loops and nested conditions.

The more detailed results reflect what was already shown on a coarse-grained level. Compared to Java SE5, new language features in Java SE14 help to reduce the required number of code lines, while the number of words stays about the same. The cyclomatic complexity is significantly reduced, most prominently seen in the fact that the most complex method in Java SE14 is only 1/4th of the complexity of the most complex method in Java SE5.

6.6.2 RQ2: How is the complexity of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to ATL?

The results for this research question are split up into two parts. We first report on our findings for Java SE5 and its comparison to ATL in Section 6.6.2.1, before reporting the findings for Java SE14 and its comparison to ATL and Java SE5 in Section 6.6.2.2.

6.6.2.1 Java SE5

Figure 6.6 shows a plot over the distribution of WMC split up into the different transformation aspects involved in a transformation written in Java SE5 and Java SE14. It shows that about 60% of the complexity involved in writing a transformation in Java SE5 stems from the actual code representing the transformations and helpers. The other 40% are distributed among the model traversal, tracing and setup code. In ATL, these three aspects are completely hidden behind ATL's syntax. In other words, this means that 40% of the complexity within the transformations written in Java SE5 stems from overhead code.

Overall, the results support the consensus from back when ATL was introduced that a significant portion of complexity can be avoided when using a dedicated MTL for writing model transformations.

6.6.2.2 Java SE14

Given the observations from **RQ1** combined with the the general improvements that Java SE14 brings to the translation scheme, one would expect better results for the complexity distribution of transformations written in that Java version. However, when looking at Figure 6.6, which again shows a plot over the distribution of McCabe complexity split up into the different transformation aspects involved in a transformation written in Java, there is still a significant portion of complexity associated with the model traversal, tracing and setup code in Java SE14.

While the complexity associated with model traversal is greatly reduced by the use of the traversal library, the overall distribution between the actual code representing the transformations and helpers and the model traversal, tracing and setup code does not change much. About 40% of the overall transformation specification complexity still stems from overhead code. Moreover, not only did this ratio stay similar compared to Java SE5, also the ratio between helper code complexity and transformation code complexity stayed about the same. One potential reason for this is that while newer Java features help to reduce complexity, they do so for all aspects of the transformation, thus the distribution stays about the same.

The reason that the code related to trace management experiences an increase in its complexity ratio compared to other parts of the transformation can be explained by the fact that this code stayed the same between the different Java versions. Thus, while the complexity of all other components shrank, the complexity of trace management methods stayed the same, leading to higher relative complexity.

Overall, the results point towards even newer versions of Java still having to deal with the complexity overhead that ATL is able to hide. Specifically, handling traces still entails a large overhead.

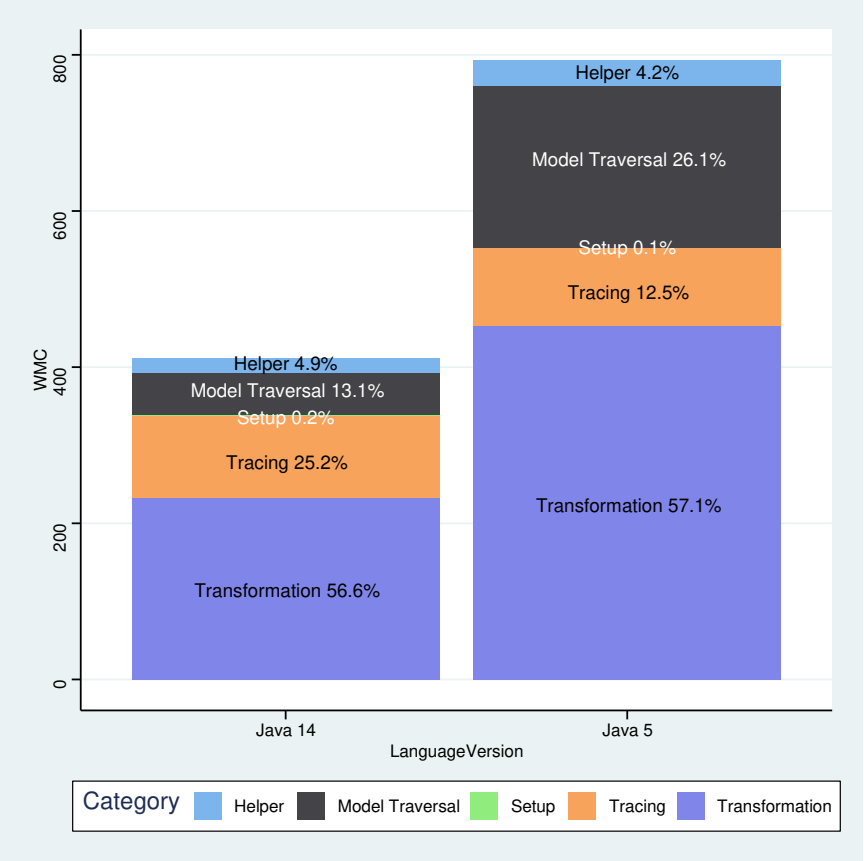


FIGURE 6.6: Distribution of WMC over transformation aspects in Java SE5 and SE14.

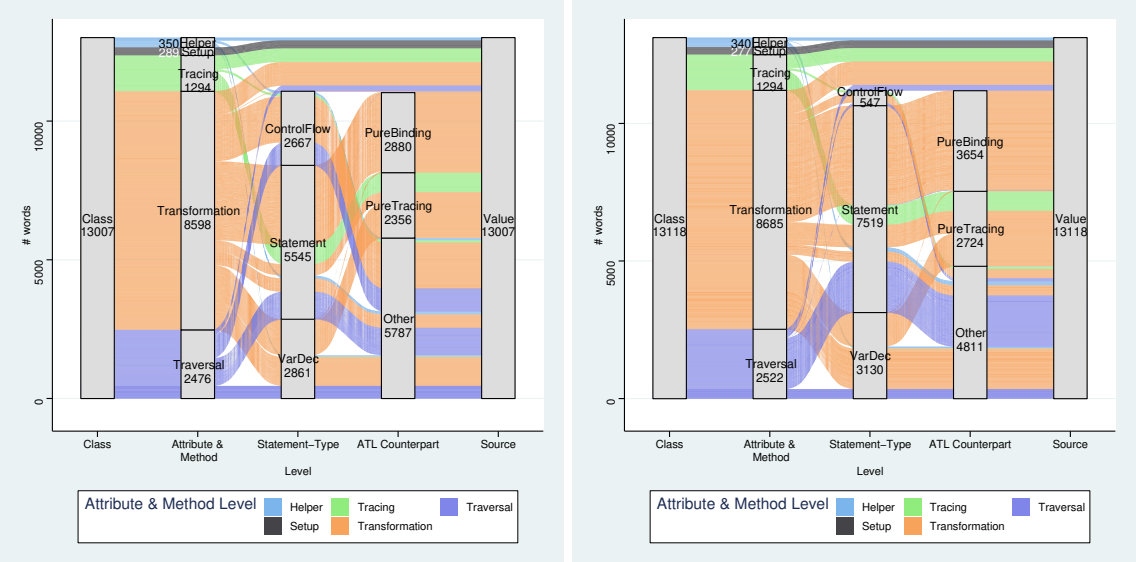
6.6.3 RQ3: How is the size of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to ATL?

The reporting of results for this research question follows the same structure as Section 6.6.2. First in Section 6.6.3.1 the results of our analysis of Java SE5 and its comparison with ATL are reported. Afterwards in Section 6.6.3.2 the results for Java SE14 and its comparison with ATL are discussed. This section also contains a comparison to the results of Java SE5.

6.6.3.1 Java SE5

The total size of Java SE5 transformations compared to ATL transformations is much larger when using word count as a measure. All ATL transformations in our set together amount to 7890 words, while the Java SE5 code needs 13007. This is an increase of 64.8%. Figure 6.7a allows us to look at the distribution of written words over the transformation aspects introduced in Section 6.4. The x-axis of the graph describes the hierarchy-levels from Section 6.4. The word count is depicted on the y-axis and on each hierarchy-level on the x-axis the word count distribution of its different aspects are shown. How each level is made up of its sub-levels is then shown by means of the alluvial lines flowing from left to right. The flow lines are coloured according to the Attribute & Method level as it represents the top level of separation and eases readability.

Looking at the graph we see a large portion of the number of words is actually associated with the transformation code itself. Overhead from tracing, traversal and setup exists but it is not as prevalent as expected from the results presented in Section 6.6.3. However looking more closely into each of the aspects and their makeup reveals that there is more overhead still hidden in the transformation-related code. In the following, we will look at the individual aspects and their more precise breakdown and what this means for transformations written in Java SE5, also in comparison to ATL.



(A) Distribution in Java SE5.

(B) Distribution in Java SE14.

FIGURE 6.7: Distribution of word count over transformation aspects in Java SE5 and SE14.

The number of words required to express Helper code for our transformation set is low. It constitutes 2.9% of all words within the transformation class which is in line with the size of helpers in ATL as seen in Figure 6.8.

Similarly, the number of words required for setup code is also of little consequence as it constitutes only about 2.2% of the total word count in the transformations considered in this work. However, even though the amount is small, the code still has to be written and maintained when evolving the transformation.

Another part of the code within the transformation classes that represents overhead in Java SE5 compared to ATL is the code related to tracing. While ATL abstracts away tracing and does target element creation implicitly, in Java this behaviour has to be recreated by hand. The library for tracing introduced in Section 6.3.3 helps reduce the implied overhead but the creation of target objects as well as traces for them still has to be initiated manually. The methods involved in this constitute for 9.9% of words used in our translated transformations and are made up of methods in style of what is described in Section 6.3.4.

As previously stated, a large portion (65.8%) of the word count comes from methods and attributes related to the actual transformation. This however changes when looking at the lower levels of classification within those methods. In ATL 60% of the total number of words and 61% of the words within rules stem from bindings, i.e., the core part responsible for transforming input into output. In our Java SE5 translation this differs greatly. The translated binding code only makes up 22% of the total word count or 33.5% within the transformation methods. This points to the fact, that much less of what is written in Java SE5 actually relates to actual transformation activities. In Java many more words are spent on code not directly transformation-related but rather on tasks necessary for the transformation to work. Three such types of code stand out.

One is statements that resolve traces built up in the tracing methods discussed in the last section (as seen by the flow from Transformation over Statement and Variable Declarations towards Tracing in Figure 6.7a). Examples of such code in the Families2Persons example from Figure 6.5 and Listing 6.8 can be found in lines *38,39* and *51*.

The second one is code to initialise temporary variables used for processing steps within the transformation (as seen by the flow from Transformation over Variable Declarations into Other in Figure 6.7a).

And lastly there is a large number of words associated with control flow via loops and conditions to process collections in order to bind their transformed contents onto attributes of of the current output object (as seen by the flow from Transformation over Control Flow towards Other in Figure 6.7a).

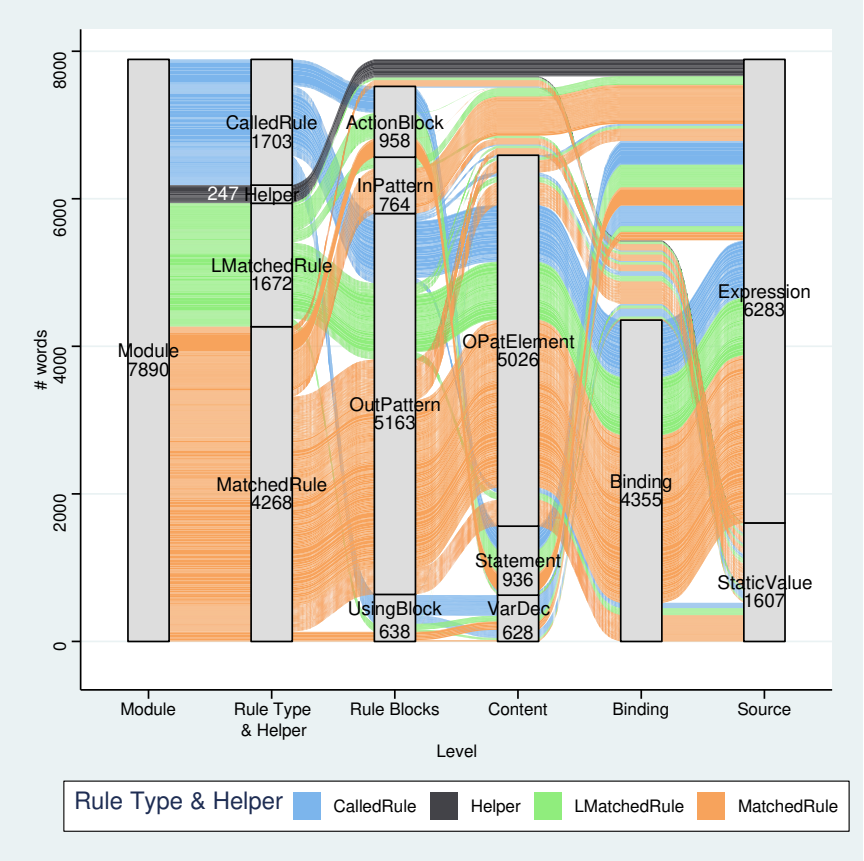


FIGURE 6.8: Distribution of word count "complexity" measure over transformation aspects in ATL calculated based on Götz et al. (2020).

Code relating to traversal is again overhead introduced due to the usage of Java over ATL. The number of words required for writing traversal-related code for our set of transformation constitutes 18.9% of the total word count of transformation classes.

Overall, the overhead produced by Tracing, Traversal and Setup code amounts to 31% of the total number of words for our Java SE5 transformations. Furthermore, while 65.8% of words within the transformation classes are related to the process of transformation, many of them are again overhead from manual trace resolving, model traversal and supplemental code.

```
1 private void simpleBinding(Member s) {
2   ...
3   t.setName(s.getFirstName());
4 }
```

LIST. 6.18: A rule with a simple binding in Java SE5.

When comparing a simple binding (see Listing 6.15) written in ATL with its translation in Java SE5 (see Listing 6.18), there is not much difference. Both require nothing more than their language constructs for accessing attribute values and assigning them to a different attribute.

This is not the case when traces are involved. While ATL allows developers to treat source elements as if they were their translated target element (see Listing 6.16), some explicit code needs to be written in Java (see Listing 6.19). As a result, the transformation specification gets larger since it is not only required to call the trace resolution functionality, but it is also necessary to put some additional type information in so the Java compiler can handle the resulting object correctly. The type information is necessary since, as described in Section 6.3.3.3, the trace library holds `EObjects` which have to be converted to the correct type after they have been retrieved based on the source object.

```
1  private void simpleBinding(Member s) {
2    ...
3    t.setName(TRACER.resolve( s.familyFather, Male.class));
4  }
```

List. 6.19: A rule with a binding using traces in Java SE5.

```
1 private List<Association> associations(Class self) {
2   List<Association> list = new LinkedList<Association>();
3   for (Association asso : ALLASSOCIATIONS) {
4     if (asso.getValue() == 1) {
5       list.add(asso);
6     }
7   }
8   return list;
9 }
```

List. 6.20: A typical helper in Java SE5.

The increase in size is even more prevalent when looking at the translation of a typical helper. The helper in Listing 6.17 requires OCL code that works with collections which, thanks to OCL's "→ syntax", can be expressed in a concise manner. In Java SE5, however, as seen in Listing 6.20, the code gets a lot more complex and bloated. This is due to, as previously stated in Section 6.6.3, the fact that the only way to implement the selection is to iterate over the collection through an explicit loop (lines 3 to 9) and to use an if-condition within the loop (lines 4 to 6). We investigate and discuss this in more detail later in Section 6.6.4.

Overall, the examples show that simple bindings can be expressed easily in both ATL and Java SE5. Bindings involving trace resolution require some additional effort in Java SE5 while ATL can handle those like any other binding. The most significant difference, however, comes from expressions involving collections. Due to the required usage of explicit loops, the Java SE5 code blows up in size and complexity compared to the more compact ATL notation.

6.6.3.2 Java SE14

Comparing the total number of words in Java SE14 transformations with ATL, a similar picture as for Java SE5 arises. The translated transformations require 13118 words while ATL only requires 7890. Surprisingly, as also discussed in Section 6.6.1, the number of words in Java SE14 is higher than that of Java SE5, although only by around 100 words, despite requiring less lines of code and cyclotomic complexity. We believe this to be the result of two effects. One, using streams for processing collections reduces the lines of code and cyclomatic complexity because they are single statements and are thus not split over as many lines as when using loops. But, setting up streams and transforming them back into the original collection requires several additional method calls which offset the overall reduction of number of words.

The distribution of the number of words between Java SE5 and Java SE14 also differs immensely, especially around the make up of transformation methods, as evident from Figure 6.7b. It also again highlights key differences between the ATL transformations and their Java counterparts.

The portion of words required for writing Setup and Helper code has slightly reduced compared to Java SE5 while the proportion of words for Transformation and Traversal methods increased. The Methods & Attributes for setting up helpers does not change which is due to the fact that the underlying code does not change between Java SE5 and Java SE14.

Thus, more can be concluded from how the number of words are distributed within the Transformation and Traversal methods in Java SE14.

For Traversal, it is noticeable that almost no control flow statements are used any more. Instead, most words now come from simple statements. This is because in Java SE14 we make use of the Traversal library, which allows us to pass only the classes to be matched and the methods to be called to the traverser instead of having to write loops and conditions manually. This evidently does not reduce the number of words, but it creates a different way of defining traversal.

Similarly, the transformation-related methods in Java SE14 also contain much less words that define control flow. The number of words for other statements not directly performing transformation tasks is also reduced. Instead, the translated bindings now make up a larger proportion of the word count. In our Java SE14 transformations, the code for translated bindings now makes up 27.8% of all words compared to the 22% in Java SE5 and 41.9% of words within the transformation methods. This stems from the usage of streams for processing collections of input elements rather than explicit loops and conditions. As a result the Java SE14 implementation is less control flow driven and focuses more on the data involved. However, while this allows for less lines of code and a reduction in cyclomatic complexity as shown in Section 6.6.1, it does not improve the required number of words. This is because in some cases, the setup overhead for streams counteracts their conciseness gain when using number of words as a measure. An example of this can be seen when comparing Listings 6.21 and 6.22. Both code segments resolve all `InElement`s from the input into their corresponding `OutElement`s and add them to the `OutElements` list of the output. The number of words required in Java SE5 for this totals 14 whereas the number of words in Java SE14 amounts to 17.

```
for (InElement i : input.getInElements()) {
  output.getOutElements()
   .add(TRACER.resolve(i, OutElement.class));
}
```

List. 6.21: Trace resolution example of a collection in Java SE5.

```
output.getOutElements()
 .addAll(input.getInElements().stream()
  .map(i -> TRACER.resolve(i, OutElement.class))
  .collect(Collectors.toList()));
```

List. 6.22: Trace resolution example of a collection in Java SE14.

Overall, our translated transformations in Java SE14 do not reduce the number of words compared to their Java SE5 counterpart. Newer language features do however help in reducing the amount of explicit control flow statements and supplemental code required. Most of this is now done directly in translated bindings which more closely follows the ATL-style. In this sense, Java SE14 helps to take a more data-oriented approach to transformation development compared to Java SE5. However, there is still much overhead from manual traversal, tracing and supplemental code compared to ATL.

When comparing the code segments for writing simple bindings and bindings involving traces in Java SE14 with ATL, there is no difference to the findings from comparing Java SE5 to ATL. This is due to the fact that no Java features introduced since SE5 help in reducing the complexity of code that needs to be written here.

```
private List<Association> associations(Class self) {
  return ALLASSOCIATIONS.stream()
  .filter(asso -> asso.getValue()==1)
  .collect(Collectors.toList());
}
```

List. 6.23: A typical helper in Java SE14.

Comparing translated helper code however, does show some improvements of Java SE14 over Java SE5. Because of the introduction of the streams API, Java SE14 (see Listing 6.23) can now handle expressions involving collections nearly as seamless as ATL (see Listing 6.17). Only the overhead of calling `stream()` and `.collect(Collectors.toList())` remains. This and other observations regarding OCL expressions translated to Java are discussed in more detail later in Section 6.6.4.

Overall, the examples show that code for both simple bindings and bindings involving traces in Java SE14 stays just as complex in comparison to ATL as in Java SE5. Code involving collections, however, can now be expressed nearly as seamless as in ATL due to the introduction of the streams API in Java which offers a notation that is close to OCL notation.

6.6.4 RQ4: How does the size of query aspects of transformations written in Java SE5 & SE14 compare to each other and ATL?

Comparing the word count numbers of helpers from the transformation modules and libraries with their translated counterparts we can once again observe an increase in Java. While all helpers in ATL combined total 2299 words the Java SE5 code totals 3801 words which is an increase of about 65.3%. This was to be expected since Java SE5 is more verbose, especially when handling collections which are required for all helpers within the libraries. This becomes clear when looking at the Java SE5 translation of Listing 6.17 in Listing 6.20. Not only does Java require a loop and if-condition to filter out the desired association subset, a new results list also has to be created and filled with values. Compared to OCLs "→ syntax" this increases the number of required words to produce the same result drastically.

Next, as described in Section 6.5.4 a linear regression was calculated to predict the word count of Java SE5 code for Helpers based on their word count. We were able to find a significant regression model ($p < 2.2e-16$) with an adjusted R^2 of 0.649. The predicted word count of Java SE5 expressions for OCL expressions is estimated as 4.85364 + 1.31554 * $HelperWC$. The hypothesis of a linear relationship is also supported by a pearson coefficient of 0.81 indicating this linear relationship.

Overall, we see a linear relationship between OCL expression code and the translated Java SE5 code. The factor with which the Java code increases in size more quickly is 1.53. This combined with the subjectively less clear way of handling collections through loops leads to the observation that Java5 was not well suited for defining expressions on models.

Looking at the number of words of Java SE14 Helpers compared with their ATL counterparts we see a similar but slightly smaller size than with Java SE5. As stated earlier all ATL library helpers total 2299 words and with 3350 words their Java SE14 counterpart is only about 45.8% larger compared to the 65.3% of Java SE5. This fits well into our observation that the verbose handling of collections is responsible for large portions of the size increase. The streams API, introduced in Java SE8, allows developers a less verbose way of handling collections as can be seen when comparing Listings 6.20 and 6.23. While there is still some overhead compared to the OCL counterpart, namely the necessary calls to `stream()` and `.collect(Collectors.toList())`, the total overhead is greatly reduced. Moreover, this difference could in principal be eliminated by using an alternative GPL. The Scala programming language, for example, does not require a conversion between streams and collections.

The decrease in size can also be observed in our linear regression model that predicts the word count of Java SE14 code for OCL expressions based on the word count of those expressions. The model we were able to find is significant ($p < 2.2e-16$) and has an adjusted R^2 of 0.64. The predicted word count of Java SE14 expressions for OCL expressions is estimated as 5.26631 + 1.09064 * $HelperWC$. And the hypothesis of a linear relationship is again supported by a pearson coefficient of 0.8. Figure 6.9 shows how well both the regression models fit the data. It also highlights the decrease of words required for translated helpers in Java SE14 compared to Java SE5.

The x axis depicts the word count value of OCL expressions while the y axis depicts the word count of Java SE5 codes. The dots within the graph then show the corresponding Java SE5 code word count for each translated OCL expression. Lastly, the red line shows the predicted correspondence based on our regression model.

Overall, we still see a linear relationship between OCL expression code and the translated Java SE14 code. However, the factor with which the Java code increases in size more quickly is only approximately 1.1. This leads us to believe that a well trained Java developer should be able to express OCL queries in Java without much difficulties.

6.7 Discussion

In this section we discuss our findings from Section 6.6 as well as our experiences from the process of translating and using transformations in Java. Our discussion revolves around two main topics.

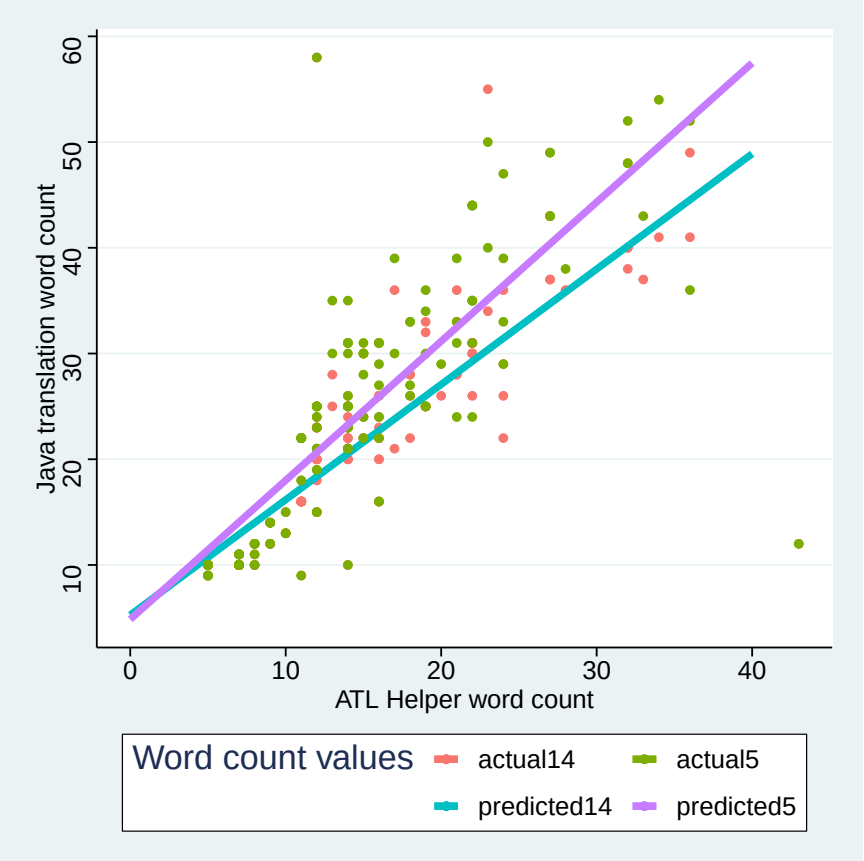


FIGURE 6.9: Comparison of actual Java SE5 and SE14 helper size with predicted size based on linear the regression models.

First, we want to discuss the impact that the design decision to not use anonymous classes to outsource traversal in our Java SE5 solution, explained in Section 6.3.3.2, has on the presented data. Then we discuss how the advancements that have been achieved in newer Java versions influence the ability for developers to efficiently develop transformations in Java. This also includes a conversation about what shortcomings still exist. And second we present a guide that suggests in what cases general purpose languages such as Java can be used in place of ATL. We also show cases where we would advise against writing transformations in Java because of its disadvantages. The argumentation of this part is based on the results presented in this publication as well as our experiences, both from this study as well as previous works (Kehrer et al. 2012, 2016; Rindt et al. 2014; Schultheiß et al. 2020a,b). Finally, we want to have a short discussion beyond the results of our study. Here we want to talk about other features that MTLs can provide and what those could mean for the comparison of MTLs vs. GPLs.

6.7.1 The impact of not outsourcing model traversal in Java SE 5

As explained in Section 6.3.3.2, we decided on using the conditional dispatcher pattern to implement traversal in our Java SE5 solution as opposed to implementing a traversal library, similar to the one used in Java SE14, using anonymous classes. This design decision has implications for the data presented throughout Section 6.6 which we discuss here.

As mentioned, using the presented approach leads to an increased McCabe complexity for the traversal implementation in Java SE5 while it reduces the LOC and number of words. This has concrete implications for the numbers discussed in Sections 6.6.1 to 6.6.3.

For one, this means that when comparing the concrete numbers as done in Section 6.6.1, the stagnation of number of words observed between the Java SE 5 and Java SE 14 variants, would not be present with the alternative Java SE 5 implementation. This is because it would be 812 words longer (making the total number of words 13819) than the presented implementation and thus one would instead observe the expected decrease in number of words in the Java SE 14 implementation.

It would still not be as significant, because only the traversal part of all transformations are affected, but it would be more in line with the reduction in code size observed with the LOC measure in the presented implementations. Moreover, the LOC reduction itself would also be more pronounced because the alternative traversal implementation does require more lines of code per rule. Specifically the total of the Java SE 5 implementation would be increased by 1020 LOC to a total of 4272 as opposed to 3252.

The difference in WMC between the Java SE 5 and Java SE 14 implementation on the other hand would be less clear-cut. As shown in Figure 6.6 a significant portion of the WMC in the presented Java SE5 implementation stems from model traversal. In the alternative implementation this complexity would be significantly reduced by 152 to a total of 640 as opposed to 792. The overall WMC of the Java SE 5 transformations would still be higher, because the utilisation of streams in Java SE 14 reduces the McCabe complexity of other parts of the transformation as well, but it would no longer be nearly halved.

Our observations regarding the differences between implementations in the two different Java versions would, however, not change significantly with the alternative Java SE 5 implementation. Thanks to the functional interfaces and streams, in newer Java versions, a more declarative style for defining transformations can still utilised. The WMC of the code is also still reduced, and the general focus can be directed a more towards the actual transformation aspects. In addition, the observations regarding the comparison of Java and ATL do not change.

6.7.2 Language Advancements and Their Influence on the Ability to Write Transformations: A Historical Perspective

The overall number of words required to write transformations in Java SE14 compared to Java SE5 has not reduced, as shown in Sections 6.6.1 and 6.6.3. However, we have also seen that less explicit control flow needs to be written and the focus shifts more to the binding expressions. This shows in the results discussed in Sections 6.6.1 and 6.6.2 as the cyclomatic complexity of transformations written in Java SE14 is greatly reduced. In principle, a shift towards more data-driven development of transformations is therefore possible. Whether this brings an overall advantage or not is still a debated topic (Götz et al. 2021a) and in our eyes depends on the experience and preference of the developers. However, there are many studies in the field of object-oriented programming that establish a connection between cyclomatic complexity and reliability (Aggarwal et al. 2007; Gopalakrishnan Nair et al. 2012; Guo et al. 2011; J. Pai et al. 2007; Singh et al. 2007), i.e., fault-proneness and error rate, as well as some that establish a connection between cyclomatic complexity and maintainability (Alshayeb et al. 2003; Olbrich et al. 2009), i.e., change frequency and change size.

It has been our experience that newer Java features such as streams and the functional interfaces make the development process easier because less work has to be put into building the traversal, and the assignments within the transformation methods are now a more prominent part of them, i.e., they are less hidden in loops and conditions. Whether these advancements justify writing transformations in Java compared to ATL is discussed in the next section.

6.7.3 A Guideline for When and When Not to Use Java or similar GPLs

As shown in Sections 6.6.2 and 6.6.3, while newer Java features shift the focus more towards a transformation-centric development, there is still significant overhead from setup, manual traversal and especially tracing. Of those three, we believe the setup overhead to be of least relevance. That is because the total overhead for setup is small and it is only an initial overhead that, for the most part, does not need to be maintained throughout the lifecycle of a transformation. The situation is similar for traversal overhead. The code required to be added for all rules or transformation methods, while more significant in its size, still only needs to be written once and can be ignored for most of the remaining development. There is little to no room for errors to be introduced, in any Java implementation that follows a style similar to our implementations, as each new rule requires nearly identical code to be added.

Tracing is where, in our opinion, most of the difficult overhead arises from. It is thus the main argument for writing transformations in ATL or similar MTLs compared to general purpose languages. Managing traces and implementing their complete semantics can not be outsourced into a library, but we can only use a library to reduce the required effort. For many of the advanced

use cases, the mapping semantic relies on String constants that are passed to both the creation and resolution methods, which is error-prone. Such cases arise when traces to objects are needed that were only a side effect of a transformation rule and not its primary output.

There is also little support through type-checking since the only way to store traces for all elements is to use the most generic type possible (i.e., `EObject`). This results in the burden of creating and fetching objects of the correct type to be shifted to the developer, which constitutes a clear disadvantage compared to ATL, where trace resolution is type-safe. In simple cases, this problem is less conspicuous, but in cases where advanced tracing is required, much of the described difficulties arise and can lead to errors that are hard to track to its origin. It also forces developers to be more aware of all parts of the transformation at all times, to make sure not to miss any possible object types that could be returned from resolving a trace. There are approaches, such as Hinkel et al. (2019b), that bring type safety to GPL transformations, but they also come with their own set of limitations when considering advanced features such as incrementality and reusability of the introduced templates, that developers need to be aware of, as well as other boilerplate code that is required to set it up.

Based on the presented reflection, we believe that general purpose languages largely excel in transformations where little to no tracing and especially no advanced tracing is required. The overhead for setup and traversal is manageable in these cases. Moreover, when no traces are required for the transformation, we can scrap the two-phase mechanism completely and thus half the total overhead of traversal is required.

There is also an argument to be made about the expressiveness of Java for complex algorithms compared to the limited capabilities of OCL. We were faced with such a concrete case during the development of a model differencing tool called SiLift (Kehrer et al. 2012). SiLift takes a so-called difference model as input and aims at lifting the given input to a higher level of abstraction by applying in-place transformations to group together interrelated changes. To achieve this low-level changes comprised by the given difference model are first grouped to so-called semantic change sets in a greedy fashion. This greedy strategy, however, can lead to too many change sets. Specifically, we need to get rid of overlapping change sets in a second phase of the transformation, referred to as post-processing in Kehrer et al. (2011). The post-processing poses a set partitioning problem which may be framed as an optimization problem: We want to cover all low-level changes by a minimum amount of semantic change sets which are mutually disjoint. We implemented the heuristics presented by Kehrer et al. (2011) in Java. This can be hardly expressed in OCL, which was developed as a language for querying object structures but not for implementing complex algorithms like the post-processing step of the in-place model transformation scenario described above.

Lastly, related to the previously discussed point of expressiveness, the heterogeneity of Java code compared to ATL code also sticks out. The structure of ATL rules, enforced by ATL's strict syntax, allows for writing consistent code across different transformations. This means that developers can quickly see the basic intent of a rule. The same can not be said for Java methods. While our translation scheme, combined with the developed libraries, produces an internal DSL for transformations, Java code is far less homogeneous due to the absence of any dedicated structure within methods that perform transformations. This can also be seen in our classification from Section 6.4.2. Each Java statement can either have transformation-specific semantics (i.e. *Binding* or *Tracing*) or perform any *other* transformation-unrelated task. This problem of intermixed transformation and non-transformation code within GPLs also persists throughout other internal transformation DSLs such as the NMF transformation languages (Hinkel et al. 2019a), YAMTL (Boronat 2018), RubyTL (Jesús Sánchez Cuadrado et al. 2006) or SiTra (Akehurst et al. 2006). But this does not only bring disadvantages. The strict structure of ATL allows to easily design mappings from one input type to one output type. This can suffice in many cases as highlighted by Götz et al. (2020). However, in cases where several different input types need to be matched to the same output type (n-to-1), one input type needs to be matched to several output types (1-to-n), or a combination of the two cases (n-to-m), code duplicates are often unavoidable. In heterogeneous Java code, such situations can be handled more easily. All in all, the relationship between the input and output meta-models should also be considered when deciding between using an MTL or a GPL.

6.7.4 Limits of our Results in the Context of the Research Field

Up till now our discussion of MTL vs. GPL largely boiled down to the abstraction of model traversal and tracing provided by ATL. This is of course by design as our study focused on the comparison

of Java and ATL. ATL being the most used model transformation language and Java being one of the most dominant programming languages of the last decade. Nonetheless, there are more model transformation-specific features that other model transformation languages provide. Depending on the situation these features could also influence the decision of using a specific model transformation language over general purpose languages.

An extension of the model traversal and matching features of ATL come in the form of graph pattern matching in graph based model transformation languages such as Henshin (Strüber et al. 2017). This allows transformation developers to define complex model element relationships that are automatically searched and matched by advanced matching engines. There exist some advances of trying to replicate this behaviour in general purpose languages for example FunnyQT (Horn 2013) or SDMLib/Fujaba (Zündorf et al. 2013) but even in those cases DSLs are used for defining the graph patterns.

Some model transformation languages allow to run analysis on the written transformations such as critical pair analysis (Born et al. 2015) or even verify property preservation by a transformation (Ehrig et al. 2008), both of which are not easily accessible for transformations written in general purpose languages. The better analysability of MTLs stems from their syntax being transformation-specific, as also seen in the structure of our classification schemata from Section 6.4.

Being able to design bidirectional transformations based on only one transformation script is also a unique property of model transformation languages. Examples of such languages are detailed and compared in Anjorin et al. (2019) or Leblebici et al. (2014). Some languages like eMoflon (Weidmann et al. 2019), NMF Synchronizations (Hinkel et al. 2019a) or Viatra (Bergmann et al. 2015) extend this further by providing the ability to perform incremental transformations both being features that are hard to reproduce in general purpose languages in our experience. Even ATL now has several extensions allowing it to run incremental transformations (Calvar et al. 2019; Martínez et al. 2017).

Currently, for general purpose languages to be considered for writing transformations, all the stated advanced features such as graph pattern matching, bidirectional and incremental transformations as well as transformation analysis and verification should not be an essential requirement of the development. This is because none of them can be implemented with justifiable effort in GPLs.

6.8 Threats to Validity

This section addresses potential threats to the validity of the presented work.

6.8.1 Internal Validity

The manual steps done throughout our study pose some threat to the internal validity of our study. Both the translation based on our translation schema and the labelling of the Java code were done manually and thus open the possibility of human error. Furthermore the program we developed to calculate the word count of the Java code could also contain errors. We counteracted these threats by testing the correctness of the resulting transformations to the extent that was possible based on available resources. This was done by testing the output of the translated transformations against the output of the ATL transformations from which they originated as well as through rigorous peer reviews. We further verified the correctness of our labels and the produced word counts through reviews as detailed in Section 6.5.

All assumptions we make about cause and effect of increase or decrease of size and complexity as well as of overhead is supported by more detailed investigations and analysis throughout our research.

6.8.2 External Validity

To mitigate a potential threat to the external validity of our work due to a bias in the selected transformation modules we chose the analysed transformations from a variety of sources and different authors. Moreover, both the purpose and involved meta-models differ between each transformation module, thus providing a diverse sample set.

However, the transformations chosen for evaluation in our work were subject to a number of constraints which poses a threat to the generalizability of our results. While we aimed to select a variety of transformation modules w.r.t. scope and size, the limitation of LOC may introduce a threat to the external validity of our work.

Due to the study setup of selecting ATL transformations and translating those into Java, there is the possibility of a bias in favour of ATL. It is potentially more likely for an ATL solution to exist, if the problem it solves is well suited for being developed in ATL. As a result the results of our study might not be applicable to all model transformations. However, our study does not try to confirm that ATL is the superior language for developing transformations, but discusses based on the presented observations, which advantages a dedicated language like ATL can offer. In order to be able to recognise why ATL is a good solution for certain cases, it is necessary to look at precisely such cases. In order to validate our results, a further study should be carried out. There, the study design should be reversed so that ATL solutions are derived from existing Java solutions.

Lastly, all our observations are limited to the comparison between ATL and Java which limits their generalizability. While the observations might also hold for comparing Java or similar languages with transformation languages similar to ATL, e.g. QVT-O, they can not be transferred to graph based transformation languages such as Henshin or even QVT-R.

6.8.3 Construct Validity

The next threat concerns the appropriateness and correctness of our translation schema and the resulting transformations. We tried to mitigate this threat by following the design science research method and using two separate reviewers for the proposed transformation schema.

The used metrics for measuring complexity and size need also be discussed. We opted to use cylomatic complexity for measuring the complexity of Java transformations because it is one of the most widely used measures for object-oriented languages, and has ben shown in numerous publications to relate both to the maintainability and reliability of code (Jabangwe et al. 2015). Because both quality attributes are of interest in the discussion of MTLs vs. GPLs, we believe the cyclomatic complexity to be a good measure to assess the impact that overhead Java code has on the quality of transformations. Likewise lines of code are a popular measure for size in all of programming but has also been criticized due to its disregard for the difference in programming styles and formatting. To counteract this problem, all Java code was developed by the same researcher using the same standard code formatter. To further counterbalance issues with lines of code as a solitary size measure, we supplemented it with the additional measure word count that has been argued to be more accurate in measuring the size of a programmed solution (Anjorin et al. 2019). In cases where their ranking differs, we then investigated the cause of the discrepancy and discussed what this means for our observations and analysis.

6.8.4 Conclusion Validity

To ensure reproducible results, we provide all the data and tools used for our study in the supplementary materials for this work. A repetition of our approach using the provided materials will end with the same results as those presented here. However, more than one way of translating ATL constructs into Java constructs and thus multiple translation schemas are possible. This impacts the conclusion validity of our study because different design decisions for the translation schema may impact the reproducibility of our results.

6.9 Related work

To the best of our knowledge, there exists no research that relates the size and complexity of transformations written in a MTL with that of transformations written in a GPL. However, there do exist several publications that provide relevant context for our work.

Hebig et al. investigate the benefit of using specialized model transformation languages compared to general purpose languages by means of a controlled experiment where participants had to complete a comprehension task, a change task, and they had to write one transformation from scratch (Hebig et al. 2018). They compare ATL, QVT-O and the GPL Xtend, and they found no clear evidence for an advantage when using MTLs. In comparison to their setup, we focus on a larger number of transformations. Furthermore, examples shown in the publication also suggest that they did not consider ATLs refining mode for their refactoring task nor did their examples focus on advanced transformation aspects such as tracing.

As previously described, parts of our research build upon the work presented in Götz et al. (2020). Here, the authors use a complexity measure for ATL proposed in the literature to investigate how

the complexity of ATL transformations is distributed over different ATL constructs such as matched rules and helpers. Their results provide a relevant data set to compare our complexity distributions in Java transformations to.

Marcel F van Amstel et al. (2011a) use McCabe complexity to measure the complexity of ATL helpers. Among others, this is also done in (Vignaga 2009). Similar to this, we use McCabe complexity on transformations written in Java, which includes translated helpers, to measure the complexity of the code.

The Model Transformation Tool Contest (TTC)[6] aims to evaluate and compare various quality attributes of model transformation tools. While some of these quality attributes (e.g., readability of a transformation specification) are related to the MTL used by the tool, most of the attributes are related to tooling issues (such as usability or performance) which are out of the scope of our study. Moreover, the contest is about comparing different MTLs with each other rather than comparing them with a GPL. Nonetheless, some cases have been presented along with a reference implementation in Java (Beurer-Kellner et al. 2020; Getir et al. 2017), which could serve as another source for comparing MTLs and GPLs more widely, including tooling- and execution-related aspects.

Sanchez Cuadrado et al. (2020) propose A2L, a compiler for parallel execution of ATL model transformations. A2L takes ATL transformations as input and generates Java code that can be run from within their self-developed engine. Their data-oriented ATL algorithm describes how ATL transformations are executed by their code and closely resembles the structure embodied in our translation schema.

Our approach to utilise libraries and define certain restrictions on the structure of code in Java defines an internal DSL for developing transformations. There exists a large body of research into the topic of the design of internal transformation languages for several general purpose languages. It would be impossible to list them all here. For this reason, we will limit our discussion to a small selection of internal DSLs which have points of contact with our Java DSL.

The Simple Transformation Library in Java (SiTra) introduced by Akehurst et al. (2006) provides a simple set of interfaces for defining transformations in Java. Their interfaces abstract rules and traversal in which they follow an approach similar to ours. However, they do not provide ways for trace management.

Another JVM based transformation DSL is presented by Boronat (2018). The language YAMTL is a declarative internal language for Xtend. In contrast to our approach, this language breaks with the imperative concepts of its host language and offers an ATL-like syntax for defining transformations.

D. S. Batory et al. (2020) describe Aocl, an implementation of OCLs underlying relational algebra for Java. Much like OCL, Aocl allows developers to define constraints and queries for a given model using a straightforward syntax. The authors further argue that, if expanded, Aocl could be used to write model-to-model transformations, but currently this feature does not exist. Using a MDE tool it is possible to generate a Java package that allows to use Aocl for a class diagram passed to the tool.

Hinkel et al. (2019a) introduce NMF-Synchronisations, an internal DSL for C# for developing bidirectional transformations. The language is built with the intention to reuse as much of the tool support form its host language as possible. Much like our Java SE14 approach, they utilise functional language constructs added to C# to allow a more declarative way of defining transformations while still retaining the full potential of the host language.

6.10 Conclusion

In this work, we presented how we developed and applied a translation schema to translate ATL transformations to Java. We also described our results of analysing the complexity and size as well as their distribution over the different transformation aspects. For this purpose, we used McCabe complexity, LOC and word count to measure the size and complexity of 12 transformations translated to Java SE5 and Java SE14, respectively. Based on our findings, we then discussed improvements of Java over the years as well as how well suited these newer language iterations are for writing model transformations.

We found that new features introduced into Java since 2006 help to significantly reduce the complexity of transformations written in Java. Moreover, while they also help to reduce the size of

[6] `https://www.transformation-tool-contest.eu/`

transformations when measured in lines of code, we saw no decrease in the number of words required to write the transformations. This suggests an ability to express more information dense code in newer Java versions. We also showed that, while the overall complexity of transformations is reduced, the distribution of how much of that complexity stems from code that implements functionality that ATL and other model transformation languages can hide from the developer stays about the same. This observation is further supported by the analysis of code size distribution. Here, we found that while large parts of the transformation classes relate to the transformation process itself, within those parts there is still significant overhead from tracing as well as general supplemental code required for the transformations to work. We conclude that while the overall complexity is reduced with newer Java versions, the overhead entailed by using a general purpose language for writing model transformations is still present.

Our regression models for predicting Java code size based on OCL expressions suggest a linear relationship for both Java SE5 and Java SE14 with the newer Java version having a slightly lower growth factor.

Overall we find that the more recent Java version makes development of transformations easier because less work is required to set up a working transformation, and the creation of output elements and the assignment of their attributes are now a more prominent aspect within the code. From our results and experience with this and other projects, we also conclude that general purpose languages are most suitable for transformations where little to no tracing is required because the overhead associated with this transformation aspect is the most prominent one and holds the most potential for errors. However, while we do not see them as prominently used, we believe that advanced features such as property preservation verification or bidirectional and incremental transformation development cannot currently be implemented with justifiable effort in a general purpose language.

For future work, we propose to also look at the transformation development process as a whole, instead of only at the resulting transformations. In particular, we are interested in investigating how the maintenance effort differs between transformations written in a GPL and those written in a MTL. For this purpose, the presented artefacts can be reused. Simple modifications to the ATL transformations can be compared to what needs to be adjusted in the corresponding Java code. Furthermore, because developers are the first to be impacted by the languages, it is also important to include users into such studies. For this reason, we propose to focus on user-centric study setups to be able to better study the impact of the language choice on developers. Such studies could also investigate several other relevant aspects. For example, how well users are aided by *tool support* or the impact of *previous knowledge* of the languages or involved models on the resulting GPL or MTL code. Moreover, the impact of language choice on transformation performance, an aspect that gets more relevant with the ever increasing size of models (Groner et al. 2021), can also be investigated with our setup. Here, we envision the use of run-time measures like execution time and memory or CPU utilization to compare MTL solutions with their GPL counterparts, to investigate the scalability of the underlying technologies.

Another potential avenue to explore is the comparison with a general purpose language that has a more complete support for functional programming such as Scala. Additional features such as pattern matching and easier use of functional syntax for translating OCL expressions could potentially help to further reduce the complexity of the resulting transformation code.

Bibliography

Aggarwal, KK et al. (2007). "Investigating effect of Design Metrics on Fault Proneness in Object-Oriented Systems." In: *J. Object Technol.* 6.10, pp. 127–141 (cit. on p. 189).

Akdur, Deniz et al. (2018). "A survey on modeling and model-driven engineering practices in the embedded software industry". In: *Journal of Systems Architecture* 91, pp. 62–82. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2018.09.007 (cit. on p. 106).

Akehurst, D. H. et al. (2006). "SiTra: Simple Transformations in Java". In: *Model Driven Engineering Languages and Systems*. MODELS 2006. DOI: 10.1007/11880240_25 (cit. on pp. 190, 193).

Alshayeb, M. et al. (2003). "An empirical validation of object-oriented metrics in two different iterative software processes". In: *IEEE Transactions on Software Engineering* 29.11, pp. 1043–1049. DOI: 10.1109/TSE.2003.1245305 (cit. on p. 189).

Alves, Rui et al. (2016). "Ceiling and Threshold of PaaS Tools: The Role of Learnability in Tool Adoption". In: *International Conference on Human-Centred Software Engineering*. HESSD 2016. DOI: 10.1007/978-3-319-44902-9_21 (cit. on p. 48).

Amstel, Marcel F van et al. (2011a). "Using Metrics for Assessing the Quality of ATL Model Transformations". In: *MtATL@ TOOLS* (cit. on pp. 145, 152, 193).

Amstel, Marcel F. van et al. (2011b). "Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare". In: *Theory and Practice of Model Transformations*. ICMT 2011. DOI: 10.1007/978-3-642-21732-6_8 (cit. on p. 44).

Anastasakis, Kyriakos et al. (2007). "UML2Alloy: A Challenging Model Transformation". In: *Model Driven Engineering Languages and Systems*. Ed. by Gregor Engels et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 436–450. ISBN: 978-3-540-75209-7 (cit. on p. 134).

Aniche, Maurício (2015). *Java code metrics calculator (CK)*. https://github.com/mauricioaniche/ck (cit. on p. 177).

Anjorin, Anthony et al. (2017). "The Families to Persons Case". In: TTC'17 (cit. on pp. 5, 66, 115, 164, 165).

Anjorin, Anthony et al. (2019). "Benchmarking bidirectional transformations: theory, implementation, application, and assessment". In: *Software and Systems Modeling (SoSyM)*. DOI: 10.1007/s10270-019-00752-x (cit. on pp. 13, 23, 27, 160, 177, 191, 192).

Arendt, Thorsten et al. (2010). "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations". In: *Model Driven Engineering Languages and Systems*. MODELS 2010. DOI: 10.1007/978-3-642-16145-2_9 (cit. on pp. 5, 35, 44, 62, 64, 113, 114, 140).

Armour, Phillip G (2004). "Beware of counting LOC". In: *Communications of the ACM* 47.3, pp. 21–24 (cit. on p. 27).

Aruoba, S. Boragan et al. (2014). *A Comparison of Programming Languages in Economics*. Tech. rep. National Bureau of Economic Research, Inc. URL: https://EconPapers.repec.org/RePEc:nbr:nberwo:20263 (cit. on p. 55).

Auer, F. et al. (2018). "Current State of Research on Continuous Experimentation: A Systematic Mapping Study". In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. DOI: 10.1109/SEAA.2018.00062 (cit. on p. 39).

Badampudi, Deepika et al. (2015). "Experiences from Using Snowballing and Database Searches in Systematic Literature Studies". In: *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. EASE '15. DOI: 10.1145/2745802.2745818 (cit. on p. 38).

Balogh, András et al. (2006). "Advanced Model Transformation Language Constructs in the VIATRA2 Framework". In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. SAC '06. DOI: 10.1145/1141277.1141575 (cit. on pp. 5, 35, 64, 113, 114, 140).

Barat, Souvik et al. (2017). "A Model-Based Approach to Systematic Review of Research Literature". In: *Proceedings of the 10th Innovations in Software Engineering Conference*. ISEC '17. DOI: 10.1145/3021460.3021462 (cit. on p. 38).

Barb, Adrian S. et al. (2014). "A statistical study of the relevance of lines of code measures in software projects". In: *Innovations in Systems and Software Engineering*. DOI: 10.1007/s11334-014-0231-5 (cit. on p. 45).

Basili, V. et al. (1979). "An Investigation of Human Factors in Software Development". In: *Computer*. DOI: 10.1109/MC.1979.1658573 (cit. on p. 55).

Basili, Victor R. et al. (1994). "The Goal Question Metric Approach". In: *Encyclopedia of Software Engineering* (cit. on p. 36).

Batory, Don et al. (2002). "Achieving Extensibility Through Product-lines and Domain-specific Languages: A Case Study". In: *ACM Trans. Softw. Eng. Methodol.* 11.02. DOI: 10.1145/505145.505147 (cit. on p. 42).

Batory, Don S et al. (2020). "Aocl: A Pure-Java Constraint and Transformation Language for MDE." In: *MODELSWARD*, pp. 319–327 (cit. on pp. 179, 193).

Benelallam, Amine et al. (2015). "Distributed Model-to-model Transformation with ATL on MapReduce". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pp. 37–48 (cit. on p. 134).

Bergmann, Gábor et al. (2015). "Viatra 3: A Reactive Model Transformation Platform". In: *Theory and Practice of Model Transformations*. Ed. by Dimitris Kolovos et al. Cham: Springer International Publishing, pp. 101–110. ISBN: 978-3-319-21155-8 (cit. on p. 191).

Beurer-Kellner, Luca et al. (2020). "Round-trip migration of object-oriented data model instances". In: *Transformation Tool Contest at the Conference on Software Technologies: Applications and Foundations (TTC@STAF)* (cit. on p. 193).

Bézivin, Jean (2004). "In search of a basic principle for model driven engineering". In: *Novatica Journal, Special Issue* 5.2, pp. 21–24 (cit. on p. 2).

Biermann, Enrico et al. (2010). "Lifting parallel graph transformation concepts to model transformation based on the eclipse modeling framework". In: *Electronic Communications of the EASST* 26 (cit. on p. 134).

Boehm, Barry et al. (1995). "Cost models for future software life cycle processes: COCOMO 2.0". In: *Annals of Software Engineering*. DOI: 10.1007/BF02249046 (cit. on p. 55).

Boot, Andrew et al. (2016). *Systematic Approaches to a Successful Literature Review*. Sage. ISBN: 978-1-4739-1245-8 (cit. on pp. 12, 34, 35, 40, 42).

Born, Kristopher et al. (2015). "Analyzing Conflicts and Dependencies of Rule-Based Transformations in Henshin". In: *Fundamental Approaches to Software Engineering*. Ed. by Alexander Egyed et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 165–168. ISBN: 978-3-662-46675-9 (cit. on p. 191).

Boronat, Artur (2018). "Expressive and Efficient Model Transformation with an Internal DSL of Xtend". In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '18. Copenhagen, Denmark: Association for Computing Machinery, pp. 78–88. ISBN: 9781450349499. DOI: 10.1145/3239372.3239386. URL: https://doi.org/10.1145/3239372.3239386 (cit. on pp. 190, 193).

Brambilla, Marco et al. (2017). *Model-Driven Software Engineering in Practice*. Springer International Publishing. DOI: 10.1007/978-3-031-02549-5 (cit. on pp. 2, 3, 101).

Brown, Alan W et al. (2005). "Introduction: Models, modeling, and model-driven architecture (mda)". In: *Model-Driven Software Development*. Springer, pp. 1–16. DOI: 10.1007/3-540-28554-7_1 (cit. on pp. 2, 63, 113).

Bucchiarone, Antonio et al. (2021). "What Is the Future of Modeling?" In: *IEEE Softw.* 38.2, pp. 119–127. DOI: 10.1109/MS.2020.3041522 (cit. on p. 100).

Buchmann, Thomas (2018). "BXtend-A Framework for (Bidirectional) Incremental Model Transformations." In: *MODELSWARD*, pp. 336–345 (cit. on p. 177).

Buckler, Frank et al. (2008). "Identifying hidden structures in marketing's structural models through universal structure modeling". In: *Marketing ZFP* 30.JRM 2, pp. 47–66 (cit. on pp. 12, 18, 99, 111, 119, 122).

Burgueño, Loli et al. (2019). "The Future of Model Transformation Languages: An Open Community Discussion". In: *Journal of Object Technology*. DOI: 10.5381/jot.2019.18.3.a7 (cit. on pp. 1, 49, 56, 102, 104, 110, 134, 140, 158).

Burkhardt, Jean-Marie et al. (2002). "Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase". In: *Empirical Software Engineering*. DOI: 10.1023/A:1015297914742 (cit. on p. 55).

Calvar, Théo Le et al. (2019). "Efficient ATL Incremental Transformations". In: *Journal of Object Technology*. DOI: 10.5381/jot.2019.18.3.a2 (cit. on pp. 56, 191).

Cary, John R. et al. (1997). "Comparison of C++ and Fortran 90 for object-oriented scientific programming". English. In: *Computer Physics Communications* (cit. on p. 56).

Charmaz, Kathy (2014). *Constructing grounded theory*. Sage. ISBN: 9780857029133 (cit. on pp. 13, 41, 74, 77).

Chechik, Marsha et al. (2016). "Perspectives of Model Transformation Reuse". In: *Integrated Formal Methods*. Ed. by Erika Ábrahám et al. Cham: Springer International Publishing, pp. 28–44. ISBN: 978-3-319-33693-0 (cit. on p. 132).

Cicchetti, Antonio et al. (2011). "JTL: A Bidirectional and Change Propagating Transformation Language". In: *Software Language Engineering*. Ed. by Brian Malloy et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 183–202. ISBN: 978-3-642-19440-5 (cit. on p. 177).

Ciccozzi, Federico et al. (2019). "Execution of UML models: a systematic review of research and practice". In: *Software & Systems Modeling*. DOI: 10.1007/s10270-018-0675-4 (cit. on pp. 35, 57).

Codd, E. F. (1970). "A relational model of data for large shared data banks". In: *Communications of the ACM* 13.6, pp. 377–387. DOI: 10.1145/362384.362685 (cit. on p. 3).

Cook, T.D. et al. (1979). *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin (cit. on pp. 27, 28).

Cuadrado, Jesús Sánchez et al. (2006). "RubyTL: A Practical, Extensible Transformation Language". In: *Model Driven Architecture – Foundations and Applications*. ECMDA-FA 2006. DOI: 10.1007/11787044_13 (cit. on pp. 5, 47, 64, 114, 190).

Czarnecki, K. et al. (2006). "Feature-based survey of model transformation approaches". In: *IBM Systems Journal*. DOI: 10.1147/sj.453.0621 (cit. on pp. 3, 5, 7, 10, 35, 54, 56, 64, 66–68, 79, 113–117, 162).

Demuth, Birgit et al. (2004). "Structure of the Dresden OCL toolkit". In: *2nd International Fujaba Days "MDA with UML and Rule-based Object Manipulation". Darmstadt, Germany, September*, pp. 15–17 (cit. on p. 25).

Demuth, Birgit et al. (2009). "Model and object verification by using Dresden OCL". In: *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa, Russia*. Citeseer, pp. 687–690 (cit. on p. 101).

Di Rocco, Juri et al. (2015). "Mining correlations of ATL model transformation and metamodel metrics". In: *2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering*. DOI: 10.1109/MiSE.2015.17 (cit. on pp. 11, 143, 152).

Dieste, Oscar et al. (2017). "Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study". In: *Empirical Software Engineering*. DOI: 10.1007/s10664-016-9471-3 (cit. on pp. 55, 100).

Dosovitskiy, Alexey et al. (2017). "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*, pp. 1–16 (cit. on p. 100).

Eclipse Foundation (2022). *Eclipse Graphical Language Server Platform (GLSP)*. URL: https://www.eclipse.org/glsp/ (cit. on p. 101).

Ege, Florian et al. (2019). "A Proposal of Features to Support Analysis and Debugging of Declarative Model Transformations with Graphical Syntax by Embedded Visualizations". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 326–330. DOI: 10.1109/MODELS-C.2019.00051 (cit. on p. 101).

Ehrig, Hartmut et al. (2008). "Semantical Correctness and Completeness of Model Transformations Using Graph and Rule Transformation". In: *Graph Transformations*. Ed. by Hartmut Ehrig et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 194–210. ISBN: 978-3-540-87405-8 (cit. on p. 191).

Facebook, Inc (2016). URL: http://facebook.github.io/graphql (cit. on p. 26).

Fang, J. et al. (2011). "A Comprehensive Performance Comparison of CUDA and OpenCL". In: *2011 International Conference on Parallel Processing*. ICPP 2011. DOI: 10.1109/ICPP.2011.45 (cit. on p. 56).

Fowler, Martin (2011). *Domain-specific languages*. Addison-Wesley, p. 597. ISBN: 0321712943 (cit. on pp. 2, 3).

Frakes, William B et al. (2001). "An industrial study of reuse, quality, and productivity". In: *Journal of Systems and Software*. DOI: 10.1016/S0164-1212(00)00121-7 (cit. on p. 55).

France, Robert (2008). "Fair treatment of evaluations in reviews". In: *Software & Systems Modeling* 7.3, pp. 253–254. DOI: 10.1007/s10270-008-0096-x (cit. on pp. 1, 25).

Francis, Nadime et al. (2018). "Cypher: An Evolving Query Language for Property Graphs". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, pp. 1433–1445. ISBN: 9781450347037. DOI: 10.1145/3183713.3190657 (cit. on p. 26).

Freeman, Steve et al. (2004). "JMock: Supporting Responsibility-Based Design with Mock Objects". In: OOPSLA '04. Vancouver, BC, CANADA: Association for Computing Machinery, pp. 4–5. DOI: 10.1145/1028664.1028667 (cit. on p. 3).

Fuchs, Christoph et al. (2009). "Using single-item measures for construct measurement in management research: Conceptual issues and application guidelines". In: *Die Betriebswirtschaft* 69.2, p. 195 (cit. on p. 121).

Galster, M. et al. (2014). "Variability in Software Systems—A Systematic Literature Review". In: *IEEE Transactions on Software Engineering*. DOI: 10.1109/TSE.2013.56 (cit. on p. 43).

George, Lars et al. (2012). "Type-Safe Model Transformation Languages as Internal DSLs in Scala". In: *Theory and Practice of Model Transformations*. ICMT 2012. DOI: 10.1007/978-3-642-30476-7_11 (cit. on pp. 64, 113).

Gerpheide, Christine M. et al. (2016). "Assessing and improving quality of QVTo model transformations". In: *Software Quality Journal* 24.3, pp. 797–834. ISSN: 1573-1367. DOI: 10.1007/s11219-015-9280-8. URL: https://doi.org/10.1007/s11219-015-9280-8 (cit. on pp. 104, 135).

Getir, Sinem et al. (2017). "State Elimination as Model Transformation Problem". In: *Transformation Tool Contest at the Conference on Software Technologies: Applications and Foundations (TTC@STAF)*, pp. 65–73 (cit. on p. 193).

Gherardi, Luca et al. (2012). "A Java vs. C++ Performance Evaluation: A 3D Modeling Benchmark". In: *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. SIMPAR 2012. DOI: 10.1007/978-3-642-34327-8_17 (cit. on p. 55).

Gopalakrishnan Nair, T.R. et al. (2012). "Defect proneness estimation and feedback approach for software design quality improvement". In: *Information and Software Technology* 54.3, pp. 274–285. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2011.10.001 (cit. on p. 189).

Götz, Stefan et al. (July 2020). "Investigating the Origins of Complexity and Expressiveness in ATL Transformations". In: *Journal of Object Technology* 19.2. Ed. by Richard Paige et al. The 16th European Conference on Modelling Foundations and Applications (ECMFA 2020), 12:1–21. DOI: 10.5381/jot.2020.19.2.a12 (cit. on pp. 135, 159, 160, 171, 172, 177, 178, 184, 190, 192).

Götz, Stefan et al. (2021a). "Claimed advantages and disadvantages of (dedicated) model transformation languages: a systematic literature review". In: *Software and Systems Modeling* 20.2, pp. 469–503. ISSN: 1619-1374. DOI: 10.1007/s10270-020-00815-4. URL: https://doi.org/10.1007/s10270-020-00815-4 (cit. on pp. 1, 14, 19, 62, 68–71, 73, 75, 98, 104, 110, 119, 134, 158, 189).

Götz, Stefan et al. (2021b). "Dedicated Model Transformation Languages vs. General-purpose Languages: A Historical Perspective on ATL vs. Java". In: *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,* INSTICC. SciTePress, pp. 122–135. DOI: 10.5220/0010340801220135 (cit. on pp. 104, 160).

Gray, J. et al. (2003). "An examination of DSLs for concisely representing model traversals and transformations". In: *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*. HICSS '03. DOI: 10.1109/HICSS.2003.1174892 (cit. on pp. 20, 140, 158).

Graziotin, Daniel et al. (2021). "Psychometrics in Behavioral Software Engineering: A Methodological Introduction with Guidelines". In: *ACM Trans. Softw. Eng. Methodol.* 31.1. DOI: 10.1145/3469888 (cit. on p. 117).

Greiner, Sandra et al. (2023). "Incremental MTL vs. GPLs: Class into Relational Database Schema". In: *Transformation Tool Contest (TTC)*. URL: http://www.transformation-tool-contest.eu/TTC_2023_paper_4.pdf (cit. on p. 26).

Groner, Raffaela et al. (2020). "An Exploratory Study on Performance Engineering in Model Transformations". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '20. Virtual Event, Canada: Association for Computing Machinery, pp. 308–319. DOI: 10.1145/3365438.3410950 (cit. on pp. 10, 11, 70, 105).

Groner, Raffaela et al. (2021). "A Survey on the Relevance of the Performance of Model Transformations". In: *Journal of Object Technology* 20.2. OPEN ISSUE, pp. 1–27. ISSN: 1660-1769. DOI: 10.5381/jot.2021.20.2.a5 (cit. on pp. 121, 122, 194).

Guo, Y. et al. (2011). "An Empirical Validation of the Benefits of Adhering to the Law of Demeter". In: *2011 18th Working Conference on Reverse Engineering*, pp. 239–243. DOI: 10.1109/WCRE.2011.36 (cit. on p. 189).

Hailpern, B. et al. (2006). "Model-driven development: The good, the bad, and the ugly". In: *IBM Systems Journal*. DOI: 10.1147/sj.453.0451 (cit. on pp. 42, 49).

Hassan, Ahmed E. (2008). "The road ahead for Mining Software Repositories". In: *2008 Frontiers of Software Maintenance*, pp. 48–57. DOI: 10.1109/FOSM.2008.4659248 (cit. on p. 12).

Hebig, Regina et al. (2018). "Model Transformation Languages Under a Magnifying Glass: A Controlled Experiment with Xtend, ATL, and QVT". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. DOI: 10.1145/3236024.3236046 (cit. on pp. 11, 20, 25, 37, 98, 104, 110, 135, 140, 158, 192).

Henderson, Robert et al. (1994). "A comparison of object-oriented programming in four modern languages". In: *Software: Practice and Experience*. DOI: 10.1002/spe.4380241106 (cit. on p. 55).

Hermans, Felienne et al. (2009). "Domain-Specific Languages in Practice: A User Study on the Success Factors". In: *Model Driven Engineering Languages and Systems*. MODELS 2009. ISBN: 978-3-642-04425-0. DOI: 10.1007/978-3-642-04425-0_33 (cit. on pp. 62, 110).

Hevner, Alan R et al. (2004). "Design science in information systems research". In: *MIS quarterly*, pp. 75–105. DOI: 10.2307/25148625 (cit. on p. 12).

Hibberd, Mark et al. (2007). "Forensic Debugging of Model Transformations". In: *Model Driven Engineering Languages and Systems*. Ed. by Gregor Engels et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 589–604. ISBN: 978-3-540-75209-7 (cit. on p. 101).

Hinkel, Georg (2013). "An approach to maintainable model transformations with an internal DSL". PhD thesis. National Research Center (cit. on p. 34).

— (2016). *NMF: A Modeling Framework for the. NET Platform*. KIT (cit. on pp. 2, 63, 113, 177).

Hinkel, Georg et al. (2019a). "Change propagation and bidirectionality in internal transformation DSLs". In: *Software & Systems Modeling*. DOI: 10.1007/s10270-017-0617-6 (cit. on pp. 5, 47, 55, 64, 113, 114, 190, 191, 193).

Hinkel, Georg et al. (2019b). "Using internal domain-specific languages to inherit tool support and modularity for model transformations". In: *Software & Systems Modeling*. DOI: 10.1007/s10270-017-0578-9 (cit. on pp. 20, 37, 62, 102, 140, 190).

Höppner, Stefan et al. (2021). *Contrasting Dedicated Model Transformation Languages vs. General Purpose Languages: A Historical Perspective on ATL vs. Java based on Complexity and Size: Supplementary Materials*. DOI: http://dx.doi.org/10.18725/OPARU-38923 (cit. on pp. 99, 110, 128, 135, 159).

Höppner, Stefan et al. (2022a). "Advantages and disadvantages of (dedicated) model transformation languages". In: *Empirical Software Engineering* 27.6, p. 159. DOI: 10.1007/s10664-022-10194-7 (cit. on pp. 1, 3, 17, 110, 113, 122, 126, 132, 134).

Höppner, Stefan et al. (2022b). *The Impact of Model Transformation Language Features on Quality Properties of MTLs: A Study Protocol*. DOI: 10.48550/ARXIV.2209.06570 (cit. on pp. 18, 111, 113, 120).

Horn, Tassilo (2013). "Model Querying with FunnyQT". In: *Theory and Practice of Model Transformations*. Ed. by Keith Duddy et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 56–57. ISBN: 978-3-642-38883-5 (cit. on pp. 5, 64, 113, 114, 191).

Hove, S. E. et al. (2005). "Experiences from conducting semi-structured interviews in empirical software engineering research". In: *11th IEEE International Software Metrics Symposium (METRICS'05)*, 10 pp.–23. DOI: 10.1109/METRICS.2005.24 (cit. on pp. 12, 68–70).

Hutchinson, John et al. (2011a). "Empirical Assessment of MDE in Industry". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. DOI: 10.1145/1985793.1985858 (cit. on pp. 10, 29, 56, 97, 99, 105).

Hutchinson, John et al. (2011b). "Model-Driven Engineering Practices in Industry". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki Honolulu HI, USA: Association for Computing Machinery, pp. 633–642. ISBN: 9781450304450. DOI: 10.1145/1985793.1985882. URL: https://doi.org/10.1145/1985793.1985882 (cit. on pp. 10, 97, 105).

Hutchinson, John et al. (2014). "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure". In: *Science of Computer Programming* 89. Special issue on Success Stories in Model Driven Engineering, pp. 144–161. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2013.03.017. URL: https://www.sciencedirect.com/science/article/pii/S0167642313000786 (cit. on pp. 10, 105).

ISO/IEC 25010:2011 (2011). ISO/IEC. URL: https://www.iso.org/standard/22749.html (cit. on p. 53).

J. Pai, G. et al. (2007). "Empirical Analysis of Software Fault Content and Fault Proneness Using Bayesian Methods". In: *IEEE Transactions on Software Engineering* 33.10, pp. 675–686. DOI: 10.1109/TSE.2007.70722 (cit. on p. 189).

Jabangwe, Ronald et al. (2015). "Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review". In: *Empirical Software Engineering* 20.3, pp. 640–693. DOI: 10.1007/s10664-013-9291-7 (cit. on pp. 27, 177, 192).

Jakumeit, Edgar et al. (2014). "A survey and comparison of transformation tools based on the transformation tool contest". In: *Science of Computer Programming*. DOI: 10.1016/j.scico.2013.10.009 (cit. on pp. 10, 11, 57).

Johannes, Jendrik et al. (2009). "Abstracting Complex Languages through Transformation and Composition". In: *Model Driven Engineering Languages and Systems*. MODELS 2009. DOI: 10.1007/978-3-642-04425-0_41 (cit. on pp. 62, 110).

Jones, Capers (2000). *Software assessments, benchmarks, and best practices*. Addison-Wesley Longman Publishing Co., Inc. ISBN: 978-0201485424 (cit. on p. 27).

Jones, T. C. (1978). "Measuring programming quality and productivity". In: *IBM Systems Journal*. DOI: 10.1147/sj.171.0039 (cit. on p. 55).

Jonkers, Hans et al. (2006). "Bootstrapping domain-specific model-driven software development within Philips". In: *6th OOPSLA Workshop on Domain Specific Modeling (DSM 2006)*. Citeseer, p. 10 (cit. on p. 102).

Jouault, Frédéric (June 2013). *ATL/Tutorials - Create a simple ATL transformation*. https://wiki.eclipse.org/ATL/Tutorials_-_Create_a_simple_ATL_transformation. Accessed: 2021-06-12 (cit. on p. 164).

Jouault, Frédéric et al. (2006). "ATL: A QVT-like Transformation Language". In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '06. DOI: 10.1145/1176617.1176691 (cit. on pp. 5, 35, 62, 64, 113, 114, 140, 158).

Jouault, Frédéric et al. (2008). "ATL: A model transformation tool". In: *Science of Computer Programming*. DOI: 10.1016/j.scico.2007.08.002 (cit. on pp. 20, 140, 158).

Juhnke, Katharina et al. (2020). "Challenges concerning test case specifications in automotive software testing: assessment of frequency and criticality". In: *Software Quality Journal*. ISSN: 1573-1367. DOI: 10.1007/s11219-020-09523-0. URL: https://doi.org/10.1007/s11219-020-09523-0 (cit. on pp. 70, 99).

Kahani, Nafiseh et al. (2019). "Survey and classification of model transformation tools". In: *Software & Systems Modeling*. DOI: 10.1007/s10270-018-0665-6 (cit. on pp. 3, 10, 35, 56, 64, 79, 110, 114).

Kallio, Hanna et al. (2016). "Systematic methodological review: developing a framework for a qualitative semi-structured interview guide". In: *Journal of Advanced Nursing* 72.12, pp. 2954–2965. DOI: https://doi.org/10.1111/jan.13031 (cit. on pp. 68, 69).

Kapová, Lucia et al. (2010). "Evaluating maintainability with code metrics for model-to-model transformations". In: *International Conference on the Quality of Software Architectures*. DOI: https://doi.org/10.1007/978-3-642-13821-8_12 (cit. on p. 143).

Karimi, Kamran et al. (2010). "A Performance Comparison of CUDA and OpenCL". In: *ArXiv* abs/1005.2581 (cit. on p. 56).

Kasunic, Mark (2005). *Designing an effective survey*. Tech. rep. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst (cit. on pp. 12, 18).

Kehrer, Timo et al. (2011). "A rule-based approach to the semantic lifting of model differences in the context of model versioning". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, pp. 163–172 (cit. on p. 190).

Kehrer, Timo et al. (2012). "Understanding model evolution through semantically lifting model differences with SiLift". In: *28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, pp. 638–641 (cit. on pp. 158, 188, 190).

Kehrer, Timo et al. (2016). "Automatically deriving the specification of model editing operations from meta-models". In: *International Conference on Theory and Practice of Model Transformations*. Springer, pp. 173–188 (cit. on pp. 158, 188).

Kernighan, Brian W. et al. (1984). *The Unix Programming Environment*. Prentice Hall, Inc. ISBN: 0-13-937699-2 (cit. on pp. 3, 64, 113).

Kieburtz, Richard B. et al. (1996). "A Software Engineering Experiment in Software Component Generation". In: *Proceedings of the 18th International Conference on Software Engineering*. ICSE'96. DOI: 10.1109/ICSE.1996.493448 (cit. on p. 42).

Kitchenham, B. et al. (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering*. DOI: 10.1.1.117.471 (cit. on pp. 12, 34).

Ko, Jong-Won et al. (2015). "Model transformation verification using similarity and graph comparison algorithm". In: *Multimedia Tools and Applications* 74.20, pp. 8907–8920. ISSN: 1573-7721. DOI: 10.1007/s11042-013-1581-y (cit. on p. 134).

Kofod-Petersen, Anders (2015). *How to do a Structured Literature Review in computer science* (cit. on p. 37).

Kolovos, Dimitrios S. et al. (2008). "The Epsilon Transformation Language". In: *Theory and Practice of Model Transformations*. ICMT 2008. DOI: 10.1007/978-3-540-69927-9_4 (cit. on pp. 35, 64, 113, 140).

Kosar, Tomaz et al. (2010). "Comparing general-purpose and domain-specific languages: An empirical study". In: *ComSIS–Computer Science an Information Systems Journal*. DOI: 10.2298/CSIS1002247K (cit. on pp. 10, 57).

Kosar, Tomaž et al. (2016). "Domain-Specific Languages: A Systematic Mapping Study". In: *Information and Software Technology*. DOI: 10.1016/j.infsof.2015.11.001 (cit. on pp. 10, 55–57, 140).

Kramer, M. E. et al. (2016). "A controlled experiment template for evaluating the understandability of model transformation languages". In: *2nd International Workshop on Human Factors in Modeling, HuFaMo 2016; Saint Malo; France; 4 October 2016 through. Ed. : M. Goulao*. Vol. 1805. CEUR Workshop Proceedings. CEUR Workshop Proceedings, pp. 11–18 (cit. on pp. 11, 105, 135).

Krause, Christian et al. (2014). "Implementing Graph Transformations in the Bulk Synchronous Parallel Model". In: *Fundamental Approaches to Software Engineering*. Ed. by Stefania Gnesi et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 325–339. ISBN: 978-3-642-54804-8 (cit. on pp. 6, 66, 116).

Krikava, Filip et al. (2014). "Manipulating Models Using Internal Domain-Specific Languages". In: *Symposium On Applied Computing*. SAC '14. DOI: 10.1145/2554850.2555127 (cit. on pp. 20, 140, 158).

Křikava, Filip et al. (2014). "SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations". In: *Model-Driven Engineering Languages and Systems*. MODELS 2014. DOI: 10.1007/978-3-319-11653-2_35 (cit. on pp. 47, 55).

Kuckartz, Udo (2014). *Qualitative text analysis: A guide to methods, practice and using software*. Sage. ISBN: 978-1-4462-6774-5 (cit. on pp. 16, 68, 74–78).

Kurniawan, Budi et al. (2004). "A Comparative Study of Web Application Design Models Using the Java Technologies". In: *Asia-Pacific Web Conference*. APWeb 2004. DOI: 10.1007/978-3-540-24655-8_77 (cit. on p. 55).

Kurtev, Ivan (2007). "State of the art of QVT: A model transformation language standard". In: *International Symposium on Applications of Graph Transformations with Industrial Relevance*. DOI: https://doi.org/10.1007/978-3-540-89020-1_26 (cit. on p. 140).

Kusel, A. et al. (2015). "Reuse in model-to-model transformation languages: are we there yet?" In: *Software & Systems Modeling*. DOI: 10.1007/s10270-013-0343-7 (cit. on p. 132).

Kusel, Angelika et al. (2013). "Reality Check for Model Transformation Reuse: The ATL Transformation Zoo Case Study." In: *Amt@ models*, pp. 1–11 (cit. on pp. 11, 152).

Lano, Kevin et al. (2015). "A framework for model transformation verification". In: *Formal Aspects of Computing* 27.1, pp. 193–235 (cit. on p. 134).

Lano, Kevin et al. (2018). "Technical Debt in Model Transformation Specifications". In: *Theory and Practice of Model Transformation*. Cham: Publishing. DOI: 10.1007/978-3-319-93317-7_6 (cit. on pp. 12, 20, 23, 141, 143, 144, 153).

Lawley, Michael et al. (2007). "Implementing a Practical Declarative Logic-based Model Transformation Engine". In: *Proceedings of the 2007 ACM Symposium on Applied Computing*. SAC '07. DOI: 10.1145/1244002.1244216 (cit. on pp. 20, 62, 140).

Leblebici, Erhan et al. (2014). "A comparison of incremental triple graph grammar tools". In: *Electronic Communications of the EASST* 67. DOI: 10.14279/tuj.eceasst.67.939 (cit. on p. 191).

Liebel, Grischa et al. (2018). "Organisation and communication problems in automotive requirements engineering". In: *Requirements Engineering* 23.1, pp. 145–167. ISSN: 1432-010X. DOI: 10.1007/s00766-016-0261-7. URL: https://doi.org/10.1007/s00766-016-0261-7 (cit. on p. 99).

Liepiņš, Renārs (2012). "Library for Model Querying: IQuery". In: *Proceedings of the 12th Workshop on OCL and Textual Modelling*. OCL '12. DOI: 10.1145/2428516.2428522 (cit. on p. 62).

Loniewski, Grzegorz et al. (2010). "A Systematic Review of the Use of Requirements Engineering Techniques in Model-Driven Development". In: *Model Driven Engineering Languages and Systems*. DOI: 10.1007/978-3-642-16129-2_16 (cit. on p. 38).

Malavolta, I. et al. (2010). "Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies". In: *IEEE Transactions on Software Engineering* 36.1, pp. 119–140. DOI: 10.1109/TSE.2009.51 (cit. on p. 62).

Martínez, Salvador et al. (2017). "Reactive model transformation with ATL". In: *Science of Computer Programming* 136, pp. 1–16. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2016.08.006 (cit. on p. 191).

Mayring, Philipp (1994). *Qualitative inhaltsanalyse*. Vol. 14. UVK Univ.-Verl. Konstanz. ISBN: 978-3-407-29393-0 (cit. on p. 68).

McCabe, T.J. (1976). "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4, pp. 308–320. DOI: 10.1109/TSE.1976.233837 (cit. on pp. 13, 160).

Meijer, Erik et al. (2006). "LINQ: Reconciling Object, Relations and XML in the .NET Framework". In: SIGMOD '06. Chicago, IL, USA: Association for Computing Machinery, p. 706. DOI: 10.1145/1142473.1142552 (cit. on p. 3).

Mens, Tom et al. (2006). "A Taxonomy of Model Transformation". In: *Electronic Notes in Theoretical Computer Science (GraMoT 2005)*. DOI: 10.1016/j.entcs.2005.10.021 (cit. on pp. 3, 10, 35, 56, 64, 114).

Mernik, Marjan et al. (2005). "When and How to Develop Domain-specific Languages". In: *ACM computing surveys (CSUR)*. DOI: 10.1145/1118890.1118892 (cit. on pp. 1, 34, 47, 48).

Metzger, Andreas (2005). "A systematic look at model transformations". In: *Model-driven Software Development*. Springer, pp. 19–33. DOI: 10.1007/3-540-28554-7_2 (cit. on pp. 2, 62, 64, 110, 113).

Meyer, M A et al. (1990). "Eliciting and analyzing expert judgment: A practical guide". In: DOI: 10.2172/5088782 (cit. on pp. 12, 68–70).

Microsoft (2022). *Language Server Protocol Specification*. URL: https://microsoft.github.io/language-server-protocol/specifications/specification-current/ (cit. on p. 101).

Mohagheghi, Parastoo et al. (2008). "Where Is the Proof? - A Review of Experiences from Applying MDE in Industry". In: *Model Driven Architecture – Foundations and Applications*. Ed. by Ina Schieferdecker et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 432–443 (cit. on pp. 10, 100, 106).

Mohagheghi, Parastoo et al. (2013a). "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases". In: *Empirical Software Engineering*. DOI: 10.1007/s10664-012-9196-x (cit. on pp. 10, 55, 95, 100, 106).

Mohagheghi, Parastoo et al. (2013b). "Where does model-driven engineering help? Experiences from three industrial cases". In: *Software & Systems Modeling* 12.3, pp. 619–639. ISSN: 1619-1374. DOI: 10.1007/s10270-011-0219-7. URL: https://doi.org/10.1007/s10270-011-0219-7 (cit. on pp. 10, 106).

Mooney, Christopher Z et al. (1993). *Bootstrapping: A nonparametric approach to statistical inference*. 95. sage (cit. on pp. 119, 122).

Newcomer, Kathryn E et al. (2015). *Handbook of practical program evaluation*. John Wiley & Sons. ISBN: 978-1-118-89360-9 (cit. on pp. 69–71).

Olbrich, S. et al. (2009). "The evolution and impact of code smells: A case study of two open source systems". In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 390–400. DOI: 10.1109/ESEM.2009.5314231 (cit. on p. 189).

OMG (July 2001). *Model Driven Architecture (MDA), ormsc/2001-07-01* (cit. on pp. 35, 63, 113).

— (Apr. 2002). *Meta Object Facility(MOF)* (cit. on pp. 2, 63, 113, 161).

— (2014). *Object Constraint Language (OCL).* URL: https://www.omg.org/spec/OCL/2.4/PDF (cit. on pp. 8, 68, 117, 141, 161).
— (2016). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification.* URL: https://www.omg.org/spec/QVT/About-QVT/ (cit. on pp. 5, 62, 66, 115).
Pietron, Jakob et al. (2018). "A study design template for identifying usability issues in graphical modeling tools." In: *MODELS Workshops*, pp. 336–345 (cit. on p. 102).
Prechelt, L. (2000). "An empirical comparison of seven programming languages". In: *Computer.* DOI: 10.1109/2.876288 (cit. on p. 55).
Raggett, Dave et al. (1999). "HTML 4.01 Specification". In: *W3C recommendation* 24 (cit. on pp. 3, 64, 113).
Rainer, Austen et al. (2021). *Recruiting credible participants for field studies in software engineering research.* DOI: 10.48550/ARXIV.2112.14186 (cit. on pp. 1, 98).
Rein, Patrick et al. (2019). "Towards Empirical Evidence on the Comprehensibility of Natural Language Versus Programming Language". In: *Design Thinking Research.* DOI: 10.1007/978-3-030-28960-7_7 (cit. on p. 55).
Rentschler, Andreas et al. (2014). "Designing Information Hiding Modularity for Model Transformation Languages". In: *Proceedings of the 13th International Conference on Modularity.* MODULARITY '14. DOI: 10.1145/2577080.2577094 (cit. on p. 159).
Rindt, Michaela et al. (2014). "Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools". In: *Demos@ MoDELS* 14 (cit. on pp. 158, 188).
Runeson, Per et al. (2012). "Case study research in software engineering". In: *Guidelines and examples*, p. 237 (cit. on p. 12).
SAEMobilus (2004). *Architecture Analysis and Design Language (AADL)* (cit. on pp. 3, 64, 113).
Samiee, Amir et al. (2018). "Model-Driven-Engineering in Education". In: *2018 18th International Conference on Mechatronics - Mechatronika (ME)*, pp. 1–6 (cit. on p. 100).
Sanchez Cuadrado, J. et al. (2020). "Efficient execution of ATL model transformations using static analysis and parallelism". In: *IEEE Transactions on Software Engineering.* DOI: 10.1109/TSE.2020.3011388 (cit. on pp. 134, 165, 193).
Schmidt, Douglas (2006). "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer - IEEE Computer Society.* DOI: 10.1109/MC.2006.58 (cit. on pp. 2, 62, 63, 110, 113, 140).
Schultheiß, Alexander et al. (2020a). "Comparison of Graph-based Model Transformation Rules". In: *Journal of Object Technology* 19.2, 3:1–21 (cit. on pp. 158, 188).
Schultheiß, Alexander et al. (2020b). "On the use of product-line variants as experimental subjects for clone-and-own research: a case study". In: *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A.* ACM, 27:1–27:6 (cit. on pp. 158, 188).
Selic, B. (2003). "The pragmatics of model-driven development". In: *IEEE Software* 20.5, pp. 19–25. DOI: 10.1109/MS.2003.1231146 (cit. on pp. 2, 63, 113).
Selim, Gehan MK et al. (2017). "How is ATL really used? Language feature use in the ATL zoo". In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS).* DOI: 10.1109/MODELS.2017.20 (cit. on pp. 140, 151).
Sendall, S. et al. (2003). "Model transformation: the heart and soul of model-driven software development". In: *IEEE Software.* DOI: 10.1109/MS.2003.1231150 (cit. on pp. 1, 2, 20, 34, 62, 64, 110, 113, 140, 158).
Shaw, M. (2003). "Writing good software engineering research papers". In: *25th International Conference on Software Engineering, 2003. Proceedings.* DOI: 10.1109/ICSE.2003.1201262 (cit. on p. 52).
Shevtsov, S. et al. (2018). "Control-Theoretical Software Adaptation: A Systematic Literature Review". In: *IEEE Transactions on Software Engineering.* DOI: 10.1109/TSE.2017.2704579 (cit. on pp. 40, 43).
Singh, Yogesh et al. (2007). "Application of Logistic Regression and Artificial Neural Network for Predicting Software Quality Models." In: *Software engineering research and practice*, pp. 664–670 (cit. on p. 189).
Sjoberg, D. I. K. et al. (2002). "Conducting realistic experiments in software engineering". In: *Proceedings International Symposium on Empirical Software Engineering.* ISESE '02. DOI: 10.1109/ISESE.2002.1166921 (cit. on p. 54).
Somasundaram, Ramanathan et al. (2003). "Research philosophies in the IOS adoption field". In: *ECIS 2003 proceedings*, p. 53 (cit. on p. 39).

Sprinkle, J. et al. (2009). "Guest Editors' Introduction: What Kinds of Nails Need a Domain-Specific Hammer?" In: *IEEE Software* 26.4, pp. 15–18. DOI: 10.1109/MS.2009.92 (cit. on pp. 3, 64, 113).

Stachowiak, Herbert (1973). *Allgemeine Modelltheorie*. Springer. ISBN: ISBN 3-211-81106-0 (cit. on p. 2).

Staron, Miroslaw (2006). "Adopting Model Driven Software Development in Industry – A Case Study at Two Companies". In: *Model Driven Engineering Languages and Systems*. MODELS 2006. DOI: 10.1007/11880240_5 (cit. on pp. 10, 100, 105).

Stegmaier, Michael et al. (2019). "Insights for Improving Diagram Editing Gained from an Empirical Study". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 405–412. DOI: 10.1109/MODELS-C.2019.00063 (cit. on p. 102).

Steinberg, Dave et al. (2008). *EMF: eclipse modeling framework*. Pearson Education (cit. on pp. 2, 63, 113, 161).

Stol, Klaas-Jan et al. (2016). "Grounded Theory in Software Engineering Research: A Critical Review and Guidelines". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: Association for Computing Machinery, pp. 120–131. ISBN: 9781450339001. DOI: 10.1145/2884781.2884833. URL: https://doi.org/10.1145/2884781.2884833 (cit. on pp. 74, 77).

Strüber, Daniel et al. (2016). "Comparing Reuse Mechanisms for Model Transformation Languages: Design for an Empirical Study." In: *HuFaMo@ MoDELS*. Citeseer, pp. 27–32 (cit. on pp. 11, 105, 135).

Strüber, Daniel et al. (2017). "Henshin: A Usability-Focused Framework for EMF Model Transformation Development". In: *Graph Transformation*. ICGT 2017. DOI: 10.1007/978-3-319-61470-0_12 (cit. on pp. 162, 191).

Tehrani, Sobhan Yassipour et al. (2016). "Requirements engineering in model-transformation development: An interview-based study". In: *International Conference on Theory and Practice of Model Transformations*. Springer, pp. 123–137. DOI: 10.1007/978-3-319-42064-6_9 (cit. on pp. 10, 11, 105).

Tolosa, José Barranquero et al. (2011). "Towards the systematic measurement of ATL transformation models". In: *Software: Practice and Experience*. DOI: https://doi.org/10.1002/spe.1033 (cit. on pp. 11, 143, 152).

Torchiano, Marco et al. (2017). "Lessons Learnt in Conducting Survey Research". In: *2017 IEEE/ACM 5th International Workshop on Conducting Empirical Studies in Industry (CESI)*, pp. 33–39. DOI: 10.1109/CESI.2017.5 (cit. on p. 12).

Tratt, Laurence (2005). "Model transformations and tool integration". In: *Software & Systems Modeling*. DOI: 10.1007/s10270-004-0070-1 (cit. on pp. 1, 34).

Troya, Javier et al. (2022). "Model Transformation Testing and Debugging: A Survey". In: *ACM Computing Surveys (CSUR)* (cit. on p. 101).

Van Deursen, Arie et al. (2000). "Domain-specific languages: An annotated bibliography". In: *ACM Sigplan Notices*. DOI: 10.1145/352029.352035 (cit. on pp. 10, 57).

Van Deursen, Arie et al. (2002). "Domain-specific language design requires feature descriptions". In: *Journal of Computing and Information Technology*. DOI: 10.2498/cit.2002.01.01 (cit. on pp. 2, 35, 64, 113).

Varró, Dániel et al. (2006). "Termination Analysis of Model Transformations by Petri Nets". In: *Graph Transformations*. ICGT 2006. DOI: 10.1007/11841883_19 (cit. on p. 44).

Vignaga, Andrés (2009). "Metrics for measuring ATL model transformations". In: *MaTE, Department of Computer Science, Universidad de Chile, Tech. Rep* (cit. on pp. 143, 152, 193).

Vollstedt, Maike et al. (2019). "An Introduction to Grounded Theory with a Special Focus on Axial Coding and the Coding Paradigm". In: *Compendium for Early Career Researchers in Mathematics Education*. Ed. by Gabriele Kaiser et al. Cham: Springer International Publishing, pp. 81–100. DOI: 10.1007/978-3-030-15636-7_4 (cit. on p. 74).

W3C (2021). *Cascading Style Sheets (CSS)*. URL: https://www.w3.org/TR/css-2021/ (cit. on p. 3).

Weiber, Rolf et al. (2021). *Strukturgleichungsmodellierung: Eine anwendungsorientierte Einführung in die Kausalanalyse mit Hilfe von AMOS, SmartPLS und SPSS*. 3rd ed. Springer-Verlag. DOI: 10.1007/978-3-658-32660-9 (cit. on pp. 12, 94, 99, 117–120, 122, 134).

Weidmann, Nils et al. (2019). "Incremental (Unidirectional) Model Transformation with eMoflon:: IBeX". In: *Graph Transformation*. Ed. by Esther Guerra et al. Cham: Springer International Publishing, pp. 131–140. ISBN: 978-3-030-23611-3 (cit. on pp. 177, 191).

Weyns, Danny et al. (2012). "Claims and Supporting Evidence for Self-adaptive Systems: A Literature Study". In: *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '12. DOI: 10.1109/SEAMS.2012.6224395 (cit. on pp. 40, 41).

Whittle, Jon et al. (2013). "Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?" In: *Model-Driven Engineering Languages and Systems*. MODELS 2013. DOI: 10.1007/978-3-642-41533-3_1 (cit. on pp. 10, 25, 49, 95, 97, 105).

Wieringa, Roelf J. (2014). *Design science methodology for information systems and software engineering*. Undefined. 10.1007/978-3-662-43839-8. Springer. ISBN: 978-3-662-43838-1. DOI: 10.1007/978-3-662-43839-8 (cit. on pp. 159, 164).

Wiger, Ulf et al. (2001). *Four-fold Increase in Productivity and Quality - Industrial-Strength Functional Programming in Telecom-Class Products* (cit. on p. 55).

Wimmer, Manuel et al. (2009). "Reviving QVT Relations: Model-Based Debugging Using Colored Petri Nets". In: *Model Driven Engineering Languages and Systems*. Ed. by Andy Schürr et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 727–732. ISBN: 978-3-642-04425-0 (cit. on p. 101).

Wohlin, Claes (2014). "Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering". In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE '14. Association for Computing Machinery. DOI: 10.1145/2601248.2601268 (cit. on p. 39).

Zündorf, Albert et al. (2013). "Story Driven Modeling Libary (SDMLib): an Inline DSL for modeling and model transformations, the Petrinet-Statechart case". In: *Sixth Transformation Tool Contest (TTC 2013), ser. EPTCS* (cit. on p. 191).

Appendix A

Appendix - Paper A

A.1 SLR results

P2 Patzina, Sven et al. (2012). "A Case Study Based Comparison of ATL and SDM". In: *Proceedings of the 4th International Conference on Applications of Graph Transformations with Industrial Relevance*. AGTIVE 2011. DOI: 10.1007/978-3-642-34176-2_18.

P3 Stephan, Matthew et al. (2009). *A Comparative Look at Model Transformation Languages*. Tech. rep. Software Technology Laboratory at Queens University. DOI: 10.1.1.712.2983.

P4 Cuadrado, J. S. et al. (2014). "A Component Model for Model Transformations". In: *IEEE Transactions on Software Engineering*. DOI: 10.1109/TSE.2014.2339852.

P9 Agrawal, Aditya et al. (2003). "A UML-based graph transformation approach for implementing domain-specific model transformations". In: *International Journal on Software and Systems Modeling*. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.152.1226.

P10 Johannes, Jendrik et al. (2009). "Abstracting Complex Languages through Transformation and Composition". In: *Model Driven Engineering Languages and Systems*. MODELS 2009. DOI: 10.1007/978-3-642-04425-0_41 (cit. on pp. 62, 110).

P15 Jouault, Frédéric et al. (2008). "ATL: A model transformation tool". In: *Science of Computer Programming*. DOI: 10.1016/j.scico.2007.08.002 (cit. on pp. 20, 140, 158).

P21 Giese, Holger et al. (2014). "Bridging the gap between formal semantics and implementation of triple graph grammars". In: *Software & Systems Modeling*. DOI: 10.1007/s10270-012-0247-y.

P22 Schoenboeck, Johannes et al. (2010). "Catch Me If You Can – Debugging Support for Model Transformations". In: *Models in Software Engineering*. MODELS 2009. DOI: 10.1007/978-3-642-12261-3_2.

P23 Hinkel, Georg et al. (2019a). "Change propagation and bidirectionality in internal transformation DSLs". In: *Software & Systems Modeling*. DOI: 10.1007/s10270-017-0617-6 (cit. on pp. 5, 47, 55, 64, 113, 114, 190, 191, 193).

P27 Sottet, J. et al. (2014). "Defining Domain Specific Transformations in Human-Com- puter interfaces development". In: *2014 2nd International Conference on Model-Driven Engineering and Software Development*. MODELSWARD '14. URL: https://ieeexplore.ieee.org/abstract/document/7018471.

P28 Acretoaie, Vlad (2013). *Delivering the Next Generation of Model Transformation Languages and Tools*. DOI: 10.1.1.708.6612.

P29 Rentschler, Andreas et al. (2014). "Designing Information Hiding Modularity for Model Transformation Languages". In: *Proceedings of the 13th International Conference on Modularity*. MODULARITY '14. DOI: 10.1145/2577080.2577094 (cit. on p. 159).

P30 Steel, Jim et al. (2011). "Domain-Specific Model Transformation in Building Quantity Take-Off". In: *Model Driven Engineering Languages and Systems*. MODELS 2011. DOI: 10.1007/978-3-642-24485-8_15.

P32 Shin, Shin-Shing (2019). "Empirical study on the effectiveness and efficiency of model-driven architecture techniques". In: *Software & Systems Modeling*. DOI: 10.1007/s10270-018-00711-y.

P33 Criado, Javier et al. (2015). "Enabling the Reuse of Stored Model Transformations Through Annotations". In: *Theory and Practice of Model Transformations*. ICMT 2015. DOI: 10.1007/978-3-319-21155-8_4.

P34 Rose, Louis M. et al. (2014). "Epsilon Flock: a model migration language". In: *Software & Systems Modeling*. DOI: 10.1007/s10270-012-0296-2.

P39 Berramla, K. et al. (2015). "Formal validation of model transformation with Coq proof assistant". In: *2015 First International Conference on New Technologies of Information and Communication*. NTIC 2015. DOI: 10.1109/NTIC.2015.7368755.

P40 Legros, Elodie et al. (2009). "Generic and reflective graph transformations for checking and enforcement of modeling guidelines". In: *Journal of Visual Languages & Computing* 4. DOI: 10.1016/j.jvlc.2009.04.005.

P41 Sánchez Cuadrado, Jesús et al. (2011). "Generic Model Transformations: Write Once, Reuse Everywhere". In: *Theory and Practice of Model Transformations*. ICMT 2011. DOI: 10.1007/978-3-642-21732-6_5.

P43 Strüber, Daniel et al. (2017). "Henshin: A Usability-Focused Framework for EMF Model Transformation Development". In: *Graph Transformation*. ICGT 2017. DOI: 10.1007/978-3-319-61470-0_12 (cit. on pp. 162, 191).

P44 Wider, Arif (2014). "Implementing a Bidirectional Model Transformation Language as an Internal DSL in Scala". In: *EDBT/ICDT Workshops*. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.428.9439.

P45 Lawley, Michael et al. (2007). "Implementing a Practical Declarative Logic-based Model Transformation Engine". In: *Proceedings of the 2007 ACM Symposium on Applied Computing*. SAC '07. DOI: 10.1145/1244002.1244216 (cit. on pp. 20, 62, 140).

P50 Liepiņš, Renārs (2012). "Library for Model Querying: IQuery". In: *Proceedings of the 12th Workshop on OCL and Textual Modelling*. OCL '12. DOI: 10.1145/2428516.2428522 (cit. on p. 62).

P52 Krikava, Filip et al. (2014). "Manipulating Models Using Internal Domain-Specific Languages". In: *Symposium On Applied Computing*. SAC '14. DOI: 10.1145/2554850.2555127 (cit. on pp. 20, 140, 158).

P56 Sun, Yu et al. (2009). "Model Transformation by Demonstration". In: *Model Driven Engineering Languages and Systems*. MODELS 2009. DOI: 10.1007/978-3-642-04425-0_58.

P58 Irazábal, Jerónimo et al. (2010). "Model Transformation Languages Relying on Models as ADTs". In: *Software Language Engineering*. SLE 2009. DOI: 10.1007/978-3-642-12107-4_10.

P59 Hebig, Regina et al. (2018). "Model Transformation Languages Under a Magnifying Glass: A Controlled Experiment with Xtend, ATL, and QVT". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. DOI: 10.1145/3236024.3236046 (cit. on pp. 11, 20, 25, 37, 98, 104, 110, 135, 140, 158, 192).

P60 Lara, Juan de et al. (2018). "Model Transformation Product Lines". In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '18. DOI: 10.1145/3239372.3239377.

P63 Sendall, S. et al. (2003). "Model transformation: the heart and soul of model-driven software development". In: *IEEE Software*. DOI: 10.1109/MS.2003.1231150 (cit. on pp. 1, 2, 20, 34, 62, 64, 110, 113, 140, 158).

P64 Tratt, Laurence (2005). "Model transformations and tool integration". In: *Software & Systems Modeling*. DOI: 10.1007/s10270-004-0070-1 (cit. on pp. 1, 34).

P66 — (2007). "Model transformations in MT". In: *Science of Computer Programming*. DOI: 10.1016/j.scico.2007.05.003.

P70 Baar, Thomas et al. (2007). "On the Usage of Concrete Syntax in Model Transformation Rules". In: *Perspectives of Systems Informatics*. PSI 2006. DOI: 10.1007/978-3-540-70881-0_10.

P74 Sánchez Cuadrado, J. et al. (2015). "Quick fixing ATL model transformations". In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems*. MODELS '15. DOI: 10.1109/MODELS.2015.7338245.

P75 Li, Dan et al. (2011). "QVT-based Model Transformation Using XSLT". In: *SIGSOFT Softw. Eng. Notes*. DOI: 10.1145/1921532.1921563.

P77 Kusel, A. et al. (2015). "Reuse in model-to-model transformation languages: are we there yet?" In: *Software & Systems Modeling*. DOI: 10.1007/s10270-013-0343-7 (cit. on p. 132).

P78 Wimmer, Manuel et al. (2011). "Reusing Model Transformations across Heterogeneous Metamodels". In: *ECEASST*. DOI: 10.14279/tuj.eceasst.50.722.

P80 Křikava, Filip et al. (2014). "SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations". In: *Model-Driven Engineering Languages and Systems*. MODELS 2014. DOI: 10.1007/978-3-319-11653-2_35 (cit. on pp. 47, 55).

P81 Akehurst, D. H. et al. (2006). "SiTra: Simple Transformations in Java". In: *Model Driven Engineering Languages and Systems*. MODELS 2006. DOI: 10.1007/11880240_25 (cit. on pp. 190, 193).

P86 Kolovos, Dimitrios S. et al. (2008). "The Epsilon Transformation Language". In: *Theory and Practice of Model Transformations*. ICMT 2008. DOI: 10.1007/978-3-540-69927-9_4 (cit. on pp. 35, 64, 113, 140).

P90 Sánchez Cuadrado, Jesús et al. (2014). "Towards the Systematic Construction of Domain-Specific Transformation Languages". In: *Modelling Foundations and Applications*. ECMFA 2014. DOI: 10.1007/978-3-319-09195-2_13.

P94 George, Lars et al. (2012). "Type-Safe Model Transformation Languages as Internal DSLs in Scala". In: *Theory and Practice of Model Transformations*. ICMT 2012. DOI: 10.1007/978-3-642-30476-7_11 (cit. on pp. 64, 113).

P95 Hinkel, Georg et al. (2019b). "Using internal domain-specific languages to inherit tool support and modularity for model transformations". In: *Software & Systems Modeling*. DOI: 10.1007/s10270-017-0578-9 (cit. on pp. 20, 37, 62, 102, 140, 190).

P664 Agrawal, Aditya et al. (2002). "Generative programming via graph transformations in the model-driven architecture". In: *In OOPSLA 2002 Workshop in Generative Techniques in the context of Model Driven Architecture*. OOPSLA '02. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.70.4824.

P665 Aßmann, Uwe (1996). "How to uniformly specify program analysis and transformation with graph rewrite systems". In: *Compiler Construction*. CC 1996. DOI: 10.1007/3-540-61053-7_57.

P667 Radermacher, Ansgar (2000). "Support for Design Patterns through Graph Transformation Tools". In: *Applications of Graph Transformations with Industrial Relevance*. AGTIVE 1999. DOI: 10.1007/3-540-45104-8_9.

P669 Mohagheghi, Parastoo et al. (2013a). "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases". In: *Empirical Software Engineering*. DOI: 10.1007/s10664-012-9196-x (cit. on pp. 10, 55, 95, 100, 106).

P670 Staron, Miroslaw (2006). "Adopting Model Driven Software Development in Industry – A Case Study at Two Companies". In: *Model Driven Engineering Languages and Systems*. MODELS 2006. DOI: 10.1007/11880240_5 (cit. on pp. 10, 100, 105).

P671 Panach, José Ignacio et al. (2011). "A Model for Dealing with Usability in a Holistic MDD Method". In: *User Interface Description Language, Lisbon, Portugal*. UIDL '11.

P672 Amelunxen, Carsten et al. (2008). "Checking and Enforcement of Modeling Guidelines with Graph Transformations". In: *Applications of Graph Transformations with Industrial Relevance*. AGTIVE 2007. DOI: 10.1007/978-3-540-89020-1_22.

P673 Schmidt, Douglas (2006). "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer - IEEE Computer Society*. DOI: 10.1109/MC.2006.58 (cit. on pp. 2, 62, 63, 110, 113, 140).

P674 Chafi, Hassan et al. (2010). "Language Virtualization for Heterogeneous Parallel Computing". In: *ACM Sigplan Notices*. DOI: 10.1145/1932682.1869527.

P675 Mernik, Marjan et al. (2005). "When and How to Develop Domain-specific Languages". In: *ACM computing surveys (CSUR)*. DOI: 10.1145/1118890.1118892 (cit. on pp. 1, 34, 47, 48).

P676 Gorp, Pieter Van et al. (2013). *The Petri-Nets to Statecharts Transformation Case*. DOI: 10.4204/EPTCS.135.3.

P677 Mens, Tom et al. (2006). "A Taxonomy of Model Transformation". In: *Electronic Notes in Theoretical Computer Science (GraMoT 2005)*. DOI: 10.1016/j.entcs.2005.10.021 (cit. on pp. 3, 10, 35, 56, 64, 114).

P800 Herndon, R. M. et al. (1988). "The realizable benefits of a language prototyping language". In: *IEEE Transactions on Software Engineering*. DOI: 10.1109/32.6159.

P801 Batory, Don et al. (1994). "Reengineering a Complex Application Using a Scalable Data Structure Compiler". In: *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT '94. DOI: 10.1145/193173.195299.

P803 Kieburtz, Richard B. et al. (1996). "A Software Engineering Experiment in Software Component Generation". In: *Proceedings of the 18th International Conference on Software Engineering*. ICSE'96. DOI: 10.1109/ICSE.1996.493448 (cit. on p. 42).

P804 Gray, J. et al. (2003). "An examination of DSLs for concisely representing model traversals and transformations". In: *36th Annual Hawaii International Conference on System Sciences,*

2003. Proceedings of the. HICSS '03. DOI: 10.1109/HICSS.2003.1174892 (cit. on pp. 20, 140, 158).

A.2 Overview over all extracted claims

See Table A.1.

Table A.1: Overview over claims per category.

Category	Valuation	CID	Claim	Publication	Evidence
Analysability	positive	C1	Declarative MTLs lend themselves to automatic analysis.	**P45**	-
Comprehensibility	positive	C2	Based on user feedback it was identified that visual syntax is beneficial when reading a transformation program.	**P43**	Experience
		C3	Bidirectional transformation languages have an advantage in comprehensibility.	**P44**	-
		C4	Rule written in a declarative MTL are more easily understood in isolation and in combination.	**P45**	-
		C5	An observation made from the empirical data is, that context selection and identification is easier for subjects working with MTLs than with GPLs.	**P59**	Empirical study
		C6	There are perceived cognitive gains of graphical representations compared to fully textual representations of transformations shown by for example the appeal of UMLs graphical representation of models.	**P63**	-
		C7	Model transformation languages incorporate high-level abstractions that make them more understandable than GPLs.	**P95**	-
	negative	C8	Comprehensibility of transformation logic is hampered as current transformation languages provide only a limited view on a transformation problem. For example graph transformation approaches only reveal parts of the meta-model.	**P22**	-
		C9	Most MTLs lack convenient facilities for understanding the transformation logic.	**P22**	-
		C10	Model transformation languages require specific skills and as a result are hard to understand for many stakeholders.	**P27**	-
		C11	Large and heterogeneous models lead to poorly understandable transformation code due to missing language concepts to master complexity.	**P29**	-
		C12	Graph transformations defined on abstract syntax are hard to read because the user has to be familiar with meta-model that defines the abstract syntax.	**P70**	-
		C13	Purely graph based transformation languages can become complex and hard to read.	**P70**	-
Conciseness	positive	C14	General purpose languages lack simplicity because of how transformations are defined.	**P3**	Examples
		C15	GPLs do not allow developers to conveniently express model manipulation concepts and the loss of abstraction in GPLs may give rise to accidental complexities.	**P52**	Cites **P673**
		C16	Transformations implemented in the pre-study using rule-based MTLs were up ot 48% smaller than corresponding Java variants.	**P59**	Preliminary study
		C17	Declarative approaches make language more concise.	**P63**	-

Table A.1 – *continued from previous page*

Category	Valuation	CID	Claim	Publication	Evidence
Conciseness	positive	C18	Graphical notation in MTLs is concise.	**P75**	-
		C19	GPLs do not conveniently express model manipulation concepts and the loss of abstraction can give rise to accidental complexities.	**P80**	Cites **673**
		C20	Model transformation languages incorporate high-level abstractions that make them more concise than GPLs.	**P95**	-
		C21	Model transformation languages are more concise.	**P95**	-
		C22	MDE and model transformation languages such as QVT help to reduce complexity.	**P673**	-
Debugging	positive	C23	Debuggers for MTLs are likely better than those for GPLs for debugging transformations since it is questionable whether the call stacks produced by debuggers of GPLs are meaningful for the developer.	**P95**	-
	negative	C24	Although numerous transformation languages exist, they lack convenient facilities for supporting debugging and understanding of the transformation logic.	**P22**	-
		C25	In ATL, TGGs and QVT-R correspondence is defined on a higher level of abstraction compared to on what execution engines operate. Thus debugging is limited on the lower level of the execution engines not on the level of the language definition.	**P22**	-
		C26	In declarative model transformation languages debugging is more difficult than in imperative ones.	**P45**	-
		C27	Model transformation languages lack proper debugging support since implementation cost is high.	**P90**	-
Ease of writing a transformation	positive	C28	We found graphical rule definition far more intuitive than syntax-based definition.	**P3**	Experience
		C29	Model transformation languages ease development efforts by offering succinct syntax to query from and map model elements between different modelling domains.	**P29**	-
		C30	Model transformation languages make it easy to work with models.	**P50**	-
		C31	Imperative transformation approaches offer a familiar paradigm, that is, sequence, selection, and iteration.	**P63**	-
		C32	It is our impression that, in general, graph transformations offer significantly better support for the specification and implementation of modelling guidelines and refactorings.	**P672**	Experience
	negative	C33	Imperative MTLs induce overhead code because many issues have to be accomplished explicitly, e.g., specification of control flow.	**P22**	-
		C34	Traditional transformation languages require specific skills to be able to write transformations.	**P27**	-

Table A.1 – *continued from previous page*

Category	Valuation	CID	Claim	Publication	Evidence
Ease of writing a transformation	negative	C35	To be able to write transformations one has to be a transformation expert.	**P28**	-
		C36	Based on user feedback we identified that writing a transformation program with a graphical syntax can be complicated.	**P43**	Experience
		C37	The syntax of declarative MTLs is unfamiliar for many developers.	**P45**	-
		C38	Model transformation languages that define transformations on meta-model level require deep understanding of the meta-model.	**P56**	-
		C39	There is no sufficient (statistically significant) evidence of a general advantage of specialized model transformation languages (ATL, QVT-O) over a modern GPL (Xtend).	**P59**	Empirical study
		C40	Developers are generally more comfortable with encoding complicated (transformation) algorithms in procedural languages.	**P63**	-
		C41	First of all some of us are not convinced that the usage of a visual notation has significant advantages compared to a textual notation. A textual notation is more compact, simplifies all kinds of version and configuration management tasks, and does not force its users to spend hours beautifying the layout of huge diagrams.	**P672**	-
Expressiveness	positive	C42	Rule-based approaches seems to be less error-prone compared to a manual implementation of pattern matching for each transformation in a general-purpose language.	**P2**	-
		C43	Model transformation languages can hide details like traversing behind simple syntax.	**P15**	-
		C44	Model transformation languages can hide traces behind simple syntax.	**P15**	-
		C45	Model transformation languages can hide rule triggering behind simple syntax.	**P15**	-
		C46	Model transformation languages can hide rule ordering behind simple syntax.	**P15**	-
		C47	Model transformation languages can hide complex transformation algorithms behind a simple syntax.	**P15**	-
		C48	Model transformation languages hide transformation complexity and burden from user.	**P27**	Cites **P671**
		C49	Graph transformations generally offer a significantly better support for the specification and implementation of modelling guidelines and refactorings.	**P40**	Cites **P672**
		C50	Declarative MTLs allow automatic traceability management.	**P45**	Cites **P677**
		C51	Declarative model transformation languages allow for implicit rule ordering lessening the load on developer.	**P45**	Cites **P677**
		C52	Declarative MTLs can do implicit target object creation.	**P45**	Cites **P677**
		C53	Declarative MTLs allow for implicit source model traversal.	**P45**	Cites **P677**
		C54	Model transformation languages syntax is more specific.	**P52**	-
		C55	GPLs do not allow developers to conveniently express model manipulation concepts.	**P52**	-

Table A.1 – *continued from previous page*

Category	Valuation	CID	Claim	Publication	Evidence
Expressiveness	positive	C56	We found that copying complex structures is more effective in MTLs.	**P59**	Empirical study
		C57	General purpose languages lack suitable abstractions for specifying transformations.	**P63**	-
		C58	Graph based MTLs are especially popular due to their high expressive power.	**P70**	-
		C59	Model transformation languages have more specific language constructs.	**P80**	-
		C60	Model transformation languages have a higher level of abstraction which leads to gains in expressiveness over GPLs.	**P80**	Cites **P675**
		C61	Model transformation languages are easier to use than GPLs.	**P80**	Cites **P675**
		C62	Model transformation languages transformation constructs are more specific.	**P94**	-
		C63	From our perspective, automatic handling/resolution of traces by transformation engine is one of the major features that make existing MTLs better suited for model transformations than GPLs.	**P95**	-
		C64	General purpose languages lack sufficient transformation concepts.	**P95**	-
		C1D	DSLs trade expressiveness in a limited domain for generality.	**P675**	Cites **P804**
		C65	GPLs lack suitable abstractions for specifying transformations.	**P677**	Cites **P63**
		C66	With a DSL/MTL a programmer can express their objective in a concise manner using a language that is much higher in expressiveness than that typically offered in a transitional programming language.	**P804**	-
	negative	C67	Having written several transformation we have identified that current MTLs are too low a level of abstraction for succinctly expressing transformations between DSLs because they demonstrate several recurring patterns that have to be reimplemented each time.	**P10**	Experience
		C68	Having written several transformation we have identified that mapping a single element to fragments of multiple elements has to be done programmatically which is counter intuitive and error-prone.	**P10**	Experience
		C69	OCL constraints cannot be transformed in MTLs.	**P32**	Empirical Study
		C70	There is no mechanism for describing and/or storing information about the properties of a transformation.	**P33**	-
Extendability	negative	C71	Extending model transformation languages is difficult.	**P50**	-
Just better	positive	C72	GGT (graph gammar and graph transformation) are a powerful technique for specifying complex transformations.	**P9**	Cites **P664-P666**
		C73	General purpose programming languages are not suitable for defining model transformations.	**P23**	Cites **P63**

Table A.1 – *continued from previous page*

Category	Valuation	CID	Claim	Publication	Evidence
Just better	positive	C74	GPLs are not well-suited for model migration.	**P34**	Examples
		C75	Dedicated MTLs offer the most potential transformation approach because the languages can be tailored for the purpose	**P63**	-
		C76	In order to transform models in a GPL one has to add increasing amounts of machinery e.g. to keep track of which elements have already been transformed. This leads to the assumption that model transformations cannot be sensibly written in a standard programming language.	**P64**	Examples
		C77	Model transformations present a number of problems which imply that dedicated approaches are required.	**P66**	Cites **P64**
		C78	The current consensus is that specialized languages with a mixture of declarative and imperative constructs are most suitable for specifying model transformations.	**P86**	-
		C79	With the help of an example we have shown that GGT (graph gammar and graph transformation) can be used to transform PIMs into PSMs.	**P664**	Examples
		C2D	DSLs open up the application domain to a larger group of developers	**P675**	Cites **P803**
		C3D	Domain specific languages increase the ease of use.	**P675**	Cites **P803**
		C4D	When using DSLs less errors are made.	**P803**	Empirical Study
	negative	C80	General purpose MTLs are not well suited for model migration since there is additional overhead but dedicated migration languages are.	**P34**	Examples
Learnability	negative	C81	The generality of General purpose MTLs can have the effect of making them less approachable and create a steep learning curve for non-expert users.	**P30**	-
		C82	Users have to learn multiple similar, but not always consistent, languages, which requires considerable time to learn.	**P52**	-
		C83	Model transformation languages have a steep learning curve.	**P58**	-
		C84	One has to learn a completely new language to transform models with MTLs.	**P81**	-
Performance	positive	C85	Model transformation languages are more performant.	**P95**	Cites **P676**
		C86	GrGen shows a better performance of transformations than Java.	**P676**	Samples
		C5D	DSLs have better performance.	**P801**	-
	negative	C87	Declarative MTLs have performance problems.	**P45**	-
		C88	The performance of model transformation languages is a shortcoming that may make users feel limited.	**P52**	-
		C89	MTLs have worse performance.	**P80**	-

Table A.1 – *continued from previous page*

Category	Valuation	CID	Claim	Publication	Evidence
Productivity	positive	C90	Model transformation languages being DSLs improve the productivity.	**P29**	-
		C91	Declarative MTLs increase programmer productivity.	**P45**	-
		C6D	DSLs increase productivity.	**P675**	Cites **P801**, **P803**
		C7D	Using DSLs increases productivity.	**P801**	Examples
		C8D	Using DSLs increases productivity.	**P803**	Empirical Study
	negative	C92	The perceived effectiveness of model transformation languages is bad.	**P32**	Empirical Study
		C93	Productivity of GPL development might be higher since expert users for GPLs are easier to hire.	**P59**	-
Reuse and Maintainability	positive	C94	Bidirectional model transformations have an advantage in maintainability	**P44**	-
		C95	There exists a plethora of reuse mechanisms for MTLs.	**P77**	Literature review
		C9D	Domain specific languages reduce the maintenance costs.	**P675**	Cites **P800**
	negative	C96	Reuse is sparse, transformations are written from scratch every time because meta-models differ slightly.	**P4**	Cites **P77**
		C97	Having written several transformation we have identified that recurring patterns have to be implemented from scratch every time.	**P10**	Experience
		C98	There exists no module concept for model transformation languages that allows programmers to control information hiding and strictly declare model and code dependencies at module interface.	**P29**	-
		C99	Model transformation languages lack sophisticated reuse mechanisms.	**P33**	-
		C100	Unfortunately the definition of model transformations is normally a type-centric activity, thus making their reuse for other meta-models difficult.	**P41**	-
		C101	Evolving and maintaining MTL requires effort.	**P52**	Cites **P674**
		C102	The emphasis of MDE on using DSLs has caused a proliferation of meta-models. In this scenario, developing a transformation for a new meta-model is usually performed manually with no reuse, even if comparable transformations for similar meta-models exist.	**P60**	-
		C103	There are barriers such as insufficient abstraction of reuse mechanisms from meta-models that hamper reuse.	**P77**	Literature review
		C104	There is little support for reusing model transformations in different contexts since they are tightly coupled to the meta-models they are defined upon.	**P78**	-
		C105	Reuse of model transformations is hardly established in practice.	**P95**	Cites **P77**

Table A.1 – *continued from previous page*

Category	Valuation	CID	Claim	Publication	Evidence
R & M	negative	C106	Developing these new languages to a sufficient degree of maturity is an enormous effort which includes for example construction and optimisation of compilers.	**P674**	-
Semantics and Verification	positive	C107	Bidirectional transformation languages have an advantage in verification.	**P44**	-
	negative	C108	For existing relational model transformation approaches, it is usually not really clear under which constraints particular implementations really conform to the formal semantics.	**P21**	-
		C109	Comprehensive verification support of model transformations is missing.	**P22**	-
		C110	There is a semantic difference between a typical programming language and formalisms that support bi-directionality and change propagation such as TGGs.	**P23**	-
		C111	Most transformation languages have no formal semantics to add detailed specifications on the expected behaviour.	**P39**	-
		C112	The semantics of many model transformation languages is not formally defined.	**P58**	-
Tool Support	positive	C113	Internal MTLs can inherit tool support of general purpose host language.	**P23**	-
		C114	Tool suport for external transformation languages is potentially more powerful than for internal MTL or GPL because it can be tailored to the DSL.	**P44**	-
		C115	Declarative MTLs provide opportunities for specialized tool support.	**P45**	-
	negative	C116	Model transformation languages lack tool support.	**P23**	Cites **P669**,**P670**
		C117	Declarative MTLs lack libraries and tool support.	**P45**	-
		C118	Model transformation languages lack tool support.	**P52**	-
		C119	Supporting tools for MTLs have not the same level of maturity as for GPLs.	**P74**	-
		C120	Model transformation languages have worse tool support.	**P80**	Cites **P94**
		C121	Tool support for external MTLs has to be developed which entails extra effort.	**P94**	-
		C122	Tool support for model transformations is not as mature as subjects would like.	**P669**	Empirical study
		C123	Tool support for model transformations is not great.	**P670**	Empirical study
Versatility	negative	C124	The syntax of model transformation languages is less versatile.	**P52**	-
		C125	Model transformation languages are less versatile than GPLs.	**P80**	-
		C126	Model transformation languages have less versatile language constructs.	**P80**	-
		C127	Model transformation languages constructs are less versatile.	**P94**	-
		C10D	DSLs are less general than general purpose programming languages	**P675**	-

Appendix B

Appendix - Paper B

B.1 Interview Questions

Demographic Questions

- In what context have you used model transformation languages? Research, industrial projects or other?
- How much experience do you have in using model transformation languages? Rough estimate in years is sufficient.
- What model transformation languages have you used to date?

Question Set 1

Ease of Writing

The use of MTLs increases the ease of writing model transformations.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Model transformation languages ease development efforts by offering succinct syntax to query from and map model elements between different modelling domains.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Model transformation languages require specific skills to be able to write model transformations.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Comprehensibility

The use of MTLs increases the comprehensibility of model transformations.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Model transformation languages incorporate high-level abstractions that make them more understandable than GPLs.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?

- What is the reasoning behind your answer?

Most MTLs lack convenient facilities for understanding the transformation logic.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Question Set 2

Tool Support

There is sufficient tool support for the use of MTLs for writing model transformations.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Tool support for external transformation languages is potentially more powerful than for internal MTL or GPL because it can be tailored to the DSL.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Model transformation languages lack tool support.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Productivity

The use of MTLs increases the productivity of writing model transformations.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Model transformation languages, being DSLs, improve the productivity.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Productivity of GPL development might be higher since expert users for GPLs are easier to hire.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Question Set 3

Reuseability & Maintainability

The use of MTLs increases the reusability and maintainability of model transformations.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Bidirectional model transformations have an advantage in maintainability.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Model transformation languages lack sophisticated reuse mechanisms.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Expressiveness

The use of MTLs increases the expressiveness of model transformations.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Model transformation languages hide transformation complexity and burden from the user.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Having written several transformations we have identified that current MTLs are too low a level of abstraction for succinctly expressing transformations between DSLs because they demonstrate several recurring patterns that have to be reimplemented each time.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

B.2 Mail Templates

Mail Template

Dear ${Author Name},

I'm a PhD student with Matthias Tichy at Ulm University. We recently conducted an SLR about the advantages and disadvantages of model transformation languages as claimed in literature. Our results have been published in the software and systems modelling journal here `http://dx.doi.org/10.1007/s10270-020-00815-4`. One of our main takeaways from the study was that a large portion of claims about model transformation languages is never substantiated. One main reason for this, we believe, is implicit knowledge authors tend to omit for different reasons.

Since you are an author of one of the publications we considered during our SLR it would be great to talk to you about your experiences and stance with regard to model transformation languages and the claims we extracted from literature. We would need max. 30 minutes of your time. The interview would be conducted by me via an online conferencing system.

In order to organize the interview dates I would like to ask you to chose a suitable date, under the following link `https://terminplaner4.dfn.de/F1mIEEwSSkTwh8XA`. Please note that the times are given in UTC. The password for the poll is "claims". Your response will not be visible to anyone other than myself. If none of the dates is suitable for you, you are welcome to contact me to find another date for the interview.

Before your interview I would like to ask you to agree to the data protection agreement under the following link `https://pmx.informatik.uni-ulm.de/limesurvey/index.php/924713?lang=en`. I have also attached a copy of how we handle the interview data to this mail.

Best regards
Stefan Götz

Reminder Mail Template

Dear ${Author Name},

If you already filled out our organization poll please ignore this mail.

I wanted to remind you to maybe take part in our interview study about the implicit knowledge of users with regards to advantages and disadvantages of model transformation languages. It would be great to talk to you about your experiences and stance with regard to model transformation languages and the claims we extracted from literature. We would need max. 30 minutes of your time.

In order to organize the interview dates I would like to ask you to chose a suitable date, under the following link `https://terminplaner4.dfn.de/F1mIEEwSSkTwh8XA`. Please note that the times are given in UTC. Please also note that you need to press the SAVE button at the right hand side of the poll. The password for the poll is "claims". Your response will not be visible to anyone other than myself. If none of the dates is suitable for you, you are welcome to contact me to find another date for the interview.

Before your interview I would like to ask you to agree to the data protection agreement under the following link `https://pmx.informatik.uni-ulm.de/limesurvey/index.php/924713?lang=en`. I have also attached a copy of how we handle the interview data to this mail.

Best regards
Stefan Götz

B.3 Demographics

See Table B.1.

Table B.1: Overview over the interviewee demographic data

PID	Background	Experience in years	Language types used for writing transformations
P1	Research	>10	GPLs
P2	Research	10-15	dedicated MTLs
P3	Research	8	dedicated MTLs
P4	Research	7	dedicated MTLs & internal MTLs
P5	Research	>5	dedicated MTLs & GPLs
P6	Research & Industry Projects	13	dedicated MTLs & GPLs
P7	Research & Industry Projects	10	dedicated MTLs & GPLs
P8	Research & Industry Projects	18	dedicated MTLs & GPLs
P9	Industry	20	dedicated MTLs
P10	Research	4	dedicated MTLs & GPLs
P11	Research	5-6	dedicated MTLs
P12	Research & Industry Projects	8	dedicated MTLs

Table B.1 – continued from previous page

PID	Background	Experience in years	Language types used for writing transformations
P13	Industry with History in Research	6	dedicated MTLs & internal MTLs
P14	Research & Industry Projects	15	dedicated MTLs & internal MTLs & GPLs
P15	Research & Industry Projects	5	dedicated MTLs & GPLs
P16	Research	7	dedicated MTLs & GPLs
P17	Research & Industry Projects	18	dedicated MTLs & GPLs
P18	Research & Industry Projects	10	dedicated MTLs & GPLs
P19	Research	7	dedicated MTLs
P20	Research & Industry Projects	3	dedicated MTLs & GPLs
P21	Research & Industry Projects	15	dedicated MTLs & GPLs
P22	Research & Industry Projects	8	dedicated MTLs & GPLs
P23	Research	13	dedicated MTLs
P24	Research & Industry Projects	15	dedicated MTLs & GPLs
P25	Research	8	dedicated MTLs
P26	Industry	>10	dedicated MTLs
P27	Industry with History in Research	10-12	dedicated MTLs & GPLs
P28	Research	15	dedicated MTLs & GPLs
P29	Research & Industry Projects	12	dedicated MTLs
P30	Research & Industry Projects	17	dedicated MTLs & GPLs
P31	Research	8	dedicated MTLs
P32	Research & Industry Projects	15	dedicated MTLs & GPLs
P33	Research	5-6	dedicated MTLs & GPLs
P34	Research	5-6	GPLs
P35	Research	10	dedicated MTLs
P36	Research	10	dedicated MTLs & GPLs
P37	Research & Industry Projects	10-11	dedicated MTLs
P38	Research	4-5	dedicated MTLs
P39	Industry	28	dedicated MTLs & GPLs
P40	Research	9	dedicated MTLs
P41	Research	7-8	dedicated MTLs
P42	Industry with History in Research	13	dedicated MTLs & internal MTLs & GPLs
P43	Research & Industry Projects	8-10	dedicated MTLs
P44	Research & Industry Projects	10	dedicated MTLs
P45	Research	1-2	dedicated MTLs
P46	Research & Industry Projects	9	dedicated MTLs
P47	Research	4	dedicated MTLs
P48	Research	7-8	dedicated MTLs & internal MTLs
P49	Research & Industry Projects	10	dedicated MTLs & GPLs
P50	Research	20	dedicated MTLs
P51	Research & Industry Projects	3	dedicated MTLs
P52	Research	13-14	dedicated MTLs
P53	Research	12	dedicated MTLs & GPLs
P54	Research	7	dedicated MTLs
P55	Research & Industry Projects	16	dedicated MTLs & GPLs
P56	Research	16	dedicated MTLs

B.4 Data Privacy Agreement

General information, declaration of consent

General information about the interviews about claims about model transformation languages of the Institute for Software Engineering and Programming Languages of Ulm University.

At the Institute for Software Engineering and Programming Languages of Ulm University model transformation languages are being examined. This includes claims about the advantages and disadvantages and evidence thereof. Within one work package of the doctoral thesis of Stefan Götz, researchers and practitioners are being surveyed about their opinions on certain claims.

The goal of this work package is to gain a deeper understanding of the reasoning people use for believing certain claims about model transformation languages.

1 Procedure

- If you decide to participate in our interview study, please fill out the poll at https://terminplaner4.dfn.de/F1mIEEwSSkTwh8XA so we can set up a date for the interview.
- We will contact you about one week in advance of the chosen date to arrange the interview.
- The interview will take place using an online conferencing tool hosted at Ulm University or via Skype depending on your preference.
- Please fill out the consent form at https://pmx.informatik.uni-ulm.de/limesurvey/index.php/924713 before the interview.
- At the beginning of the interview we will ask you questions about your experience level with regards to model transformation languages such as the languages you have used.
- During the interview we will show you different claims from literature about model transformation languages for which we would like you to tell us if you agree with them or not and your reasoning behind the decision.
- **An audio recording of the interview will be made if you consent to it.**
- The audio recordings will be transcribed after the interview and deleted as soon as transcribing has been completed.

2 Conditions of participation:

- You have some for of experience with model transformation languages.

3 Handling the data in the research project

1. Your interview will be recorded on audio and notes will be taken. The procedure described in 2-5 is followed for anonymizing your interview.
2. The non-anonymous raw data (first name, last name, e-mail address, notes and the audio recording of your interview) is only shared between the project partners (Stefan Götz, Yves Haas and Matthias Tichy from Ulm University) for transcribing and analyzing the answers.
3. We will anonymize your interview by transcribing it and delete the information about your first name, last name, e-mail and the audio recording of your interview as soon as possible and not later than 30 October 2020. Which means that individuals cannot be deduced from their interview.
4. For scientific publication, anonymized answers to this interview (transcribed interview and notes) will be further processed, e.g., coded or aggregated. Special risks for your person are not apparent with the processed results because individual persons cannot be inferred.
5. We strongly believe in open data to allow replication of our results as well as enabling further research. The anonymized answers (unprocessed and processed) to this interview (transcribed interview and notes) will be made available online to the public to accompany publications and stored in open data repositories. Please note that such published data can no longer be completely deleted and can be accessed and used by any person. Special risks for your person are not apparent with the anonymized answers, because individual persons cannot be inferred.

Contact person

If you have any questions, concerns or doubts, please do not hesitate to contact us:

Stefan Götz
Ulm University
Institute of Software Engineering and Programming Languages
89081 Ulm
E-Mail: stefan.goetz@uni-ulm.de

Declaration of consent

.....
(Name, Surname of the participating person)

I have read the general information on the interview study and agree to participate in the research project and the associated data processing.

(You can also give the following consent:)

☐ I have read the general information on the interview, which is conducted as part of a research project. I consent to participate in the interview and I consent to the related data processing as described in 1-4.

☐ I hereby consent to the publication of my anonymized answers (unprocessed and processed) to this interview (transcribed interview, notes and linked questionnaire) as described in 5.

I am aware that the consents are voluntary and can be refused without disadvantages (even individually) or revoked at any time without giving reasons. I am aware that in case of revocation, the legality of the processing carried out on the basis of the consent until revocation is not affected. I understand that I can simply contact the contact person named in the information for a revocation and that no disadvantages arise from the refusal of consent or its revocation.
I was informed and provided with the information on the collection of personal data during the interview study. I have also received a copy of this consent form.

B.5 Quotations

See Table B.2.

Table B.2: Selection of quotations from interview participants for specific factors

Factor	QID	PID	Quotation
GPL Capabilities	$Q_{gpl}1$	**P42**	*"[General purpose languages are] very good at constructing objects and filling in their fields [...] and computing 'simple' expressions."*
	$Q_{gpl}2$	**P14**	*"I think that in the end you have more tools for development. And I feel more productive."*
	$Q_{gpl}3$	**P8**	*"[...] when you reach the maintenance phase, maybe the [original] developers are gone. And you have an [MTL] program that might be more difficult to understand for [new] developers"*
Domain Focus	$Q_{df}1$	**P6**	*"What is better by using MTLs instead of GPLs is the fact that you are on the same abstraction level of the modelling language. You are basically treating apples with apples."*
	$Q_{df}2$	**P19**	*"[...] you are gonna cut away all those unneeded code and complexity and focus on your problem."*
	$Q_{df}3$	**P23**	*"Once you have things like rules and helpers and things like left hand side and right hand side and all these patterns then [it is] easier to create things like meta-rules to take rules from one version to another version [...]"*
	$Q_{df}4$	**P13**	*"To do this [tool support for analysing rule dependencies] [...] you have to resolve parameter dependencies and I immediately run into Turing completeness. And I don't have that with an external language [...]"*
	$Q_{df}5$	**P6**	*"They have existing infrastructure and people and everything that is based on established languages which is hard to change."*
Bidirectionality	$Q_{bx}1$	**P42**	*"in a general purpose programming language you would have to add a bit of clutter, a bit of distraction, from the real heart of the matter"*
	$Q_{bx}2$	**P40**	*"So either you write your own program to create unidirectional transformations in both directions or you write both directions by hand and that has the disadvantage that if, in the future, something changes in the transformation, then you have to rework both directions"*
	$Q_{bx}3$	**P41**	"*[...] That makes it harder for them to see whether something is correct or not and to master the complexity of these transformations.*"
	$Q_{bx}4$	**P11**	*"And as soon as I am at bidirectional transformations and there is somehow a loss of information. [...] And then [the question is] how difficult it is to access e.g. context elements that I have already created and need again later, because I want to refer to them."*
Incrementality	$Q_{inc}1$	**P56**	*"Declarative MTLs may have different computation paradigms which may be unfamiliar for developers used to imperative languages"*

Table B.2 – continued from previous page

Factor	QID	PID	Quotation
Mappings	$Q_{inc}2$	**P42**	*"[...] do not try to do it manually, because you will definitely have bugs,[...] because there will be some specific kind of change trajectory that you have missed, [...] this is a super hard problem."*
	$Q_{map}1$	**P24**	*"They hide those dimensions that reflect how graph-wise it would be computationally complex to interpret the problem to transform one model into another"*
	$Q_{map}2$	**P25**	*"So it restricts you in the way you can work and that makes it easier because that is what you need to do."*
	$Q_{map}3$	**P55**	*"This means that you can write the rules independently of the execution sequence, you can define them more declaratively and, at least in my experience, you can still manage to define these rule blocks in a comprehensible way for large transformations."*
	$Q_{map}4$	**P5**	*"I mentioned language engineering because a lot of the transformation difficulties are understanding the syntactical and semantic differences between two domain specific languages."*
	$Q_{map}5$	**P30**	*"[...] Hidden mechanisms or built in mechanisms may be more difficult to understand [thus] learning the language may be a bit more difficult."*
	$Q_{map}6$	**P38**	*"[...] you have a formal correspondence between the two models. And if you can transform in both directions, then you can practically keep both models, between which you want to transform back and forth, synchronous."*
	$Q_{map}7$	**P16**	*"Whereas when you need to do some more elaborate business logic or when you need to hook some external services or other sources of information into your transformation then I am saying that MTLs can start to be a little bit of a limit"*
	$Q_{map}8$	**P3**	*"If I want to reuse this model transformation just changing 2 words in ATL [is enough], if I wanted to do the same in Java instead of changing something in 2 places I have to do it in 5 or 6."*
Traceability	$Q_{trc}1$	**P22**	*"So that is something you often have to do manually in a GPL. So you have to maintain the trace information yourself and kind of re-implement that."*
	$Q_{trc}2$	**P32**	*"You have to know what a trace is. [...] And at some point, at the latest when you do something more complex, you need this stuff. "*
	$Q_{trc}3$	**P31**	*"[...] a model transformation rule only [contains] the domain transformation, so which domain object of the source domain is mapped to an object and how the object is mapped to the target domain. And that is what someone who tries to understand the model transformation is trying to get [...] out of the source code."*
Automatic Traversal	$Q_{trv}1$	**P49**	*"That means abstracting away from the order of traversal and then also knowing in which context this thing came up, that is a bit of a double-edged sword for me, [...] it has the potential to mask serious errors."*

Table B.2 – continued from previous page

Factor	QID	PID	Quotation
Pattern-Matching	$Q_{pm}1$	**P14**	*"[...] all the complexity of pattern matching is in the engine, but if you try to implement a mapping then all the complexity of keeping the traces you have to do that manually."*
Model Navigation	$Q_{nav}1$	**P41**	*"[...] you don't have to worry about the efficiency of the procedure, just figures out the optimal way of kind of traversing it. For me that is the biggest thing actually, they go and get me the data, if we can hide that from the user, that is great."*
	$Q_{nav}2$	**P11**	*"[...] I do not have to iterate over the model. I only say, I need this or that."*
Model Management	$Q_{man}1$	**P2**	*"[...] this technical level, how I access a model, [...]I get the elements out. That gets abstracted away."*
Reuse Mechanism	$Q_{rm}1$	**P51**	*"[...] we usually use object oriented programming languages and those already have some pretty strong tools for reusability in the appropriate contexts. So i think the bar here, that would we want model transformation languages to jump over, is to provide something more targeted towards modeling [...]"*
	$Q_{rm}2$	**P30**	*"[...] for ATL there are things like module superimposition, and other kinds, we have helper libraries."*
	$Q_{rm}3$	**P27**	*"[...] in the case of VIATRA one of the main goals of the pattern language we are using there is to allow reusing previously defined patterns. Basically any pattern can be included. So there is a lot of stuff you can do to reuse the element."*
Learnability	$Q_{ler}1$	**P23**	*"So the learning curve is pretty steep when trying to use MTLs. You need to learn a lot of stuff before you can use them properly."*
	$Q_{ler}2$	**P6**	*"You can take 10 Java developers and out of them probably 2 would understand what a MTL is. They don't have experience in modelling. Not because they are dumb, because they are not used to that."*
Debugging Tooling	$Q_{db}1$	**P51**	*"Well I think one of the other important points would be to [be] able to prove properties of transformations or check properties of transformations, [...] but we don't really have that for model transformations."*
Ecosystem	$Q_{eco}1$	**P49**	*"people from industry have a hard time when they are required to use multiple languages."*
	$Q_{eco}2$	**P49**	*"It is often on a technical level that the integration into the overall ecosystem of tools you have is not so great."*
	$Q_{eco}3$	**P31**	*"Something I see as a problem with some model transformation languages, which limit the applicability, is the coupling to Eclipse. This is what will cause us as a research community big problems some day [...]."*
Interoperability	$Q_{int}1$	**P36**	*"But the technologies, to combine them, it is difficult [...]"*
Tooling Awareness	$Q_{awa}1$	**P35**	*"And, I think, it is hard for new users to see, for example, what, which tool to use. Or which technology you should work with."*

Table B.2 – continued from previous page

Factor	QID	PID	Quotation
Tool Creation Effort	$Q_{tce}1$	**P01**	*"I am keenly aware of the cost to being able to develop a good programming language, the cost of maintaining it and the cost of adding debuggers and refactoring engines. It is enormous."*
	$Q_{tce}2$	**P6**	*"But it is definitely easier and faster to build the tool support and it allows you to do more advanced stuff. You can play around with your domain specific concepts in a lot of different ways."*
Tool Learnability	$Q_{tle}1$	**P34**	*"Because when i started to work with model transformation languages and to hear about them, [...] I do not think that [...] there was like initial go-to documentation."*
Tool Usability	$Q_{use}1$	**P22**	*"Basically all the good aids you see in a Java environment should be there even better in a MTL tool because model transformation is so much more abstract and more relevant that you should be having tools that are again more abstract and more relevant."*
	$Q_{use}2$	**P48**	*"There are quite a few corner cases, which are often not quite fixed and especially the usability is often very bad."*
Tool Maturity	$Q_{mat}1$	**P23**	*"Because [MTLs] have been around for like 30 years. And other languages and frameworks, they are created in 2-3 years, and they are good to go. And MTLs have been around for so long. And I think its mostly because industry has not taken it in. And it's just a problem of manpower put into the languages."*
Validation Tooling	$Q_{val}1$	**P8**	*"For example I can not remember any tool that offers reasonable support for testing. In Java you have JUnit and other. In ATL there is nothing."*
Language Skills	$Q_{skl}1$	**P1**	*"[...] this is the way you have to think in terms of formulating your problem"*
	$Q_{skl}2$	**P12**	*"And then you [need to] learn a language, the MTL."*
User Experience/ Knowledge	$Q_{exp}1$	**P21**	*"One of the reasons why Ada is virtually extinct is that developers preferred to have C++ on their CVs. Simply because there were more job postings with C++. And that develops a momentum of its own, which of course makes languages suffer. That also applies to DSLs."*
	$Q_{exp}2$	**P06**	*"Many MDSE courses are just given too late, when people are too acquainted with GPLs, and then its really hard for students to see the point of using models, modelling and MTLs, because it's comparable with languages and stuff they have already learned and worked with."*
Involved (meta-) models	$Q_{mod}1$	**P28**	*"As soon as you venture into eGenericType there is a lot of pain to be had and there is poor documentation."*
I/O Semantic gap	$Q_{gap}1$	**P22**	*"[...] as soon as I wanted to do something a bit more complex, I have often found that I was not able to express what I wanted to do easily and I had to resort to advanced features of the language in order to achieve what I want to do."*

Table B.2 – continued from previous page

Factor	**QID**	**PID**	**Quotation**
Size	$Q_{siz}1$	**P55**	*"The size is a good point. I would reduce that now to rules. But if I have several rules that then build on each other, then it will probably be easier with an MTL. Especially if you have a lot of dependencies between the rules."*

Appendix C

Appendix - Paper C

C.1 USM Results for Moderation Effects

See Tables C.1 to C.10.

Table C.1: Overview of moderation effects of meta-model size

	Comprehensibility	Ease of Writing	Expressiveness	Tool Support	Maintainability	Productivity	Reusability
Bidirectionality	0.0009	0.1644	0.0004	0.0341	0.0247	0.0197	0.0218
Incrementality	0.0174	0.0003	0.0086	0.0335	0	0.0224	0.0054
Mappings	0.018	0.0309	0.0058	0.0049	0	0.0012	0.0106
Model Management	0.1389	0.0264	0.0006	0.023	0	0.0177	0.0199
Model Navigation	0.001	0.0438	0.0019	0.0128	0.0032	0.0422	0.018
Model Traversal	0.0382	0.059	0	0.0422	0.0023	0.037	0.0106
Pattern Matching	0.0091	0	0.0003	0.03	0.0028	0.0418	0.0093
Reuse Mechanisms	0.0495	0	0.0077	0.0542	0.0693	0.0943	0.0021
Traceability	0.0778	0.0464	0.00001	0.0714	0.021	0.076	0.0223

Table C.2: Overview of moderation effects of model size

	Comprehensibility	Ease of Writing	Expressiveness	Tool Support	Maintainability	Productivity	Reusability
Bidirectionality	0.1	0.066	0.036	0.048	0.03	0.069	0.012
Incrementality	0.065	0.14	0.019	0.036	0.031	0.052	0.009
Mappings	0.085	0.033	0.076	0.042	0.032	0.083	0.003
Model Management	0.36	0.089	0.059	0.023	0.047	0.079	0.04
Model Navigation	0.074	0.007	0.04	0.053	0.05	0.072	0.011
Model Traversal	0.125	0.069	0.038	0.048	0.031	0.104	0.008
Pattern Matching	0.133	0.052	0.038	0.056	0.024	0.036	0.014
Reuse Mechanisms	0.096	0.00009	0.029	0.005	0.11	0.093	0.036
Traceability	0.078	0.173	0.02	0.048	0.096	0.108	0.016

Table C.3: Overview of moderation effects of transformation size

	Comprehensibility	Ease of Writing	Expressiveness	Tool Support	Maintainability	Productivity	Reusability
Bidirectionality	0.1	0.15	0	0.27	0.033	0.097	0.044
Incrementality	0.086	0.0075	0.0086	0.33	0.043	0.092	0.050
Mappings	0.13	0.0055	0	0.17	0.042	0.16	0.034
Model Management	0.32	0.094	0.021	0.23	0.012	0.13	0.056
Model Navigation	0.15	0.058	0.0022	0.25	0.049	0.13	0.064
Model Traversal	0.069	0.045	0.0038	0.21	0.061	0.15	0.015
Pattern Matching	0.16	0.055	0.040	0.28	0.075	0.12	0.055
Reuse Mechanisms	0.21	0.019	0.0077	0.16	0.27	0.23	0.087
Traceability	0.11	0.11	0	0.25	0.067	0.12	0.040

Table C.4: Overview of moderation effects of the amount of bidirectional use-cases

	Comprehensibility	Ease of Writing	Expressiveness	Tool Support	Maintainability	Productivity	Reusability
Bidirectionality	0.067	0.011	0.136	0.101	0.026	0.092	0.010
Incrementality	0.036	0.005	0.120	0.115	0.003	0.099	0.019
Mappings	0.029	0.034	0.065	0.084	0.002	0.072	0.001
Model Management	0.143	0.048	0.175	0.066	0.010	0.053	0.014
Model Navigation	0.058	0.003	0.155	0.049	0.037	0.071	0.020
Model Traversal	0.024	0.068	0.142	0.071	0.023	0.117	0.012
Pattern Matching	0.045	0.006	0.110	0.092	0.002	0.058	0.002
Reuse Mechanisms	0.040	0.026	0.130	0.220	0.100	0.190	0.013
Traceability	0.065	0.120	0.150	0.160	0.060	0.150	0.012

Table C.5: Overview of moderation effects of developer experience

	Comprehensibility	Ease of Writing	Expressiveness	Tool Support	Maintainability	Productivity	Reusability
Bidirectionality	0.05	8.4E-06	0.011	0.08	0.037	0.084	0.033
Incrementality	0.045	0.0003	0.007	0.083	0.032	0.055	0.046
Mappings	0.00032	0	0.0013	0.072	0.052	0.1	0.039
Model Management	0.15	0.025	0.032	0.085	0.05	0.073	0.034
Model Navigation	0.053	0.043	0.013	0.08	0.032	0.074	0.05
Model Traversal	0.045	0.0008	0.011	0.1	0.024	0.14	0.054
Pattern Matching	0.013	0.024	0.0052	0.081	0.032	0.078	0.048
Reuse Mechanisms	0.047	0	0.0015	0.08	0.077	0.055	0.022
Traceability	0.072	0.017	0.011	0.055	0.042	0.066	0.04

Table C.6: Overview of moderation effects of the semantic gap between input and output

	Comprehensibility	Ease of Writing	Expressiveness	Tool Support	Maintainability	Productivity	Reusability
Bidirectionality	0.01	0.033	0.008	0	0.089	0.056	0.041
Incrementality	0.006	0.067	0.009	0	0.119	0.046	0.045
Mappings	0.013	0.033	0.002	0	0.053	0.049	0.043
Model Management	0.1	0.023	0.021	0.009	0.163	0.052	0.054
Model Navigation	0.027	0.01	0.001	0.007	0.112	0.1	0.043
Model Traversal	0.023	0.032	0	0.002	0.194	0.067	0.029
Pattern Matching	0.032	0.033	0	0.029	0.239	0.096	0.032
Reuse Mechanisms	0.016	0.033	0.008	0.007	0.237	0.068	0.033
Traceability	0.032	0.027	0.0003	0.012	0.184	0.058	0.038

Table C.7: Overview of moderation effects of the sanity of involved meta-models

	Comprehensibility	Ease of Writing	Expressiveness	Tool Support	Maintainability	Productivity	Reusability
Bidirectionality	0.2	0.0001	0.00002	0.035	0.025	0.074	0.045
Incrementality	0.06	0.06	0.001	0.03	0.013	0.097	0.047
Mappings	0.09	0.03	0.03	0.07	0.005	0.074	0.059
Model Management	0.21	0.02	0.02	0.006	0.03	0.086	0.073
Model Navigation	0.11	0.12	0.002	0.06	0.04	0.099	0.044
Model Traversal	0.1	0.08	0	0.08	0.02	0.13	0.069
Pattern Matching	0.09	0	0.01	0.05	0.01	0.07	0.081
Reuse Mechanisms	0.11	0	0.008	0.03	0.05	0.094	0.059
Traceability	0.11	0.001	0	0.05	0.05	0.14	0.079

Table C.8: Overview of moderation effects of the amount of incremental use-cases

	Comprehensibility	Ease of Writing	Expressiveness	Tool Support	Maintainability	Productivity	Reusability
Bidirectionality	0	0.06	0.039	0.0045	0.034	0	0.00031
Incrementality	0	0.00067	0.012	0.0042	0.079	0	0.006
Mappings	0	0.017	0.079	0.0018	0.061	0.0072	0.017
Model Management	0.13	0.027	0.068	0.016	0.062	0.00038	0.039
Model Navigation	0	0.24	0.066	0.037	0.098	0.033	0.023
Model Traversal	0	0.047	0.025	0.028	0.083	0	0.0091
Pattern Matching	0.052	0	0.024	0.00048	0.073	0.032	0.034
Reuse Mechanisms	0.047	0	0.024	0.0036	0.14	0.051	0.034
Traceability	0.059	0	0.088	0.005	0.13	0	0.029

Table C.9: Overview of moderation effects of the choice of language

	Comprehensibility	Ease of Writing	Expressiveness	Tool Support	Maintainability	Productivity	Reusability
Bidirectionality	0	0	0.045	0	0.022	0	0
Incrementality	0	0	0.058	0	0.0035	0	0
Mappings	0	0.0063	0.043	0	0.0047	0.0052	0
Model Management	0.080	0	0.0050	0.0084	0.0077	0.0058	0
Model Navigation	0	0.031	0.051	0.022	0.038	0.027	0.0015
Model Traversal	0	0.00081	0.049	0.0075	0.040	0	0
Pattern Matching	0.028	0.00081	0.018	3.4E-05	0.019	0.0076	0.0065
Reuse Mechanisms	0.037	6.2E-09	0.042	0.0015	0.102	0.027	0.029
Traceability	0.036	0.022	0.0042	0.026	0.024	0	0.041

TABLE C.10: Overview of moderation effects of the experience in the used languages

	Comprehensibility	Ease of Writing	Expressiveness	Tool Support	Maintainability	Productivity	Reusability
Bidirectionality	0	0	0	0	0.0006	0	0
Incrementality	0	0	0.0086	0	0	0	0
Mappings	0	2.4E-17	4E-07	0	0	0.0006	0
Model Management	0.090	0	0.021	0.024	0	0.0012	0
Model Navigation	0	0.008	0.0022	0.0088	0.0014	0.043	0.015
Model Traversal	0	0.0008	0	0.0065	0.0001	0	0
Pattern Matching	0.0005	1.5E-19	4.4E-05	0.00049	0.00069	0.0011	0.0065
Reuse Mechanisms	0.0056	0.0014	0.0077	0.0071	0.018	0.025	0.0022
Traceability	0.022	2E-17	4E-06	0.028	0.014	0	0.016

C.2 Survey Overview

The Impact of Model Transformation Language Capabilities on the Perception of Language Quality Properties

In a prior interview study (https://doi.org/10.1007/s10664-022-10194-7 (https://doi.org/10.1007/s10664-022-10194-7)), we elicited expert opinions on what advantages result from what factors surrounding model transformation languages as well as a number of moderating factors that moderate the influence.

We now aim to quantitatively asses the interview results to confirm or reject the influences and moderation effects posed by different factors and to gain insights into how valuable different factors are to the discussion. As an expert in the field of model2model transformations your opinion is of high value for us because your answers can provide meaningful insights.

Participating in the survey will take about 25 minutes.

There are 3 pages in this survey.

There are 35 questions in this survey.

Quality properties of Model Transformation Languages

In the following you will assess **quality attributes** of model transformations and the languages used for writing them.

Each question presents a description of the quality attribute that is being assessed.

How *Incomprehensible* or *Comprehensible* are model transformations?

❗ Choose one of the following answers
Please choose **only one** of the following:

- ◯ Very Incomprehensible
- ◯ Incomprehensible
- ◯ Neither Incomprehensible nor Comprehensible
- ◯ Comprehensible
- ◯ Very Comprehensible

Comprehensibility describes the degree of effectiveness and efficiency with which the purpose and functionality of a transformation can be understood.

How *Hard* or *Easy* is it to *write* model transformations?

Choose one of the following answers
Please choose **only one** of the following:

- ◯ Very Hard
- ◯ Hard
- ◯ Neither Hard nor Easy
- ◯ Easy
- ◯ Very Easy

Ease of writing describes the degree of effectiveness and efficiency with which a developer can produce a transformation for a specific purpose.

How *Inexpressive* or *Expressive* are the languages used for writing model transformations?

Choose one of the following answers
Please choose **only one** of the following:

- ◯ Very Inexpressive
- ◯ Inexpressive
- ◯ Neither Inexpressive nor Expressive
- ◯ Expressive
- ◯ Very Expressive

Expressiveness describes the degree of effectiveness and efficiency with which language constructs support transformation development.

How *Unmaintainable* or *Maintainable* are model transformations?

❗ Choose one of the following answers
Please choose **only one** of the following:

- ◯ Very Unmaintainable
- ◯ Unmaintainable
- ◯ Neither Unmaintainable nor Maintainable
- ◯ Maintainable
- ◯ Very Maintainable

Maintainability describes the degree of effectiveness and efficiency with which a transformation can be modified.

How *Unproductive* or *Productive* is developing model transformations?

❗ Choose one of the following answers
Please choose **only one** of the following:

- ◯ Very Unproductive
- ◯ Unproductive
- ◯ Neither Unproductive nor Productive
- ◯ Productive
- ◯ Very Productive

Productivity describes the degree of effectiveness and efficiency with which transformations can be developed and used.

How *Constrained* or *Reusable* are parts of transformations in the languages used for writing model transformations?

Choose one of the following answers
Please choose **only one** of the following:

- ◯ Very Constrained
- ◯ Constrained
- ◯ Neither Constrained nor Reusable
- ◯ Reusable
- ◯ Very Reusable

Reuseability describes the degree of effectiveness and efficiency with which parts of a transformation can be reused to create new transformations (with different purpose).

How B*ad* or G*ood* is the tool support for languages used for writing model transformations?

Choose one of the following answers
Please choose **only one** of the following:

- ◯ Very Bad
- ◯ Bad
- ◯ Neither Bad nor Good
- ◯ Good
- ◯ Very Good

Tool Support describes the degree of effectiveness and efficiency with which tools support developers in their effort.

Capability utilisation of Model Transformation Languages

In the following you will be asked to estimate how often you use certain **capabilities** of model transformation languages.

Each question presents a description of the language capability that is being assessed.

What percentage of your transformations utilise *Bidirectionality* functionality of the used languages?

❗ Only numbers may be entered in this field.
❗ Your answer must be between 0 and 100
Please write your answer here:

Capabilities to define transformations from source to target and target to source in one transformation rule. E.g. relations defined in QvTR can be executed in both directions.

What percentage of your transformations utilise *Incrementality* functionality of the used languages?

❗ Only numbers may be entered in this field.
❗ Your answer must be between 0 and 100
Please write your answer here:

Capabilities to define transformations that are applied whenever the source model changes and only manipulate those parts that did change.

What percentage of your transformations utilise *Mapping* functionality of the used languages?

❗ Only numbers may be entered in this field.
❗ Your answer must be between 0 and 100
Please write your answer here:

Capabilities to explicitly define correspondence between input and output elements. E.g. ATL rules require specification of input type and output type(s).

What percentage of your transformations utilise *Model Management* functionality of the used languages?

❗ Only numbers may be entered in this field.
❗ Your answer must be between 0 and 100
Please write your answer here:

Capabilities to automatically read and write models from and to files or other sources. E.g. for transformation languages built on EMF you can specify the input model and output path without having to handle reading and writing manually.

What percentage of your transformations utilise *Model Navigation* functionality of the used languages?

❗ Only numbers may be entered in this field.
❗ Your answer must be between 0 and 100
Please write your answer here:

Capabilities to seamlessly navigate a given model structure. E.g. OCL provides dedicated model navigation.

What percentage of your transformations utilise *Model Traversal* functionality of the used languages?

❗ Only numbers may be entered in this field.
❗ Your answer must be between 0 and 100
Please write your answer here:

Capabilities to automatically find and apply transformations to model elements. E.g. in ATL you do not have to specify where the elements on which a rule is applied to are taken from. The transformation engine selects those from the input model automatically.

What percentage of your transformations utilise *Pattern Matching* functionality of the used languages?

❗ Only numbers may be entered in this field.
❗ Your answer must be between 0 and 100
Please write your answer here:

Capabilities to automatically match patterns of model elements and apply transformations to them. E.g. Henshin allows you to define a graph structure that is matched in the source model during transformation.

What percentage of your transformations utilise *Reuse* functionality of the used languages?

❗ Only numbers may be entered in this field.
❗ Your answer must be between 0 and 100
Please write your answer here:

Capabilities to enable reusing whole transformations or parts of transformations. E.g. any type of rule inheritance. Copying parts of your code does not constitute reuse.

What percentage of your transformations utilise *Tracing* functionality of the used languages?

❗ Only numbers may be entered in this field.
❗ Your answer must be between 0 and 100
Please write your answer here:

Capabilities to (automatically) generate and maintain trace links between source and target elements. E.g. ATL automatically resolves traces to the corresponding output model element when an input model element is referenced as the value for an attribute.

Experience

How many *years* have you worked on developing model transformations?

Only numbers may be entered in this field.
Please write your answer here:

How many *hours* do you work on developing model transformations *per month*?

Only numbers may be entered in this field.
Please write your answer here:

Which *5* of the following languages do you *use most often* to develop model transformations?

❗ Check all that apply
❗ Please select at most 5 answers
Please choose **all** that apply:

- ☐ AGG
- ☐ ATL
- ☐ eMoflon
- ☐ ETL
- ☐ Fujaba
- ☐ GReAT
- ☐ GrGen
- ☐ Henshin
- ☐ Java
- ☐ JavaScript
- ☐ JTL
- ☐ Kermeta
- ☐ QVTo
- ☐ QVTr
- ☐ Ruby
- ☐ RubyTL
- ☐ Tefkat
- ☐ Viatra
- ☐ Xtend
- ☐ Other 1
- ☐ Other 2
- ☐ Other 3
- ☐ Other 4
- ☐ Other 5

Name the *first* other language you are using for developing model transformations.

Only answer this question if the following conditions are met: ((LANGS_Other1.NAOK (/limesurvey/index.php/questionAdministration/view/surveyid /112652/gid/121/qid/1722) == "Y"))

Please write your answer here:

Name the *second* other language you are using for developing model transformations.

Only answer this question if the following conditions are met: ((LANGS_Other2.NAOK (/limesurvey/index.php/questionAdministration/view/surveyid /112652/gid/121/qid/1722) == "Y"))

Please write your answer here:

Name the *third* other language you are using for developing model transformations.

Only answer this question if the following conditions are met: ((LANGS_Other3.NAOK (/limesurvey/index.php/questionAdministration/view/surveyid /112652/gid/121/qid/1722) == "Y"))

Please write your answer here:

Name the *fourth* other language you are using for developing model transformations.

Only answer this question if the following conditions are met: ((LANGS_Other4.NAOK (/limesurvey/index.php/questionAdministration/view/surveyid/112652/gid/121/qid/1722) == "Y"))

Please write your answer here:

Name the *fifth* other language you are using for developing model transformations.

Only answer this question if the following conditions are met: ((LANGS_Other5.NAOK (/limesurvey/index.php/questionAdministration/view/surveyid/112652/gid/121/qid/1722) == "Y"))

Please write your answer here:

For how many *years* have you *used* these *languages*?

Only numbers may be entered in these fields.
Each answer must be at least 0
Please write your answer(s) here:

AGG

ATL

eMoflon

ETL

Fujaba

GReAT

GrGen

Henshin

Java

JavaScript

JTL

Kermeta

QVTo

QVTr

Ruby

RubyTL

Tefkat

Viatra

Xtend

Other1

Other2

Other3

Other4

Other5

For how many *hours* do you *use* these *languages per month*?

Only numbers may be entered in these fields.

Each answer must be at least 0

Please write your answer(s) here:

AGG

ATL

eMoflon

ETL

Fujaba

GReAT

GrGen

Henshin

Java

JavaScript

JTL

Kermeta

QVTo

QVTr

Ruby

RubyTL

Tefkat

Viatra

Xtend

Other1

Other2

Other3

Other4

Other5

How *many elements* do the meta-models involved in your transformations have? Please estimate the percentage of your use cases that fall in the following ranges.

❗ Each answer must be between 0 and 100
❗ The sum must be at most 100
❗ Only integer values may be entered in these fields.
Please write your answer(s) here:

#elements ≤ 10

10 < #elements ≤ 20

20 < #elements ≤ 50

50 < #elements ≤ 100

100 < #elements ≤ 1.000

#elements > 1.000

E.g. If half of the meta-models in your transformations have 25 elements and the other half are meta-models with 4 elements, you would put 50 for #elements ≤ 10 and 50 for 20 < #elements ≤ 50.

How *large* are the models you transform measured in *number of model elements*? Please estimate the percentage of your use cases that fall in the following ranges.

❗ Each answer must be between 0 and 100
❗ The sum must be at most 100
❗ Only integer values may be entered in these fields.
Please write your answer(s) here:

#elements ≤ 10

10 < #elements ≤ 100

100 < #elements ≤ 1.000

1.000 < #elements ≤ 10.000

10.000 < #elements ≤ 100.000

#elements > 100.000

E.g. if 1/3 of all models you tranform contain 200 elements and the rest are larger than 100.000 elements, you would put 33 for 100 < #elements ≤ 1.000 and 66 for #elements > 100.000.

How *Small* or *Large* are your transfomrations? Please estimate the percentage of your use cases that fall in the following ranges.

❗ Each answer must be between 0 and 100
❗ The sum must be at most 100
❗ Only integer values may be entered in these fields.
Please write your answer(s) here:

Tiny (e.g. LOC ≤ 100)

Small (e.g.100 < LOC ≤ 500)

Medium (e.g. 500 < LOC ≤ 1.000)

Large (e.g. 1.000 < LOC ≤ 5.000)

Very Large (e.g. 5.000 < LOC ≤ 10.000)

Huge (e.g. LOC : 10.000)

How *Dissimilar* or *Similar* in structure are input and output meta-models in your transformations? Please estimate the percentage of your use cases that fall in the following ranges.

❗ Each answer must be between 0 and 100
❗ The sum must be at most 100
❗ Only integer values may be entered in these fields.
Please write your answer(s) here:

very dissimilar

dissimilar

neither similar nor dissimilar

similar

very similar

The structure of a meta-model is define by the number of elements and their associations with each other.

How *Dissimilar* or *Similar* are the attribute types of input and output elements that are related to each other in your transformations? Please estimate the percentage of your use cases that fall in the following ranges.

❗ Each answer must be between 0 and 100
❗ The sum must be at most 100
❗ Only integer values may be entered in these fields.
Please write your answer(s) here:

very dissimilar

dissimilar

neither similar nor dissimilar

similar

very similar

E.g. when mapping a Class to a Table and assigning the ClassName as the TableName the attribute types are identical when both are Strings.

How *Bad* or *Well structured* are the meta-models in your transformations? Please estimate the percentage of your use cases that fall in the following ranges.

- Each answer must be between 0 and 100
- The sum must be at most 100
- Only integer values may be entered in these fields.

Please write your answer(s) here:

very bad

bad

neither well nor bad

well

very well

A well structured meta-model does for example not split related data over a large number of meta-model elements if it can be avoided.

How *Bad* or *Well documented* are the meta-models in your transformations? Please estimate the percentage of your use cases that fall in the following ranges. Only consider documentation of the meta-model itself not documentation in you code.

❗ Each answer must be between 0 and 100
❗ The sum must be at most 100
❗ Only integer values may be entered in these fields.
Please write your answer(s) here:

very bad

bad

neither well nor bad

well

very well

Documentation means description of the meta-model elements, their attributes and associations as well as any invariants on them.

What percentage of your use cases require *Synchronization* between "input" and "output".

❗ Only numbers may be entered in this field.
❗ Your answer must be between 0 and 100
Please write your answer here:

What percentage of your use cases require *Incrementality*?

❗ Only numbers may be entered in this field.
❗ Your answer must be between 0 and 100
Please write your answer here:

Thank you very much for participating in this survey! We appreciate you took the time.

Good bye and have a nice day!

02.02.2023 – 12:56

Submit your survey.
Thank you for completing this survey.

C.3 Mail Templates

Dear ${Author Name},

We found your contact information while conducting a structured literature search on model-to-model transformations. We seek your expertise on that topic.

We invite you to participate in our study 'The Impact of Model Transformation Language Capabilities on the Perception of Language Quality Properties' (see below for the URL). It will only take about 20-25 minutes to complete our online survey. We believe your answers can provide meaningful insights and help drive the field further.

Our survey is based on a large-scale interview study that qualitatively assessed what the community believes to be the main factors that drive the advantages and disadvantages of Model Transformation Languages for model-to-model transformations. Our results have been published in the Empirical Software Engineering journal `https://doi.org/10.1007/s10664-022-10194-7`

Our survey now quantifies our results to provide a clear picture of which of our identified factors are most important. The methodology for this survey has been peer reviewed at the Registered Reports track at ESEM'22 and is available under `https://doi.org/10.48550/arXiv.2209.06570`

The survey is available at

`https://sp2.informatik.uni-ulm.de/limesurvey/index.php/112652?lang=en`

It will be open till January 15, 2023.

All responses are completely anonymous.

Many thanks for supporting our research!

Best regards

Stefan Höppner

C.4 Data Privacy Agreement

To invite people to participate in this survey, we used author information (first name, last name and email address) from published academic papers in the domain of model driven software engineering. We will delete your personal information 2 months after you received the invitation.

The participation in this online survey is anonymous. Your name and email address are only used to invite you.

For future publications, we will publish and further process the anonymous raw data collected in this survey. This includes aggregating and statistical analysis of answers provided by participants. We will perform this in a way that does not allow inferring the identity of individual participants (e.g., by stripping free text answers of information that could identify a participant). Contact Information

If you have any questions about this survey or the data you provided, please contact us:

Stefan Höppner
stefan.hoeppner@uni-ulm.de
Institute of Software Engineering and Programming Languages,
Ulm University,
James-Franck-Ring, 89069 Ulm, Germany

Appendix D

Appendix - Paper D

Paper D does not have an associated appendix. This page ensures consistency between the paper identifiers and their associated appendix.

Appendix E

Appendix - Paper E

E.1 OCL expression translations in Java SE5

```
Collection<Type> newCollection = new Collection<>();
for (Type t: collection) {
  if (e) {
    newCollection.add(t);
  }
}
```

List. E.1: Translation of collection->select(e) in Java SE5.

```
Collection<ResultType> newCollection = new Collection<>();
for (Type t: collection) {
  ResultType r = ...; //manipulate t in accordance with e
  newCollection.add(r);
}
```

List. E.2: Translation of collection->collect(e) in Java SE5.

```
boolean includes = false;
for (Type t: collection) {
  includes |= t == x;
}
```

List. E.3: Translation of collection->includes(x) in Java SE5.

```
element.getAttribute();
```

List. E.4: Translation of element.attribute in Java SE5.

```
Collection<AttributeType> newCollection = new Collection<>();
for (Type t: collection) {
  if (e) {
    newCollection.add(t.getAttribute());
  }
}
```

List. E.5: Translation of collection.attribute in Java SE5.

```
if (i > 5) {}
```

List. E.6: Translation of i | i > 5 in Java SE5.

Appendix F

Published Versions of included Articles

F.1 Paper A

Claimed advantages and disadvantages of (dedicated) model transformation languages: a systematic literature review

S. Götz, M. Tichy, R. Groner

International Journal on Software and Systems Modeling (SoSyM), volume 20, pages 469–503, 2021
Springer Nature

DOI: 10.1007/s10270-020-00815-4

Software and Systems Modeling (2021) 20:469–503
https://doi.org/10.1007/s10270-020-00815-4

REGULAR PAPER

Claimed advantages and disadvantages of (dedicated) model transformation languages: a systematic literature review

Stefan Götz[1] · Matthias Tichy[2] · Raffaela Groner[2]

Received: 20 November 2019 / Revised: 9 June 2020 / Accepted: 16 June 2020 / Published online: 14 July 2020

Abstract
There exists a plethora of claims about the advantages and disadvantages of model transformation languages compared to general-purpose programming languages. With this work, we aim to create an overview over these claims in the literature and systematize evidence thereof. For this purpose, we conducted a systematic literature review by following a systematic process for searching and selecting relevant publications and extracting data. We selected a total of 58 publications, categorized claims about model transformation languages into 14 separate groups and conceived a representation to track claims and evidence through the literature. From our results, we conclude that: (i) the current literature claims many advantages of model transformation languages but also points towards certain deficits and (ii) there is insufficient evidence for claimed advantages and disadvantages and (iii) there is a lack of research interest into the verification of claims.

Keywords Model transformation language · DSL · Model transformation · MDSE · Advantages · Disadvantages

1 Introduction

Ever since the dawn of model-driven engineering at the beginning of the century, model transformations, supported by dedicated transformation languages [31], have been an integral part of model-driven development. Model transformation languages (MTLs), being domain-specific languages, have ever since been associated with advantages in areas like productivity, expressiveness and comprehensibility compared to general-purpose programming languages (GPLs) [50,55,60]. Such claims are reiterated time and time again in the literature, often without any actual evidence. Nowadays, such an abundance of claims runs through the whole literature body that one can be forgiven when losing track of which claims verifiably apply and which are still purely visionary.

The **goal** of this study is to identify and categorize claims about advantages and disadvantages of model transformation languages made throughout the literature and to gather available evidence thereof. We do not intend to provide a complete overview over the current state of the art in research. For this purpose, we performed a systematic review of claims and evidence in the literature.

The main **contributions** of our study are:

- a systematic review and overview over the advantages and disadvantages of model transformation languages as claimed in the literature;
- insights into the state of verification of aforementioned advantages and disadvantages;

This study is intended for researchers to (i) raise awareness for the current state of research and (ii) incentivise further research in areas where we identified gaps. The study can also be of interest to practitioners who wish to gain an overview over what research claims about MTLs compared to a practitioners view of the matter.

To systematize information from the literature, we performed a systematic literature review [14,41] based on the research questions we defined (see Sect. 3.1). As a first step,

Communicated by Alfonso Pierantonio.

Stefan Götz
stefan.goetz@uni-ulm.de

Matthias Tichy
matthias.tichy@uni-ulm.de

Raffaela Groner
raffaela.groner@uni-ulm.de

[1] Ulm University, 89081 Ulm, Germany

[2] Ulm University, 80901 Ulm, Germany

during the review we selected 58 publications from which to extract claims and evidence for advantages and disadvantages of model transformation languages. Afterwards, we categorized claims and systematized the evidence to produce (i) a categorization of claimed advantages and disadvantages into 15 separate categories (namely *analysability*, *comprehensibility*, *conciseness*, *debugging*, *ease of writing a transformation*, *expressiveness*, *extendability*, *just better*, *learnability*, *performance*, *productivity*, *reuse and maintainability*, *tool support*, *semantics and verification*, *versatility*) and (ii) a systematic representation of which claims are verified through what means. From our results, we conclude that:

1. The current literature claims many advantages and disadvantages of model transformation languages.
2. A large portion of claims are very broad.
3. There is insufficient or no evidence for a large portion of claims.
4. There is a number of claims that originate in claims about DSLs without proper evidence why they hold for MTLs too.
5. There is a lack of research interest in evaluation and especially verification of claimed advantages and disadvantages.

We hope our results can provide an overview over what MTLs are envisioned to achieve, what current research suggests they do and where further research to validate the claimed properties is necessary.

The remainder of this paper is structured as follows: Sect. 2 introduces the background of this research, model-driven engineering and model transformation languages. In Sect. 3, we will detail the methodology used for the conducted literature review. We present our findings in Sect. 4. Afterwards, in Sect. 5, we discuss the results of our findings. This section will also include propositions for much needed validation of claims about model transformation languages synthesized from the literature review. Section 6 contains information about related work, and in Sect. 7 potential threats to the validity of this research are discussed. Lastly, Sect. 8 draws a conclusion for our research.

2 Background

In this section, we provide the necessary background for our study and explain the context in which our study integrates.

2.1 Model-driven engineering

In 2001, the Object Management Group published the software design approach called *Model-Driven Architecture* [52] as a means to cope with the ever-growing complexity of software systems. MDA placed models at the centre of development rather than using them as mere documentation artefacts. The approach envisions an automated, continuous specialization from abstract models towards code. Starting with the so-called *Computation Independent Models* (CIMs), each specialization step should provide the models with more specific information about the intended system, transforming them from *CIM* into *Platform Independent Models* (PIMs) and then into *Platform Specific Models* (PSMs) and finally into production ready source code.

The different abstraction levels were designed to enable practitioners to be as platform, system and language independent as possible. The notion of using models as the central artefact during development is what is commonly referred to as *Model-Driven (Software-) Engineering* (MDE/MDSE) or *Model-Based (Software-) Engineering* (MBE/MBSE) [20].

The structure of a model is defined by a so-called meta-model whose structure is then also defined by meta-models of their own.

2.2 Domain-specific languages

"A domain-specific language (DSL) provides a notation tailored towards an application domain and is based on relevant concepts and features of that domain" [61]. The idea behind this design philosophy is to increase expressiveness and ease of use through more specific syntax. As such, DSLs provide an auspicious alternative for solving tasks associated with a specific domain. Representative DSLs include *HTML* for designing Web pages or *SQL* for database querying and manipulation.

2.3 Model transformation languages

Models are transformed into different models of the same or a different meta-model via the so-called *model transformations*. Driven by the appeal of DSLs, a plethora of dedicated MTLs have been introduced since the emergence of MDE as a software development approach [3,7,38,43]. Unlike general-purpose programming languages, MTLs are designed for the sole purpose of enabling developers to transform models. As a result, model transformation languages provide explicit language constructs for tasks performed during model transformation such as model matching. Similar to GPLs, model transformation languages can differ vastly in several aspects, starting with features that can be found in GPLs as well like language paradigm and typing all the way to transformation-specific features such as directionality [22]. There are numerous of features that can be used to distinguish model transformation languages from one another. For a complete classification of these features, please refer

to Kahani et al. [39], Mens and Gorp [49] or Czarnecki and Helsen [22].

Model transformation languages, being DSLs, promise dedicated syntax tailored to enhance the development of model transformations.

3 Methodology

Our review procedures are based on the descriptions of literature and mapping reviews from Boot, Sutton and Papaioannou [14]. First of all, a protocol for the review was defined. The protocol, as defined in Boot, Sutton and Papaioannou [14], describes (I) the research background (see Sect. 2), (II) the objective of the review and review questions (see Sect. 3.1), (III) the search strategy (see Sect. 3.2), (IV) selection criteria for the studies (see Sect. 3.3), (V) a quality assessment checklist and procedures (see Sect. 3.4), (VI) the strategy for data extraction and (VII) a description of the planned synthesis procedures (see Sect. 3.5). A complete overview of all steps of our literature review can be found in Sect. 1.

The remainder of this section will describe in detail each of the introduced protocol elements, with the exemption of the research background which we already covered in Sect. 2.

3.1 Objective and research questions

To formulate the objective as well as to derive the research questions for our review, we first applied the *Goal-Question-Metric* approach [11] which splits the overall goal into four separate concerns, namely *purpose*, *issue*, *object* and *viewpoint*.

Purpose	Find and categorize
Issue	claims of and evidence for advantages and disadvantages
Object	of model transformation languages
Viewpoint	from the standpoint of researchers and practitioners.

Based on the described goal, we then extracted the two main research questions for our literature review:

RQ1	What advantages and disadvantages of model transformation languages are claimed in the literature?
RQ2	What advantages and disadvantages of model transformation languages are validated through empirical studies or by other means?

The aim of *RQ1* is to provide an extensive overview over what kinds of advantages or disadvantages are explicitly attributed to using dedicated model transformation languages compared to using general-purpose programming languages. We consider such an overview to be necessary, because the number of claims and their repetition in the literature to date makes it difficult to keep track of which claims verifiably apply and which are still purely visionary. Naturally to be able to distinguish between substantiated and unsubstantiated claims, it is also required to record which claims are supported by evidence. With *RQ2*, we aim to do exactly that. Combining the results of *RQ1* and *RQ2* then makes it possible to determine if, and how, a positive or negative claim about MTLs is verified. Additionally, this also enables us to identify those claims that have yet to be investigated.

3.2 Search strategy

Our search strategy consists of seven consecutive steps. A visual overview of the complete search process is shown in Fig. 3. The figure visualizes steps *Database search* to *Snowballing* from Fig. 1 in more detail.

In the first step, we defined the search string to be used for automatic database searches. For this, we identified major terms concerning our research questions. Each new term was made more specific than the previous one. The resulting terms and justifications for including them were:

- *Model-driven engineering* The overall context we are concerned with. This was included to ensure only papers from the relevant context were found.
- *Model transformation* The more specific context we are concerned with.
- *Model transformation language* Since our focus is on the languages to express model transformations.

We used a thesaurus to identify relevant synonyms for each term in order to enhance our search string. In addition, we included one representative model transformation language with graphical syntax, one imperative language, one declarative language and one hybrid language as well as the term *domain-specific language* and its synonyms. The selection of the representative languages was made on the basis of their widespread use, active development and in the case of *QVT* because it is the standard for model transformations adopted by the Object Management Group. All these additional terms were included as synonyms for the *model transformation language* term.

We dropped the terms *advantage* and *disadvantage* after initial searches, because they resulted in a too narrow of a result set which excluded key publications [29,33] manually identified by the authors.

To combine all keywords, we followed the advice of Kofod-Petersen [42] to use the Boolean ($\vee$) to group together synonyms and the Boolean ($\wedge$) to link our major term groups.

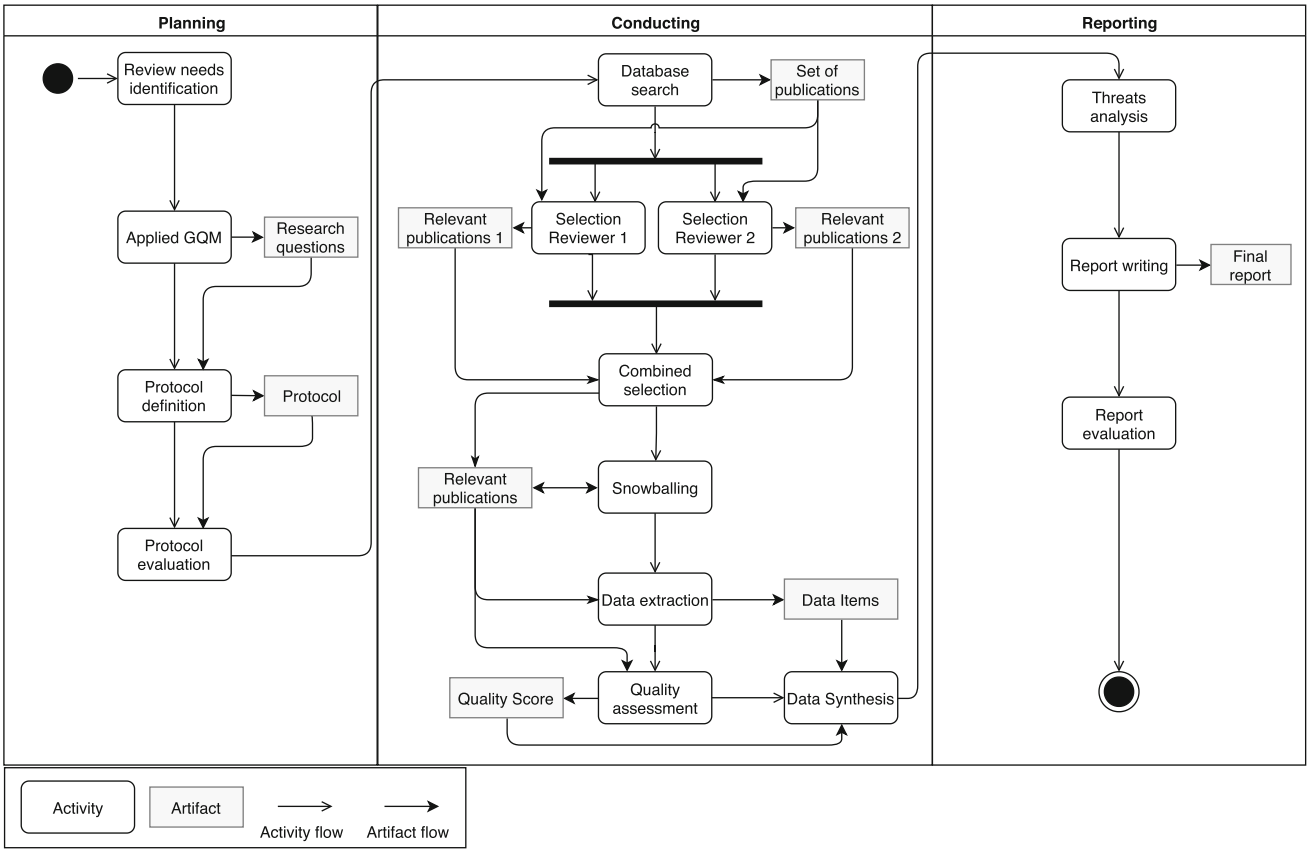


Fig. 1 Protocol overview

(Model Driven Engineering ∨ MDE ∨ Model Based Engineering ∨
MBE ∨ Model Driven Development ∨ MDD ∨
Model Driven Software Engineering ∨ MDSE ∨
Model Driven Software Development ∨
MDSE ∨ Model-Driven Software Development ∨
Model-Driven Engineering ∨ Model-Based Engineering ∨
Model-Driven Software Engineering)
∧
(Model Transformation ∨ Transformation ∨
Model Transformations ∨ Transformations)
∧
(Model Transformation Language ∨ Transformation Language ∨
ATL ∨ Henshin ∨ QVT ∨ TL ∨
Transformation Languages ∨ DSL ∨ domain specific language ∨
Model Transformation Languages)

Fig. 2 Search string used for automatic database searches

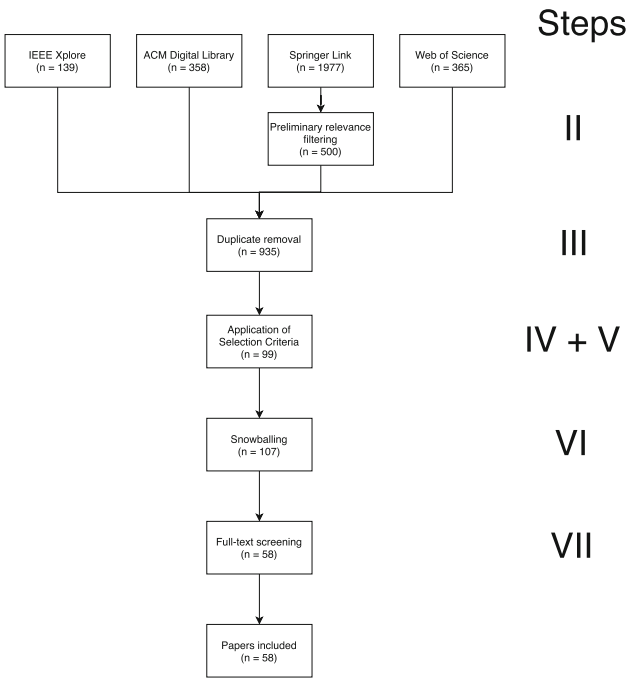


Fig. 3 The search and selection process

This resulted in the search string shown in Fig. 2 which was applied in full text searches.

We decided on the following four search engines to use for automated literature search:

– ACM Digital Library

- IEEE Xplore
- Springer Link
- Web Of Science

Search engines were chosen based on their overall coverage, completeness, the availability of accessible publications and usage in other literature reviews in this field such as Loniewski, Insfran, and Abrahão [8,48]. The online library *Science Direct*, which is often used in this domain, was excluded from our list due to us only having limited access to the publications in the database. We decided that the overhead of requesting access to all publications for which our proceedings would require a full text review (see step 4) would take up too much time; thus, we excluded the database from our automatic search process. Badampudi, Wohlin, and Petersen [6] also show that combining the automatic database searches with an additional snowballing process can make up for a reduced list of searched databases. We also decided against using Google Scholar as a search engine due to our experience with it producing too many irrelevant results and having a large overlap with ACM Digital Library and IEEE.

We conducted several preliminary searches on all four databases during the construction of the search string, to validate the resulting publications included key publications.

After the definition and validation of the search string, the second step consisted of full text searches using the search engines of *ACM Digital Library*, *IEEE Xplore Digital Library* and *Web of Science*.

For the *Springer Link* database, we realized early on that a full text search would result in too many hits and instead opted to query only the titles for the keyword *model transformation language* and its synonyms and filtered these results by applying a full text search based on the remaining keywords and their synonyms. The remaining results still far exceeded those of all other databases combined. We further realized during preliminary sifting that neither title nor abstracts of publications beyond the first 200 results suggested a relevance to our study. For that reason, we decided to cap our search at 500 publications, doubling the size of results from the point where the relevance of publications started to slide. This decision is supported by the fact that any publication which ended up in our data extraction set was found within the first 200 results.

All automated database searches were conducted between June 17 and June 28, 2019.

In the third step, all duplicates that resulted from using multiple search engines were filtered out based on the publication title and date. This also included the removal of publications that had extended versions published in a journal. This resulted in a total of 935 publications.

During the fourth step, two researchers independently used the selection criteria (see Sect. 3.3) on the titles and abstracts to select a set of relevant publications. The researchers categorized literature as either *relevant* or *irrelevant*. And in cases where they could not deduce the relevance based on the title and abstract, the publication was marked as *undecidable*.

Afterwards, in step 5 the results for each publication of the independent selection processes were compared. In cases where the two researchers agreed on *relevant* or *irrelevant*, the paper was included or excluded from the final set of publications. In cases of either a disparity between the categorizations or an agreement on *undecidable*, the full text of the publications was consulted using adaptive reading techniques to decide whether it should be included or excluded. Adaptive reading in this context meant going from reading the introduction to reading the conclusion and if a decision was still not reached reading the paper from start to finish until a decision could be reached. The step resulted in a total of 99 publications to use as a start set for the sixth step.

In the sixth step, we applied exhaustive backward and forward snowballing, meaning, as described in many previous studies [5,59], until no new publication was selected. The snowballing procedures followed the guidelines laid out by Wohlin [67]. Our start set was comprised of all 99 publications from step 5. We then applied backward and forward snowballing to the set. For backward snowballing, we used the reference lists contained in the publications, and for forward snowballing we used Google Scholar as suggested by Wohlin [67] and because from our experience it provides the most reliable source for the cited by statistic. To the cited and citing publications, we then applied our inclusion and exclusion criteria as described in step 4. All publications that were deemed as relevant were then used as the starting set for the next round of snowballing until no new publications were selected as relevant. The result of this step was a set of 107 *relevant* publications.

Lastly, in step 7, we filtered out all publications that did not explicitly mention advantages or disadvantages of model transformation languages by reading the full text of all publications. This step was introduced to filter out the noise that arose from a broader search string and less restrictive inclusion criteria (see Sect. 3.3). The remaining 58 publications form our final set on which data synthesis was performed on. (A list of all included publications with an unique assigned ID can be found in "Appendix B".)

3.3 Selection criteria

We decided that a publication be marked as relevant, if it satisfies at least one inclusion criteria and does not satisfy any exclusion criteria. The inclusion criteria were chosen to include as many papers that potentially contain advantages or disadvantages as possible. A publication was included if:

IC1	The publication introduces a model transformation language.
IC2	The publication analyses or evaluates properties of one or multiple model transformation languages.
IC3	The publication describes the application of one or multiple model transformation languages.

IC1 is an inclusion criteria, because the introduction of a new language should include a motivation for the language and possibly even a section on potential shortcomings of the language. Such shortcomings can be attributed either to the design of the language or to the concept of model transformation languages as a whole.

A publication that is covered by *IC2* can help answer both *RQ1* and *RQ2* depending on the analysed/evaluated properties.

IC3 forms our third inclusion criteria since experience reports can be a good source for both strengths and weaknesses of any applied technique or tool.

Our exclusion criteria were:

EC1	Publications written in a language other than English.
EC2	Publications that are tutorial papers, poster papers or lecture slides.
EC3	Publications that are a Doctoral/Bachelor /Master thesis.

EC1 ensures that the scientific community is able to verify our extracted data from publications.

Because tutorial papers, poster papers and lecture slides are less reliable and do not provide enough information to work with, they are excluded with *EC2*.

Lastly, to reduce the required workload, we excluded all thesis publications with *EC3* as full text reviews would take up too much time. We also argue that relevant thesis findings are most likely also published in journal or conference papers.

3.4 Quality assessment checklist and procedures

Assessing the quality of publications found during the selection process is an essential part of a literature review [14].

For that reason, we adopted a list of six quality attributes for studies. The quality attributes (seen in Table 1) are taken from Shevtsov et al. [57] which adapted quality criteria from Weyns et al. [64]. Each quality item has a set of three characteristics for which a value between 0 and 2 is assigned. The quality score of a publication is calculated by summing up the values for each characteristic, making 12 the maximum quality score for a publication. The quality score did not influence the decision to include or exclude a publication.

3.5 Data extraction strategy

Based on our research questions, and general documentation concerns, we devised a total of eight data items to extract from each selected publication. Table 2 lists all extracted data items.

Data items *D1–D3* are recorded for documentation purposes.

To gather explicitly, claimed advantages and disadvantages of model transformation languages *D4* and *D5* are necessary items to include.

Another goal of our literature review is to find out which advantages or disadvantages are empirically verified. It is therefore necessary to extract information about whether empirical evidence exists and which advantage or disadvantage it is concerned with (*D6*). Similarly, citations used to back up claimed advantages or disadvantages are also documented (*D7*). Our goal is it to either track down references that provide evidence and find sources of common claims about advantages and disadvantages of model transformation languages.

Lastly, in order to evaluate the quality of publications the quality score *D8* for each publication is recorded.

All data items were extracted during full text reviews of all selected publications.

3.6 Synthesis procedures

The synthesis of the collected data was split into multiple parts with multiple results for each research question.

3.6.1 RQ1: What advantages and disadvantages of model transformation languages are claimed in the literature?

The first part of the synthesis for *RQ1* was a simple collection of all claimed advantages and disadvantages. This was done in order to create a basic overview.

Next, an analysis of all collected items was performed in order to devise categories for the advantages and disadvantages. To develop categories, we used initial coding and focused coding as described by Charmaz [19]. First, all claims were analysed claim by claim to extract common phrases or similar topics. These were then used to group together claims and develop descriptive terms when then served as the name for the category formed by the grouped claims. The categories themselves were split into a positive section and a negative section to contrast negative and positive mentions with each other.

Using the devised categorization allows for quick identification of contradictory claims. Such claims then have to be further analysed in terms of origin, context and supporting evidence.

Table 1 Quality assessment criteria [64]

Q1: Problem definition	
2	The authors provide an explicit problem description
1	The authors provide a general problem description
0	There is no problem description
Q2: Problem context	
2	If there is an explicit problem description for the research, this problem description is supported by references
1	If there is a general problem description, this problem description is supported by references
0	There is no description of the problem context
Q3: Research design	
2	The authors explicitly describe the plan (different steps, timing, etc.) they have used to perform the research, or the way the research was organized
1	The authors provide some general words about the research plan or the way the research was organized
0	There is no description of how the research was planned/organized
Q4: Contributions	
2	The authors explicitly list the contributions/results
1	The authors provide some general words about the results
0	There is no description of the research results
Q5: Insights	
2	The authors explicitly list insights/lessons learned
1	The authors provide some general words about insights/lessons learned
0	There is no description of the derived insights
Q6: Limitations	
2	The authors explicitly list problems and/or limitations
1	The authors provide some general words about limitations and/or problems
0	There is no description of the limitations

Table 2 Data items

ID	Data	Purpose
D1	Author(s)	Documentation
D2	Publication year	Documentation
D3	Title	Documentation
D4	Named advantage(s) of MTL(s)	RQ1
D5	Named disadvantage(s) MTL(s)	RQ1
D6	Empirical evidence of advantage(s) or disadvantage(s)	RQ2
D7	Cited evidence	RQ2
D8	Quality score	Documentation

3.6.2 RQ2: What advantages and disadvantages of model transformation languages are validated through empirical studies or by other means?

To analyse evidence of claimed advantages and disadvantage, we started by assessing the quality of each respective publication using the quality score system from Sect. 3.4.

Afterwards, we devised a visual representation for claims and evidence thereof in publications. The representation allows a straightforward identification of substantiated and unsubstantiated claims and tracking of citations back to the origin of cited claims. This in turn enabled us to easily identify whether citations back up stated claims or serve as nothing more than a reference to a publication which claims the same thing.

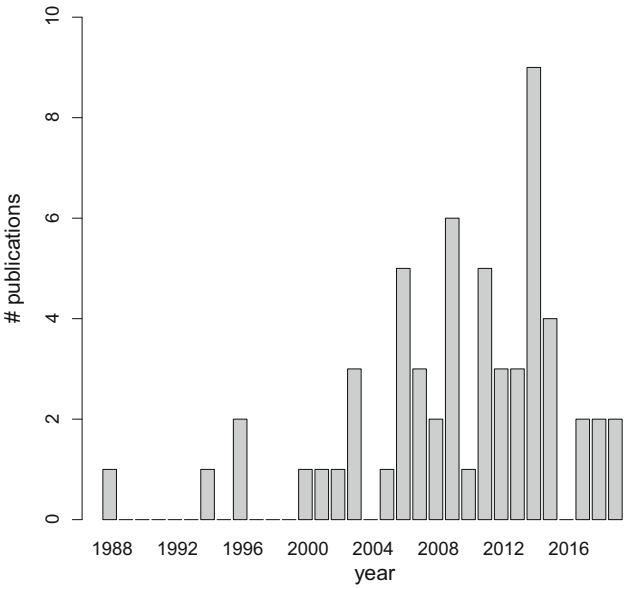


Fig. 4 Number of publications that mention or evaluate advantages or disadvantages of MTLs per year

4 Findings

In this section, we provide a summary of the synthesized data as well as an analysis of the demographics and quality of publications. The summary will be in narrative form, supported by plots and graphs as suggested by Boot, Sutton and Papaioannou [14]. Before describing our findings with regard to the research questions from Sect. 3.1, we first offer statistics and information about the demographic data of the collected literature as well as an overview over their quality which we assessed using the quality criteria from Sect. 3.4.

4.1 Demographics

Figure 4 provides an overview over the quantity of included publications per year. An interesting thing to note is that it took only two years from the introduction of the *Model-Driven Architecture* in 2001 to the first mentions of advantages of model transformation languages. One of the most cited papers about model transformations in our literature review was published that year too (**P63**). Its title shapes introductions of publications in the community even today: *Model transformation: The heart and soul of model-driven software development*.

Scrutinizing claims about MTLs, however, just recently started to be a focus of research, with the first study (**P59**) dedicated to evaluating advantages of MTLs being published in 2018. To us, this suggests that research might be slowly catching on to the fact that evaluation of specific properties of MTLs is necessary instead of relying on broad claims. Simply relying on the fact that model transformation languages are DSLs and that DSLs in general fare better compared to non-domain-specific languages [12,28,40] is not enough.

Industrial case studies about the adoption of MDSE have been performed much earlier than 2018, but such studies mainly focus on the complete MDSE workbench and do not analyse the impact of the used MTLs in great detail. The case study **P670** for example, while stating that "The technology used in the company should provide advanced features for developing and executing model transformations", does not go into detail about neither current shortcomings nor any

Table 3 Number of publications that mention specific MTLs

Model transformation language	# of mentions
ATL	16
EMT	1
ETL	3
GreAT	1
Henshin	1
Iquery	1
JTL	1
MOFLON	1
MT	1
NTL	2
QVT-O	4
QVT-R	2
SDM	1
SIGMA	1
SiTra	1
Tefkat	1
TGG	1
TN	1
VMTL	1

other specifics of model transformation languages used during the development process.

Overall, there are *32* publications that mention advantages and *36* publications that mention disadvantages. Moreover, *four* publications provide empirical evidence for either advantages or disadvantages, while *12* publications use citations to support their claims and *14* publications use other means such as examples and experience (more on this in Sect. 4.4).

Lastly, Table 3 shows which transformation languages were directly involved in publications used in our data extraction. We counted a transformation language as being involved if it was used, analysed or introduced in the publication. Simply being mentioned during enumerations of example MTLs was not sufficient.

The table paints an interesting picture. ATL far exceeds all other model transformation languages in involvement, and most languages are only discussed in a single publication.

4.2 Quality of publications

The results from the quality assessment, summarized in Fig. 5, shows that both the problem context and definition as well as the overall contributions are well defined in a majority of publications. Insights drawn from the work described in these publications, while less comprehensive in many cases, are also described most often. However, thorough descriptions of the research design, the used methods or steps taken

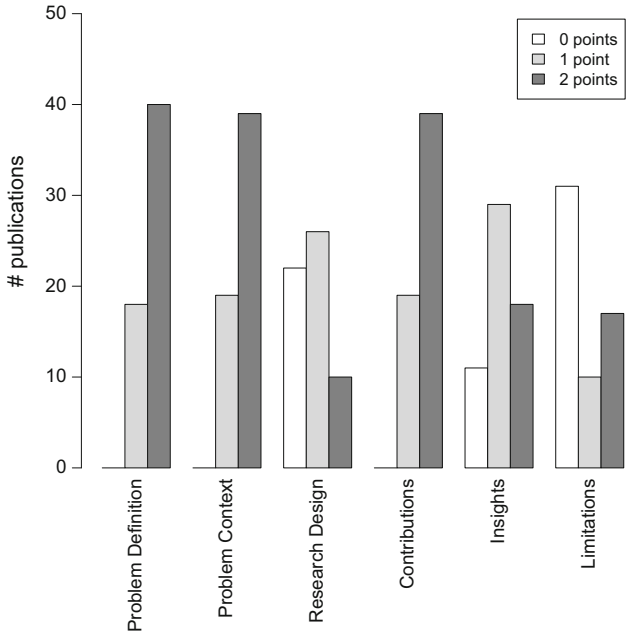


Fig. 5 Quality score distribution

are less common, a trend which is even more prominent for the presentation and discussion of limitations that act upon the studies. Similar observations have already been made by other literature reviews in different domains [26,57].

4.3 RQ1: Advantages and disadvantages of model transformation languages

We used data items D4 and D5 to answer our first research question, namely which advantages or disadvantages of dedicated model transformation languages are claimed in the literature. The resulting statements were sorted into 15 different categories (seen in Fig. 6) which arose naturally from the collected statements. An overview over all claims sorted into the different categories is given in Table 4. The table ascribes each claim with a unique ID (Cxx) for reference throughout this work. The table also contains evidence used to support a claim (if existent) to which we will come back later in Sect. 4.4. For almost all categories, there exist papers that describe model transformation languages as being advantageous as well as publications that describe them as disadvantageous in the category. In the following, we discuss the statements made in publications for each category.

4.3.1 Analysability

Throughout our gathered literature, there is only one publication, **P45**, that mentions analysability. According to them, a declarative transformation language comes with the added advantage of being automatically analysable which

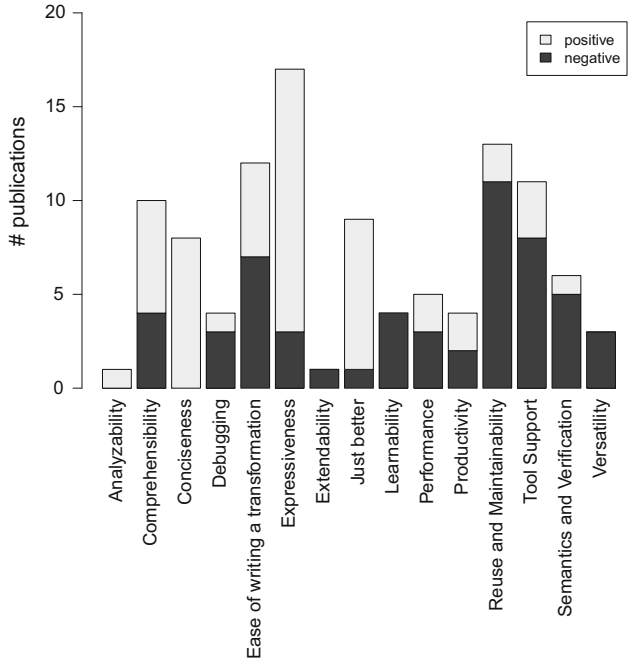


Fig. 6 Number of publications that claim an advantage or disadvantage of MTLs in a category

enables optimizations and specialized tool support (*C1*). While a detailed discussion of this claim within the publication remains owed, the authors provide examples of how static analysis allows the engine to implicitly construct an execution order. While our literature review found only a single publication that explicitly mentions analysability as an advantage of model transformation languages, there do exist multiple publications [2,3,63] that contain analysis procedures for model transformations.

4.3.2 Comprehensibility

Comprehensibility is a much disputed and multifaceted issue for model transformation languages. A total of eleven publications touch on several different aspects of how the use of MTLs influences the understandability of written transformations.

The first aspect is the use of graphical syntax compared to a textual one which is typically used in general-purpose programming languages. In **P63**, the authors talk about "perceived cognitive gains" of graphical representations of models when compared to textual ones (*C6*). A pronouncement that is echoed in **P43** states that graphical syntax for transformations is more intuitive and beneficial when reading transformation programs (*C2*).

While all these claims about graphical notation increasing the comprehensibility of transformations stand undisputed in our gathered literature, there are other facets in which graphical notation is said to be disadvantageous. We will come back to them later on in Sect. 4.3.5.

Declarative textual syntax is another commonly used syntax for defining model transformations. The authors of **P45** contend that a declarative syntax makes it easy to understand transformation rules in isolation and combination (*C3*). However, declarative transformation languages are typically based on graph transformation approaches which can become complex and hard to read according to **P70** (*C13*). They additionally assert that the use of abstract syntax hampers the comprehensibility of transformation rules (*C12*). Furthermore, **P22** insist that the use of graph patterns results in only parts of a meta-model being revealed in the transformation rules and that current transformation languages exhibit a general lack of facilities for understanding transformations (*C8*). **P22** also reports that understanding transformations in current model transformation languages is hampered, specially by the fact that many of the involved artefacts such as meta-models, models and transformation rules are scattered across multiple views (*C9*). **P29** brings forward the concern that large models are also a factor that hampers comprehensibility since there exist no language concepts to master this complexity (*C11*). Adding to this point, **P27** describes that for non-experts (e.g. stakeholders) transformations written in a traditional model transformation language are *"very complex to understand"* because they lack the necessary skills (*C10*). The authors of **P95** on the other hand claim that the usage of dedicated MTLs, which incorporate high-level abstractions, produces transformations that are more concise and more understandable (*C7*). This sentiment is shared in **P44** which explains the belief that using GPLs for defining synchronizations brings disadvantages in comprehensibility compared to model transformation languages (*C3*).

Understanding a transformation requires, among other things, understanding which elements are affected by it and in which context a transformation is placed. Using a model transformation language is beneficial for this as shown in the study described in **P59** (*C5*).

4.3.3 Conciseness

Interestingly, there seems to be a consensus on the conciseness of model transformation languages compared to GPLs.

In general, dedicated model transformation languages are seen as more concise (**P63** *C17*, **P95** *C21*) which, apart from textual languages, is also stated for graphical languages in **P75** (*C18*).

The fact that MTLs are more abstract making them more concise and thus better is claimed multiple times in **P80** (*C19*), **P52** (*C15*), **P3** (*C14*) and **P95** (*C20*), while **P673** claims that the abstraction in MTLs helps to reduce their overall complexity (*C22*).

The SLOC metric has also been drawn from as a way to compare MTLs with other MTLs and even GPLs. According to an experiment described in **P59**, using a rule-based model

transformation reduces the transformation code by up to 48% (*C16*). Whether or not this is any indication of superiority is a disputed subject [9].

4.3.4 Debugging

Debugging support is much less disputed than comprehensibility. Of the five publications that talk about debugging in model transformation languages, none praise the current state of debugging support.

P22 (*C24*, *C25*) and **P90** (*C27*) both describe that currently no sufficient debugging support exist for MTLs. And while in **P95** it is stated that debugging of transformations in a dedicated languages is likely better than when the transformation is written in a general-purpose language (*C23*) they fail to bring forth a single example for their assertion.

Lastly, **P45** lauded declarative syntax for its benefit in comprehension but also note that imperative syntax is easier to debug in general (*C26*).

4.3.5 Ease of writing a transformation

The main purpose of model transformation languages is to improve the ease with which developers are able to define transformations. Hence, this should also be a main benefit when compared to general-purpose languages. However, the authors of the study described in **P59** found: *"no sufficient (statistically significant evidence) of general advantage of the specialized model transformation language QVTO over the modern GPL Xtend"* (*C39*). This is not to say that there are none as the authors admit the conclusions were *"made under narrow conditions"* but is still a concerning finding. Much more so because claims about such benefits of using MTLs persist through the literature. Claims such as those described in **P29** (*C29*), **P672** (*C32*) and **P50** (*C30*) state that their simpler syntax makes it easier to handle and transform models. These claims draw from statements about the expressiveness, to which we will come to in the next section, and reason that better expressiveness must lead to an easier time in writing transformations. A potential reason that hampers model transformation languages from evidentially being better for writing transformations is cited in **P27** (*C34*) and **P28** (*C35*). They both state that using a model transformation language requires skill, experience and a deep knowledge of the meta-models involved (**P56** *C38*). In our opinion, however, this holds true regardless of the language used to transform models.

Moreover, many model transformation languages use declarative syntax which can be unfamiliar for many programmers, according to **P45** (*C37*) and **P63** (*C40*), which are much more familiar with the status quo, i.e. imperative languages. The authors of **P22**, on the other hand, state that imperative MTLs often require additional code since many issues have to be accomplished explicitly compared to implicitly in declarative languages (*C33*).

Lastly, graphical syntax is said to make writing model transformations easier as the syntax is purported to be more intuitive for this task compared to a textual one in **P3**. In **P43** (*C36*) and **P672** (*C41*), however, the authors claim that graphical syntax can be complicated to use and that textual syntax is more compact and does not force users to spend time to beautify the layout of diagrams.

4.3.6 Expressiveness

As described in Sect. 2.2, the idea behind domain-specific languages is to design languages around a specific domain, thus making it more expressive for tasks within the domain [50]. Since model transformation languages are DSLs, it should not be a surprise that their expressiveness in the domain of model transformations is mentioned almost exclusively positive by a total of 19 different publications found in our literature review.

A large portion (**P95**, **P80**, **P94**, **P63**, **P15**, **P40**, **P52**, **P70**) of publications refer to expressiveness state that the higher level of abstraction that results from specific language constructs for model manipulation increases the conciseness and expressiveness of MTLs. **P80** additionally asserts that model transformation languages are just easier to use (*C61*).

Another portion (**P2**, **P15**, **P45**, **P677**, **P27**, **P63**, **P95**, **P27**) explains that the expressiveness is increased by the fact that model transformation engines can hide complexity from the developer. One such complex task is pattern matching and the source model traversal as mentioned in **P2** (*C42*), **P15** (*C43*) and **P45** (*C53*), respectively. According to them, not having to write the matching algorithms increases the expressiveness and ease of writing transformations in MTLs. Implicit rule ordering and rule triggering is another aspect that **P15** (*C46*), **P45** (*C51*) and **P677** (*C65*) claim increases the expressiveness of a transformation language. Related to rule ordering is the internal management and resolution of trace information which is stated by **P15** (*C44*), **P45** (*C50*), **P677** (*C65*) and **P95** (*C64*) to be a major advantage of model transformation languages. Furthermore, **P45** asserts that implicit target creation is another expressiveness advantage that MTLs can have over general-purpose languages (*C52*). Lastly, the study described in **P59** observed that copying complex structures can be done more effectively in MTLs (*C56*).

However, we also uncovered some shortcomings in current syntaxes. **P10** argues that the lack of expressions for transforming a single element into fragments of multiple targets is a detriment to the expressiveness of transformation languages, going as far as to allege that without such constructs model transformation languages are not expressive enough (*C68*). **P32** implies that MTLs are unable to transform OCL constraints on source model elements to target

model elements (*C69*). And lastly **P33** critiques that model transformation languages lack mechanisms for describing and storing information about the properties of transformations (*C70*).

4.3.7 Extendability

Being able to extend the capabilities of a model transformation language seems to be less of a concern to the community. This can be seen by the fact that only **P50** touches this issue. They explain that external MTLs can only be extended (*"if at all"*) with a specific general-purpose language (*C71*). Internal model transformation languages of course do not suffer from this problem since they can be extended using the host language [21,32,46].

4.3.8 Just better

Apart from specific aspects in which the literature ascribes advantages or disadvantages to model transformation languages, there are also several instances where a much broader claim is made.

P86 for example states that there exists a consensus that MTLs are most suitable for defining model transformations (*C78*). This claim is also reiterated in several other publications using statements such as "the only sensible way" or "most potential due to being tailored to the purpose" (**P9**, **P23**, **P63**, **P64**, **P66**). However, one publication claims that both GPLs and MTLs are not well suited for model migrations and that instead dedicated migration languages are required (**P34** *C80*).

4.3.9 Learnability

The learnability issues of tools have been shown to positively correlate with usability defects [1] and thus their general acceptance.

However, the learnability of model transformation languages is rarely discussed in detail. **P30** (*C81*), **P58** (*C83*) and **P81** (*C84*) all express concerns about the steep learning curve of model transformation languages, and **P52** explain that transformation developers are often required to learn multiple languages, which requires both time and effort (*C82*).

4.3.10 Performance

The execution performance of transformations is an important aspect of model transformations. Often times, the goal is to trigger a chain of multiple transformations with each change to a model. Hence, good transformation performance is paramount to the success of model transformation languages.

Opinion on performance in the literature is divided. On the one hand, there are publications such as **P52** (*C88*) and **P80** (*C89*) which describe that the performance of dedicated MTLs is worse than that of compiled general-purpose programming languages, while on the other hand there is **P95** which states that some introduced transformation languages are more performant (*C85*), citing articles from the Transformation Tool Contest (TTC), and **P675** which shows a performance comparison of transformations written in Java and GrGen where GrGen performs better than Java (*C86*). There are also more nuanced views on the subject. **P45** describes that practitioners sometimes perceive the performance as worse and that there exist factors that hamper the performance (*C87*). The listed factors are the fact that the transformation languages are often interpreted, a mismatch with hardware and less control over the algorithms that are used. However, they also describe that specialized optimizations can bridge the performance gap.

4.3.11 Productivity

Increased productivity through the use of DSLs is a much cited advantage [50] (*C6D*). Unsurprisingly, it resurfaces in various forms in the context of model transformation languages as well. For instance, in **P45** it is described that the use of declarative MTLs improves the productivity of developers (*C91*). **P29** goes even further, claiming that the use of any model transformation language results in higher productivity (*C90*).

This is contrasted by the hypothesis that productivity in general-purpose programming languages might be higher due to the fact that it is easier to hire expert users, which was put forward in **P59** (*C93*). Lastly, **P32** raises the concern that some of the interviewed subjects perceive model transformation languages as not effective, i.e. not helpful for the productivity of developers (*C92*).

4.3.12 Reuse and maintainability

In our gathered literature, maintainability is used as a motivation for modularization and reuse concepts. **P29**,**P60** and **P95** all claim that reuse mechanisms are necessary to keep model transformations maintainable. Combined with a total of eight (**P4**, **P10**, **P29**, **P33**, **P41**, **P60**, **P95**, **P78**) publications that state that reuse is hardly, if at all, established in current model transformation languages, this paints a bleak picture for both maintainability and reuse. The need for reuse mechanisms has already been recognized in the research community as stated by **P77** in which the authors explain that a plethora of mechanisms have been introduced (*C95*) but are hindered by several barriers such as insufficient abstraction from meta-models and platform or missing repositories of reusable artefacts (*C103*).

There exists only a single claim that directly addresses maintainability. **P44** states that bidirectional model transformation languages have an advantage when it comes to maintenance (*C94*).

Apart from the maintainability of written code, there is also the maintainability of languages and their ecosystems. Surprisingly, this is hardly discussed in the literature at all. Only **P52** explains that evolving and maintaining a model transformation language is difficult and time-consuming (*C101*).

4.3.13 Semantics and verification

Three publications (**P39**, **P23**, **P58**) all suggest that most model transformation languages do not have well-defined semantics which in turn makes verification and verification support difficult (**P22** *C109*). **P44**, however, explains that bidirectional transformations are advantageous with regards to verification (*C107*).

4.3.14 Tool support

Tools are another important aspect in the MDE life cycle according to Hailpern and Tarr [28]. They are essential for efficient transformation development. Regrettably, MTLs lack good tool support according to **P23**, **P45**, **P52** and **P80** and if tools exist, they are not close to as mature as those of general-purpose languages as stated in **P74** (*C119*). Additionally, the authors of **P94** explain that developers of MTLs need to put extra effort into the creation of tool support for the language (*C121*). This might, however, be worthwhile, because **P44** presumes that dedicated tools for model transformation languages have the potential to be more powerful than tools for GPLs in the context of transformations (*C114*). And due to the high analysability of MTLs, **P45** explains that tool support could potentially thrive (*C115*). Internal MTLs, on the other hand, are able to inherit tool support from their host languages as reported by **P23** (*C113*). This helps to mitigate the overall lack of tool support, at least for internal MTLs.

An interesting discussion to be held is how important tool support for the acceptance of MTLs actually is. Whittle et al. [65] describe that organizational effects are far more impactful on the adoption of MDE, while the results of Cabot and Gérard [16] contradict this observation citing interviewees from commercial tool vendors that stopped the development of tools due to lack of customer interest.

4.3.15 Versatility

It should be self-evident that languages that are designed for a special purpose do not possess the same level of versatility and area of applicability than general-purpose languages. Hence, it is not surprising that all mentions of versatility of model transformation languages in our gathered literature paint MTLs as less versatile compared to GPLs (**P52** (*C124*), **P80** (*C125*), **P94** (*C127*)).

4.4 RQ2: Supporting evidence for advantages and disadvantages of MTLs

We found a number of different ways used by authors of our gathered literature to support their assertions. The largest portion of "supporting evidence" is made up of cited literature, i.e. a claim is followed by a citation that supposedly supports the claim.

The second way claims are supported is by example, i.e. authors implemented transformations in MTLs and/or GPLs and reported on their findings. Another aspect of this is relying on experience, i.e. authors state that from experience it is clear that some pronouncement is true or that it is a well-established fact within the community that a claim is true.

Third, there is empirical evidence, i.e. studies designed to measure specific effects of model transformation languages or case studies designed to gather the state of MTL usage in industry.

Last, there are those assertions that are not supported by any means. Authors simply suggest that an advantage or disadvantage exists. We assume that some claims made in this way implicitly rely on experience but do not state so. Nevertheless, since there is no way of testing this assumption we have to record such claims exactly the way they are made, without any evidence.

In the following sections, we will talk in detail about how each group of evidence is used in the literature to support claims about advantages or disadvantages of model transformation languages. As mentioned previously, Table 4 contains a complete overview over each claim and through what evidence the claim is supported.

4.4.1 Citation as evidence

Using citations to support statements is a core principle in research. It should therefore come as no surprise that citations are used to support claims about model transformation languages. An interesting aspect to explore for us was to trace how the cited literature supports the claim. For that, as stated in Sect. 3, we created a graphical representation to trace citations used as evidence through literature. The graph is shown in Fig. 7. It is inspired by UML syntax for object diagrams. The head of an "object" contains a *publication id*, while the body contains the categories for which advantages *(+)* or disadvantages *(–)* are claimed in the publication. Each category within the body is accompanied by an ID which can be used to find the corresponding claim within Table 4. We use different *borders* around publications to denote the type of evidence

provided by the publication and *arrows* from one category within a publication to a different publication stand for the use of a citation to support a claim. Lastly, if the content of a publication does not concern itself with model transformation languages but instead with DSLs, the publication id is followed by "*(DSL)*".

Our graph allows to easily gauge information about the following things:

- What publication claims an advantage or disadvantage of MTLs in which category?
- What type of evidence (if any) is used to support claims in a publication?
- Which exact claims are supported through the citation of what publication?

In the following, we discuss observations about citations as evidence that can be made with help from the citation graphs.

First, only a total of 25 citations, split among 12 out of the 58 gathered publications, are used to support claims. This constitutes less than ten percent of all assertions found during our literature review. Seven of the 25 citations cite a publication that itself only states claims without any evidence thereof (**P63**, **P94**, **P673**, **P674**, **P800**). A further 11 end in a publication that uses examples or experience (see also Sect. 4.4.3) (**P664**, **P665**, **P667**, **P671**, **P672**, **P676**, **P77**, **P64**, **P804**, **P801**). Next, there are 3 citations that cite publications which in turn cite further publications to support their claims (**P677**, **P675**), leaving only 4 citations that cite empirical studies (**P669**, **P670**, **P803**) (see also Sect. 4.4.2). To us, this is worrying because the practice of citing literature that only restates an assertion corrodes the confidence readers can have in citations as supporting evidence.

From the graph, it is clearly evident that there exists no single cited source for claims about model transformation languages. This is clearly indicated by the fact that only five publications (**P63**, **P77**, **P673**, **P675**, **P803**) are cited more than once; twice to be exact. And no publication is cited more than two times. Moreover, of those five publications **P675** and **P803** are each cited by a single publication, respectively. **P675** is cited twice by **P80** and **P803** by **P675**. Related thereto, nearly each claim, even within the same category, is being supported through different citations.

Furthermore, only claims about *conciseness*, *expressiveness*, *reuse & maintainability*, *tool support*, *performance* and statements that MTLs are *just better* are supported using citations. It is interesting to note that claims within these categories which are supported by citations are either all positive or all negative. This is not to say that there are no contrasting claims, see for example *C113* and *C116* in **P23**, only that, if citations are used for a category the supported claims are either all positive or all negative.

Another thing to note is that in some instances claims about model transformation languages are being supported by citing publications on domain-specific languages in general. This can be seen in **P80**. The claims *C60* and *C61* are both supported by a citation of **P675** which is a publication that concerns itself with DSLs. Interestingly, **P675** itself then cites both publications about DSLs (**P800**, **P801**, **803**) and a publication about model transformation languages (**P804**) to support claims stated within the publication.

Coming back to citations of empirical studies, we have to report that while there exist 4 citations of empirical studies only a single claim about model transformation languages (*C116* in **P23**) is actually supported thereby. This is due to **P803** being an empirical study about DSLs and **P669** and **P670** both being cited as evidence for *C116*.

Lastly, apart from those publications that only make a single claim, no publication supports all their claims using citations. Extreme cases of this can be seen in **P45** and **P52** which make a total of 16 claims, only supporting three of them with citations while leaving the other 13 unsubstantiated.

4.4.2 Empirical evidence

To our disappointment, we have to report a lack of overall empirical evidence for properties of model transformation languages. Only four publications (**P32**, **P59**, **P669**, **P670**) in our gathered literature assess characteristics of model transformations using empirical means (see Fig. 7 and Table 4). Of those four, only **P59** focuses on MTLs as its central research object, while the other three are case studies about MDA that happen to contain results about transformation languages. **P803** too is an empirical study, but as mentioned in Sect. 4.4.1 focuses on domain-specific languages in general not on MTLs. In order to provide the necessary context for scrutinizing the claims extracted from the publications, we provide a short overview over the central aspects of **P32**, **P59**, **P669**, **P670** in the following.

The study described in **P59** was comprised of a large-scale controlled experiment with over 78 subjects from two universities as well as a preliminary study with a single individual. Subjects had to solve 231 tasks using three different languages (ATL, QVT-O and Xtend). The tasks focused on one of three aspects in transformation development, namely comprehending an existing transformation, changing a transformation and creating a transformation from scratch. After analysing the results, the authors come to the disillusioning conclusion that there is "no statistically significant benefit of using a dedicated transformation language over a modern general-purpose language".

The authors of **P32** report on an empirical study on the efficiency and effectiveness of MDA. A total of 38 subjects, selected from a model-driven engineering course, were asked

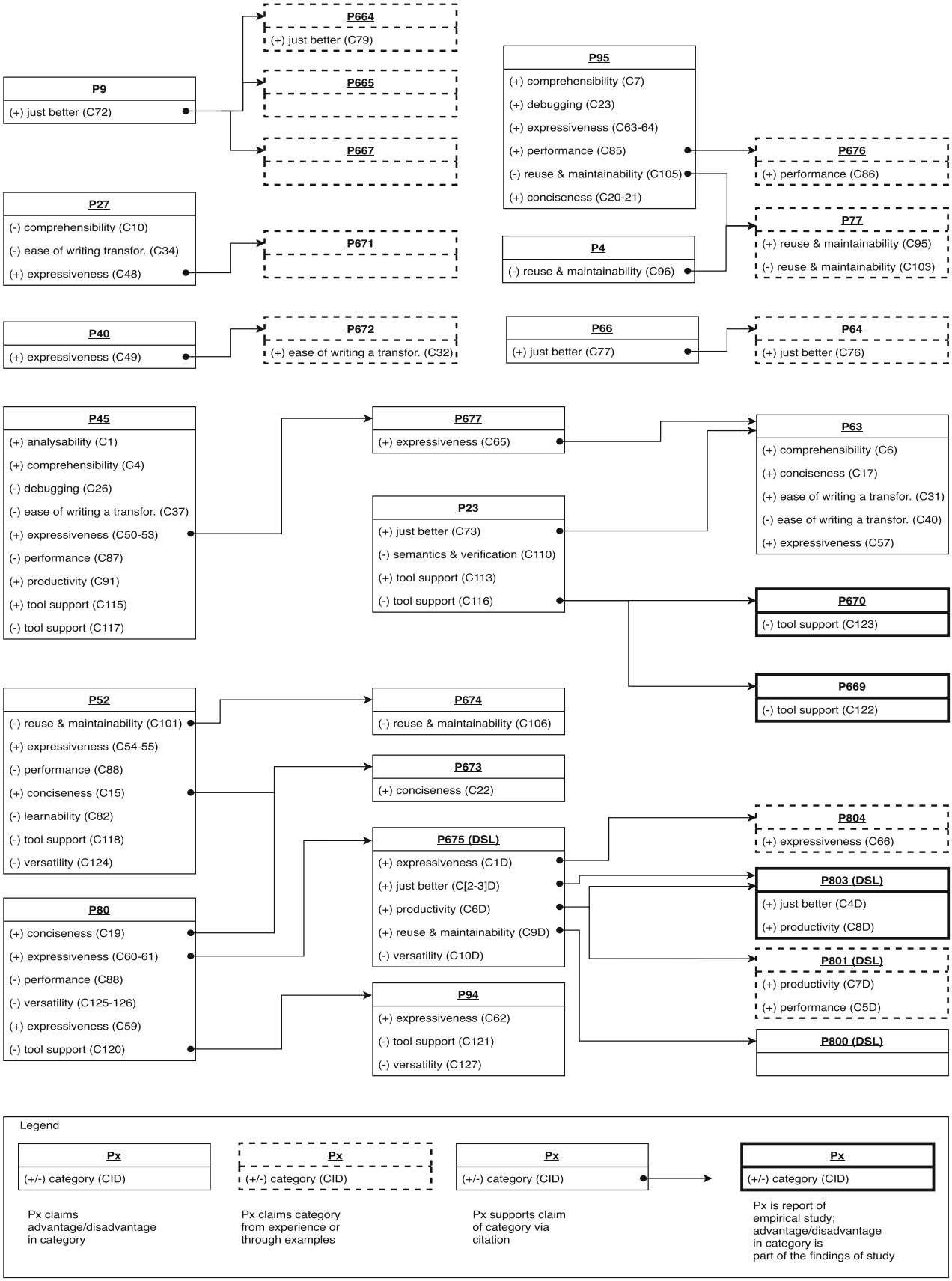


Fig. 7 Graph tracking citations of claims of various categories through literature

to implement the book-purchasing functionality of an e-book store system. Afterwards, the subjects evaluated the perceived efficiency and effectiveness of the used methodology. This also included questions about the used QVT language which was perceived as only marginally efficient.

Both **P669** and **670** are reports of industrial case studies. The objective of the study in **P669** was to investigate the state of practice of applying MDSE in industry. To achieve this, they collected data from tool evaluations, interviews and a survey. Four different companies were consulted to collect the data. Again while some reported results concerned themselves with transformations, model transformation languages were not explicitly discussed. Similarly, **P670** reports on an industrial case study involving two companies aiming to collect factors that influence the decision to adopt MDE. For that purpose, multiple preselected individuals at both companies were interviewed. Just as **P669**, the study did not directly focus on transformations or transformation languages.

As evident from Fig. 7, the results from **P32** and **P59** have yet to be used in the literature for supporting claims about MTLs. Since both of them have only been published recently, we are, however, optimistic about this prospect.

4.4.3 Evidence by example/experience

Using examples to demonstrate shortcomings of any kind has a long-standing tradition not only in informatics. Using examples to demonstrate an advantage, however, can result in less robust claims (especially toy or textbook examples Shaw [56]). As such, it is important to differentiate whether a claim is made by demonstrating a shortcoming or benefit.

In our gathered literature, ten publications use examples to support a claim. Interestingly, examples are mainly used to support broad claims about model transformation languages. This can be observed by the fact that **P34** and **P64** use examples to try and demonstrate that GPLs are not well suited for transforming models, while **P664**, **P665**, **P667**, **P672**, **P804** and **P676** try to demonstrate the general superiority of MTLs by showing examples of transformations written in MTLs. Other claims that are supported through examples are a demonstration of the reduction in code size when using rule-based MTLs in **P59** and statements about the extensive amount of reuse mechanisms for MTLs through listing gathered publications about the proposed mechanisms in **P77**.

Long-time practitioners of model transformation languages or programming languages in general often rely on their experience to make assertions about aspects of the language. And while the experience of long-term users can create valuable insights, it is still subjective and can therefore vary in accuracy. In our case, six publications directly state that their assertions come from experience. **P3** report on their experiences using different languages to implement transformations, coming to the conclusion that graphical rule definition is more intuitive, an experience shared by **P40**. **P43** name user feedback as grounds for claiming that visual syntax has advantages in comprehension but makes writing transformations more difficult. And **P672** share that they are under the impression that graph transformations are the superior method for defining refactorings.

Since experience is subjective, contradicting experiences are bound to occur sometime. While the authors of **P10** believe from experience that current MTLs are not abstract enough for expressing transformations, **P671** feel that the difficulty of writing transformations in a MTL does stem from the chosen MDD method rather than the syntax of the language.

4.4.4 No evidence

Figure 7 and especially Table 4 make it clear that a large portion of both positive and negative claims about model transformations are never substantiated. In fact, of the 127 claims ~69% are unsubstantiated. Adding those that are supported by a citation that in the end turns out to be unsupported as well brings the number up to ~77%. Particularly, the categories concerning the usability of MTLs such as *comprehensibility*, *ease of writing a transformation* and *productivity* lack meaningful evidence. All three of them being cornerstones of language engineers arguments for the superiority of model transformation languages make this especially worrisome.

We believe that a realization in the community about this fact is necessary. The necessity or superiority of model transformations has to be properly motivated. This means that it is not sufficient to claim advantages or disadvantages without providing at least some form of explanation on why this claim is valid (more on this in Sect. 5.3).

5 Discussion

In this section, we reflect on the previously presented findings. Our focus for this is fourfold. First, we feel it is necessary to draw parallels between our categorization and attributes of product quality. Next, we want to briefly discuss how claims are made in regards to transformation language features. Afterwards, a discussion about lack of empirical studies about properties of model transformation languages is warranted. And last we feel a discussion about the research direction for the community is also necessary.

5.1 Claims about model transformation languages in context of software quality

There are undeniable parallels between the categories we developed for claims and characteristics of software quality

as defined by *ISO/IEC 25010:2011* [35]. This can be seen by the fact that many of our categories can be directly placed within the characteristics of the software product quality model (namely functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, portability).

Both *expressiveness* and *semantics and verification* are part of functional suitability. *Performance* and *productivity* can be classified under performance efficiency. Furthermore are *comprehensibility*, *conciseness*, *debugging*, *ease of writing a transformation*, *learnability* and *tool support* part of *Usability*. Maintainability covers *analysability* and *reuse & maintainability*. And lastly, *extendability* and *versatility* can be classified under portability. This leaves only our generic category *just better* without a corresponding characteristic which is to be expected.

However, there are also compatibility, reliability and security which have no corresponding categories from our categorization. This does not necessarily mean that the current research is not focused on aspects related to these quality criteria. It instead suggests a lack of concrete statements regarding them. And while security is justifiably less of a concern for model transformation languages, both the compatibility of different approaches and their reliability should definitely be focused on (see also Sect. 5.4).

Lastly, even though most claims we collected during our review could be categorized within the software product quality model we opted to develop a classification based on the claims alone since we believe the resulting categories to be more specialized and allow for a more nuanced view on the subject matter than the generic characteristics defined by *ISO/IEC 25010:2011* [35].

5.2 Claims about model transformation languages in context of language features

An effort by us to categorize the extracted claims along an existing taxonomy of model transformation language features such as the one by Czarnecki and Helsen [22] failed because a large portion of claims (~70%) are made broadly without reference to specific features of MTLs that aid the advantage or disadvantage.

We suggest that claims on benefits and disadvantages of model transformation languages be made more specific and include mentions of the features that aid or hamper the benefits. For example, incrementality aids the performance of model transformations since only parts of a transformation have to be re-executed and bidirectional transformation languages provide special support for incremental execution giving them an edge in performance.

5.3 Lack of evidence for MTL advantages and disadvantages

The current literature exhibits a deficit in evidence (empirical or otherwise) for asserted properties of model transformation languages. We believe there to be several factors which can explain this lack of evidence.

First, designing and conducting rigorous studies to examine model transformation languages requires a substantial amount of time and effort. Studies are further complicated by the lack of easily available study subjects due to the community being relatively small compared to the body of general-purpose programming language users. The study described in **P59**, for example, had to be conducted over the timespan of three semesters and at two universities just to attain 78 subjects. And even when a pertinent number of study subjects is found, ensuring comparable levels of experience within the subjects is another challenge, even more so when collaborating with industrial partners [58].

Relying on the fact that transformation languages are DSLs and hence bear all the benefits that are proclaimed for those might also be a factor. Describing the advantages of DSLs in the introduction of a paper about transformation languages is far from uncommon in the literature. And while we too believe that there are benefits when using DSLs, we would caution against broad usage of the fact that model transformation languages are DSLs to claim them advantageous over general-purpose languages (as is done in publications such as **P29**, **P63** or **P804**), especially because the manpower that goes into the development of the ecosystems of GPLs far exceeds that of MTLs.

Another problem is that statements can become "established" facts by virtue of being cited by a paper which is in turn cited. Suppose one author claims that model transformation languages are more expressive than GPLs. A second author claims the same thing and references the first author to provide context. Next, a third author, assuming that the second author verifies their claim via the citation, cites the second author to support a similar claim. Over time, this can lead to the statement being treated as a fact rather than an assumption made multiple times. This can be seen on multiple occasions in Fig. 7. **P63** makes an unsubstantiated claim (*C57*) that the expressiveness of MTLs is superior to that of GPLs. This claim is then reiterated by **P677** (*C65*) citing **P67**. Lastly, **P677** is cited by **P45** to support their assertion about the expressiveness of model transformation languages (*C50-53*). Such a chain is not even the worst case in our results. The chain **P80** → **P675** → **P801-804** is even more worrisome, in that some of the claims stated in **P80** (*C75*) actually originate in claims about domain-specific languages from **675** (*C1D*). **P80** claims two advantages of MTLs using **P675** as reference. **P675** again uses citations to support their claims. However, the papers cited by **P675** do not make statements

about model transformation languages but DSLs in general. This shows how such chains can create a blurred factual picture. Moreover, in the presented cases it is still possible to find the origin of claims and realize how the claims were changed throughout the citation chains. If authors deemed it unnecessary to support claims that are "established" facts, this is no longer possible. Quite likely this is the case for a non-negligible number of publications (see Table 4) where no citations or any other substantiation for claimed properties of MTLs is given.

As previously described, it is not uncommon for authors to ascribe properties to model transformation languages due to them being DSLs. However, a language does not necessarily have to be more expressive, easier to use or easier to maintain simply by being domain specific. In fact we believe that everything about a DSL stands and falls with the domain itself as well as the design of the language. As a result, all advantages and disadvantages for DSLs, described in the literature, only define potential properties. Thus, it is necessary to evaluate advantages and disadvantages anew for each domain, especially in complex domains such as model transformations.

5.4 Research direction

In our opinion, the research community has to acknowledge that the current way of language development is not expedient. There needs to be a shift away from constant development of new features and transformation languages with, at best, prototypical evaluation. Tomaž Kosar, Bohra and Mernik [44] share this sentiment after a mapping study on the development of DSLs in general (see Sect. 6).

Instead, it is necessary to extensively evaluate current transformation languages, first to identify their actual strengths and weaknesses and then to compare these results with the expected (and desired) results to determine which aspects of MTLs still need improving.

We believe the categories from Sect. 4 to be a good reference for possible areas to evaluate.

It is not necessary to evaluate each category empirically: For some categories, empirical evaluation might not be sensible at all. Such categories include analysability, and semantics and verification for example, since there exist no universally accepted measures to base evaluation on. Additional literature reviews are also conceivable. Analogous to how **P77** gathered different reuse mechanisms, a comprehensive review of verification and analysis approaches can be useful to assess the *analysability* and *verifiability* of model transformation languages.

Designing and executing appropriate studies also entails significant effort which is why it becomes necessary to carefully weigh up which properties should be evaluated. Additionally, some categories should also be examined more urgently than others.

The *ease of writing a transformation* and *comprehensibility* are two such categories for which evaluation is most pressing. Also given that in the domain of programming languages (especially object-oriented programming), many studies exploring the comprehensibility and ease of use, such as Burkhardt et al. [15], Rein et al. [54], and Kurniawan and Xue [47], already exist. Study designs similar to the one described in **P59** are in our opinion most suitable for this purpose. This is supported by the fact that many studies for comparing programming languages follow a similar structure in that a common problem or task is solved in multiple languages and the resulting code is analysed [4,30,53]. It may also be useful to design the cases in such a way that the complete capabilities of the used transformation languages have to be used. In the study described in **P59**, for example advanced features such as QVTs *late resolve* were not part of the evaluation. Such a design can help to better understand if the most "advanced" features of transformation languages have practical value and how complex a GPL for these features is.

Comprehensibility can also be tested in isolation by requiring subjects to describe functionality of given transformations written in both a dedicated model transformation language and a GPL.

According to Mohagheghi et al. [51], one of the main motivations for adopting MDE in industry is to improve *productivity*; hence, we believe that evaluation of the productivity when using model transformation languages should be a focus too. Admittedly measuring productivity is a challenging task, a fact that has been observed as early as 1978 [37]. But since then, numerous ways have been proposed and tested out in practice [10,13] which should allow for productivity studies on MTLs to be carried out. A potential study into the productivity could require subjects to develop transformations in either a model transformation language or a general-purpose language within a certain time frame followed by measuring and comparing how productive the subjects were in both cases. Researchers can also draw from the large corpus of productivity studies on different aspects of programming, such as Wiger and Ab [66], Frakes and Succi [25] and Dieste et al. [23].

The *performance* of model transformations can have huge impact on development, especially when multiple transformations have to be executed in succession. Many language engineers already pay tribute to that fact by providing performance comparisons between their languages and other MTLs or general-purpose languages such as Java [32,46]. And the Transformation Tool Contest (TTC) provides a venue for comparing MTLs. However, we believe extensive comparisons between the performance of model transformation languages and general-purpose programming languages to

be necessary to abolish the prejudice that dedicated transformation languages cannot outperform current compilers. Comparison of performance between different programming languages that are used for the same purpose is a well-established practice demonstrated by comparisons between Java and C++ for robotics programming done by Gherardi, Brugali and Comotti [27] or C++ and F90 for scientific programming by Cary et al. [18]. Performance comparisons are also common practice in other domains such as GPU programming where specialized DSLs are used and performance is of high importance (Karimi et al. [24]). It is conceivable to compare the performance of transformations written in dedicated MTLs and GPLs by either manually solving the same tasks as described previously or by using existing mechanisms (for example Calvar et al. [17]) for transforming transformation scripts written in a MTL into GPL code.

We also believe that special focus needs to be given to the question of what model transformation languages are expected to achieve (such as easy synchronization of multiple artefacts or fast transformations through incremental transformations): first, because this can allow to direct more resources on evaluating relevant aspects of MTLs; and second, because model transformation languages will appear more streamlined and mature when the focus of development lies in improving their core features instead of overloading them with "experimental" features. An opinion Tomaž Kosar et al. [44] share is that this can enable practitioners to truly understand the effectiveness and efficiencies of DSLs.

6 Related work

To the best of our knowledge, there exists no other literature review that explores advantages and disadvantages of model transformation languages. There does, however, exist some literature that can be related to our work.

A closely related survey and open discussion about the future of model transformation languages was held by Cabot and Gérard [16]. They report on the results of an online survey and subsequent open discussion during the 12th edition of the International Conference on Model Transformations (ICMT'2019). The survey was designed to gather information about why developers used MTLs or why they hesitate to do so and what their predictions about the future of these languages were. An open discussion was held after the results of the online survey were presented to the audience at ICMT'2019. The results of the study point towards MTLs becoming less popular not only because of technical issues but also due to tooling and social issues as well as the fact that some GPLs have assimilated ideas from MTLs and thus making them less bad alternatives to writing transformations in dedicated languages.

Hutchinson et al. [34] conducted an empirical study into MDSE in industry. The authors used questionnaires and interviews to explore different factors that influence the success of MDSE in organizations and attempt to provide empirical evidence for hailed benefits of MDSE. They report on a total of over 250 questionnaire responses as well as interviews with 22 practitioners from 17 different companies. While the main focus of the study was on MDSE adoption in general, the authors do report on some findings regarding model transformations, such as negative influences of writing and testing transformations on the productivity and influences of transformations on the portability. However, no results regarding used transformation languages are included.

Mens and Gorp [49] propose a taxonomy for model transformation languages. They define groups of transformation languages based on answers to a set of questions. The answers are split into multiple subgroups themselves. The authors describe in great detail different possible characteristics within the groups. In part, this also includes listings of properties for transformation languages that fall into specific groups. The authors, however, have not provided any evidence or more precise explanations. Similarly, Czarnecki and Helsen [22] propose a classification framework for model transformation approaches based on several approaches such as VIATRA, ATL and QVT. The framework is given as a feature diagram to allow to explicitly highlight different design choices for transformations. At the top level, the feature model contains features such as rule organization, incrementality, directionality and tracing. Each feature and its sub-components are extensively discussed and demonstrated with examples of transformation tools that boast different aspects of the features. In contrast to the two described classifications, our study categorizes claims about MTLs on a qualitative dimension rather than on language features.

Kahani et al. [39] describe a classification and comparison of a total of 60 model transformation tools. Their classification differentiates tools based on two levels. The first level describes whether the tool is a model-to-model (M2M) or model-to-text (M2T) tool. The second level differentiates M2M tools based on their transformation approach meaning whether the approach is relational, operational or graph-based and M2T tools based on the underlying implementation approach meaning visitor-based, template-based or hybrid. Unlike our study, the described comparison focuses on comparing different model transformation tools on a technical basis based on six categories (general, model level, transformation, user experience, collaboration support and runtime requirements), while we focus on qualitative aspects of claims made throughout literature about any kind of dedicated model transformation language.

Van Deursen et al. [62] gathered an annotated bibliography on the premise of *domain-specific languages versus generic programming languages*. The bibliography con-

tains 73 different DSLs differentiated by their application domains: *Software Engineering*, *Systems Software*, *Multi-Media*, *Telecommunication* and *Miscellaneous*. Additionally, they provide a discussion of terminology as well as risks and benefits of DSLs. And while parts of the listed risks and benefits such as enhanced productivity or cost of education can be found in the listed advantages and disadvantages of our literature review, their bibliography does not contain any model transformation languages.

Tomaž Kosar et al. [44] report on the results of a systematic mapping study they conducted to understand the DSL research field, to identify research trends and to detect open issues. Their data comprised a total of 1153 candidates which they condensed into 390 publications for classification. The results from the study corroborate observations made during our literature review. The research community is mainly concerned with the development of new techniques, while research into the effectiveness of languages and empirical evaluations is lacking.

Tomaz Kosar et al. [45] describe an empirical study comparing a domain-specific language with a general-purpose language with a focus on learning, perceiving and evolving programs. The two languages considered were XAML as a DSL representative and the GPL C#. The experiment is comprised of 36 programmers which were asked to construct a graphical interface using both XAML and C# Forms. Afterwards, the subjects had to answer a questionnaire. In contrast to the results of **P59**, their results show a statistically significant advantage of DSLs for learning, comprehending and evolving programs.

Jakumeit et al. [36] provide an extensive overview over and comparison of 13 state-of-the-art transformation tools used in the TTC 2011. The authors give detailed descriptions of the tools based on a "Hello World" case posed at the contest. They also describe for what use cases the individual tools are best suited and provide a novel taxonomy based on which the tools are compared. The introduced taxonomy features many of the same categories we synthesized from the claims in our literature review, such as expressiveness, extendability, learnability, reuse and verification, but also other categories such as maturity and license.

7 Threats to validity

To ensure reproducibility and a high quality of the results, we followed a systematic approach as detailed in Sect. 3. However, possible threats to validity still remain. In this section we discuss these threats.

7.1 Internal validity

Internal validity describes the extent to which a casual conclusion based on the study is warranted. This validity is threatened by possible differences in the interpretation of our selection criteria. To alleviate the potential threat, two researchers independently applied the selection criteria and in cases of different decisions about the inclusion of a publication, full text cross-reading was applied.

A threat to the internal validity we could not meet with prevention measures was the fact that our categorization is based on certain defining expressions like *"expressive"* and *"versatile"*. It is possible that different authors ascribe different meanings to these phrases. While we believe that for most cases this is less of a problem, it is still a problem that we could not fully solve since not every publication defines their understanding of used phrases.

7.2 External validity

External validity describes the extent to which the findings of a study can be generalized. For structured literature reviews, a threat to this validity arises from the existence of relevant but undetected or excluded publications [20]. To mitigate this threat as much as possible, we used both automatic searches and exhaustive backward and forward snowballing. The automatic search was also conducted on multiple literature databases to broaden the field of searched literature. Furthermore, we employed a "when uncertain include" strategy for including publications, as well as less strict inclusion criteria which helped prevent relevant publications from being overlooked.

7.3 Construct validity

Construct validity describes the extent to which the right measures were obtained and whether the right scope was defined in relation to our research questions. The construct validity of our research is not under threat since the research questions define easily producible results. Cited advantages or disadvantages of model transformation languages can be directly extracted, and the same also holds for used evidence for claims.

7.4 Conclusion validity

Conclusion validity describes the extent to which conclusions based on data are reproducible.

Prior to the execution of our literature review, we defined a review protocol for all phases of the review. We followed the protocol rigorously to ensure reproducibility of the study. The protocol did not only include descriptions of how the review had to be conducted but also detailed how data should be extracted from the selected literature (see Sect. 3). It is of course possible that, with the passage of time, a repetition of the literature review can draw different conclusions due to the added body of literature between then and now.

8 Conclusion

In this study, we have reported on a systematic literature review intended to extract and categorize claims about model transformation languages as well as the current state of evaluation thereof. The goal of the study was to compile a comprehensive list and the categorization of positive and negative claims about model transformation languages. We further wanted to investigate the current state of evaluation of claims as well as identify gaps in the area of evaluation of MTLs.

We combed over 4000 publications for that purpose, 58 of which we selected for the study. To this end, we followed a rigorous process by using a combination of automatic searches on literature databases, exhaustive backward and forward snowballing and multiple researchers during the selection phase. The selected publications were combed for mentions of advantages and disadvantages of MTLs and evidence of the stated claims. Lastly, we analysed and discussed the extracted claims and evidence to: (i) provide an overview over claimed advantages and disadvantages and their origin, (ii) the current state of evidence thereof and (iii) identify areas where further research is necessary.

We conclude that: (i) the current literature claims many advantages of MTLs but also points towards deficits owed to the mostly experimental nature of the languages and its limited domain, (ii) there is insufficient evidence for and (iii) research about properties of model transformation languages.

The results of our study suggest that there is much to be done in terms of evaluation of model transformation languages and that effort that is currently being invested into the development of new features might be better spent evaluating the state of the art in hopes of ascertaining both what current MTLs are lacking most and where their strengths really lie.

Acknowledgements Open Access funding provided by Projekt DEAL.

A Overview over all extracted claims

See Table 4.

Table 4 Overview over claims per category

Category	Valuation	CID	Claim	Publication	Evidence
Analysability	Positive	C1	Declarative MTLs lend themselves to automatic analysis	**P45**	–
Comprehensibility	Positive	C2	Based on user feedback, it was identified that visual syntax is beneficial when reading a transformation program	**P43**	Experience
		C3	Bidirectional transformation languages have an advantage in comprehensibility	**P44**	–
		C4	Rules written in a declarative MTL are more easily understood in isolation and in combination	**P45**	–
		C5	An observation made from the empirical data is that context selection and identification are easier for subjects working with MTLs than with GPLs	**P59**	Empirical study

Table 4 continued

Category	Valuation	CID	Claim	Publication	Evidence
		C6	There are perceived cognitive gains of graphical representations compared to fully textual representations of transformations shown by for example the appeal of UMLs graphical representation of models	**P63**	–
		C7	Model transformation languages incorporate high-level abstractions that make them more understandable than GPLs	**P95**	–
	Negative	C8	Comprehensibility of transformation logic is hampered as current transformation languages provide only a limited view on a transformation problem. For example, graph transformation approaches only reveal parts of the meta-model	**P22**	–
		C9	Most MTLs lack convenient facilities for understanding the transformation logic	**P22**	–
		C10	Model transformation languages require specific skills and as a result are hard to understand for many stakeholders	**P27**	–
		C11	Large and heterogeneous models lead to poorly understandable transformation code due to missing language concepts to master complexity	**P29**	–
		C12	Graph transformations defined on abstract syntax are hard to read because the user has to be familiar with meta-model that defines the abstract syntax	**P70**	–
		C13	Purely graph-based transformation languages can become complex and hard to read	**P70**	–
Conciseness	Positive	C14	General-purpose languages lack simplicity because of how transformations are defined	**P3**	Examples
		C15	GPLs do not allow developers to conveniently express model manipulation concepts and the loss of abstraction in GPLs may give rise to accidental complexities	**P52**	Cites **P673**
		C16	Transformations implemented in the pre-study using rule-based MTLs were up to 48% smaller than corresponding Java variants	**P59**	Preliminary study
		C17	Declarative approaches make language more concise	**P63**	–
		C18	Graphical notation in MTLs is concise	**P75**	–

Table 4 continued

Category	Valuation	CID	Claim	Publication	Evidence
		C19	GPLs do not conveniently express model manipulation concepts and the loss of abstraction can give rise to accidental complexities	**P80**	Cites **673**
		C20	Model transformation languages incorporate high-level abstractions that make them more concise than GPLs	**P95**	–
		C21	Model transformation languages are more concise	**P95**	–
		C22	MDE and model transformation languages such as QVT help to reduce complexity	**P673**	–
Debugging	Positive	C23	Debuggers for MTLs are likely better than those for GPLs for debugging transformations since it is questionable whether the call stacks produced by debuggers of GPLs are meaningful for the developer	**P95**	–
	Negative	C24	Although numerous transformation languages exist, they lack convenient facilities for supporting debugging and understanding of the transformation logic	**P22**	–
		C25	In ATL, TGGs and QVT-R correspondence is defined on a higher level of abstraction compared to on what execution engines operate Thus debugging is limited on the lower level of the execution engines not on the level of the language definition	**P22**	–
		C26	In declarative model transformation languages debugging is more difficult than in imperative ones	**P45**	–
		C27	Model transformation languages lack proper debugging support since implementation cost is high	**P90**	–
Ease of writing a transformation	Positive	C28	We found graphical rule definition far more intuitive than syntax-based definition	**P3**	Experience
		C29	Model transformation languages ease development efforts by offering succinct syntax to query from and map model elements between different modelling domains	**P29**	–
		C30	Model transformation languages make it easy to work with models	**P50**	–
		C31	Imperative transformation approaches offer a familiar paradigm, that is, sequence, selection, and iteration	**P63**	–

Table 4 continued

Category	Valuation	CID	Claim	Publication	Evidence
		C32	It is our impression that, in general, graph transformations offer significantly better support for the specification and implementation of modelling guidelines and refactorings	**P672**	Experience
	Negative	C33	Imperative MTLs induce overhead code because many issues have to be accomplished explicitly, e.g. specification of control flow	**P22**	–
		C34	Traditional transformation languages require specific skills to be able to write transformations	**P27**	–
		C35	To be able to write transformations, one has to be a transformation expert	**P28**	–
		C36	Based on user feedback we identified that writing a transformation program with a graphical syntax can be complicated	**P43**	Experience
		C37	The syntax of declarative MTLs is unfamiliar for many developers	**P45**	–
		C38	Model transformation languages that define transformations on meta-model level require deep understanding of the meta-model	**P56**	–
		C39	There is no sufficient (statistically significant) evidence of a general advantage of specialized model transformation languages (ATL, QVT-O) over a modern GPL (Xtend)	**P59**	Empirical study
		C40	Developers are generally more comfortable with encoding complicated (transformation) algorithms in procedural languages	**P63**	–
		C41	First of all, some of us are not convinced that the usage of a visual notation has significant advantages compared to a textual notation. A textual notation is more compact, simplifies all kinds of version and configuration management tasks, and does not force its users to spend hours beautifying the layout of huge diagrams	**P672**	–
Expressiveness	Positive	C42	Rule-based approaches seem to be less error-prone compared to a manual implementation of pattern matching for each transformation in a general-purpose language	**P2**	–
		C43	Model transformation languages can hide details like traversing behind simple syntax	**P15**	–

Table 4 continued

Category	Valuation	CID	Claim	Publication	Evidence
		C44	Model transformation languages can hide traces behind simple syntax	**P15**	–
		C45	Model transformation languages can hide rule triggering behind simple syntax	**P15**	–
		C46	Model transformation languages can hide rule ordering behind simple syntax	**P15**	–
		C47	Model transformation languages can hide complex transformation algorithms behind a simple syntax	**P15**	–
		C48	Model transformation languages hide transformation complexity and burden from user	**P27**	Cites **P671**
		C49	Graph transformations generally offer a significantly better support for the specification and implementation of modelling guidelines and refactorings	**P40**	Cites **P672**
		C50	Declarative MTLs allow automatic traceability management	**P45**	Cites **P677**
		C51	Declarative model transformation languages allow for implicit rule ordering lessening the load on developer	**P45**	Cites **P677**
		C52	Declarative MTLs can do implicit target object creation	**P45**	Cites **P677**
		C53	Declarative MTLs allow for implicit source model traversal	**P45**	Cites **P677**
		C54	Model transformation languages syntax is more specific	**P52**	–
		C55	GPLs do not allow developers to conveniently express model manipulation concepts	**P52**	–
		C56	We found that copying complex structures is more effective in MTLs	**P59**	Empirical study
Expressiveness	Positive	C57	General-purpose languages lack suitable abstractions for specifying transformations	**P63**	–
		C58	Graph-based MTLs are especially popular due to their high expressive power	**P70**	–
		C59	Model transformation languages have more specific language constructs	**P80**	–
		C60	Model transformation languages have a higher level of abstraction which leads to gains in expressiveness over GPLs	**P80**	Cites **P675**
		C61	Model transformation languages are easier to use than GPLs	**P80**	Cites **P675**

Table 4 continued

Category	Valuation	CID	Claim	Publication	Evidence
		C62	Model transformation languages transformation constructs are more specific	**P94**	–
		C63	From our perspective, automatic handling/resolution of traces by transformation engine is one of the major features that make existing MTLs better suited for model transformations than GPLs	**P95**	–
		C64	General-purpose languages lack sufficient transformation concepts	**P95**	–
		C1D	DSLs trade expressiveness in a limited domain for generality	**P675**	Cites **P804**
		C65	GPLs lack suitable abstractions for specifying transformations	**P677**	Cites **P63**
		C66	With a DSL/MTL, a programmer can express their objective in a concise manner using a language that is much higher in expressiveness than that typically offered in a transitional programming language	**P804**	–
	Negative	C67	Having written several transformation, we have identified that current MTLs are too low a level of abstraction for succinctly expressing transformations between DSLs because they demonstrate several recurring patterns that have to be reimplemented each time	**P10**	Experience
		C68	Having written several transformation, we have identified that mapping a single element to fragments of multiple elements has to be done programmatically which is counterintuitive and error-prone	**P10**	Experience
		C69	OCL constraints cannot be transformed in MTLs	**P32**	Empirical Study
		C70	There is no mechanism for describing and/or storing information about the properties of a transformation	**P33**	–
Extendability	Negative	C71	Extending model transformation languages is difficult	**P50**	–
Just better	Positive	C72	GGT (graph grammar and graph transformation) are a powerful technique for specifying complex transformations	**P9**	Cites **P664**-**P666**
		C73	General-purpose programming languages are not suitable for defining model transformations	**P23**	Cites **P63**

Table 4 continued

Category	Valuation	CID	Claim	Publication	Evidence
		C74	GPLs are not well suited for model migration	**P34**	Examples
		C75	Dedicated MTLs offer the most potential transformation approach because the languages can be tailored for the purpose	**P63**	–
		C76	In order to transform models in a GPL, one has to add increasing amounts of machinery, e.g. to keep track of which elements have already been transformed. This leads to the assumption that model transformations cannot be sensibly written in a standard programming language	**P64**	Examples
		C77	Model transformations present a number of problems which imply that dedicated approaches are required	**P66**	Cites **P64**
		C78	The current consensus is that specialized languages with a mixture of declarative and imperative constructs are most suitable for specifying model transformations	**P86**	–
		C79	With the help of an example, we have shown that GGT (graph grammar and graph transformation) can be used to transform PIMs into PSMs	**P664**	Examples
		C2D	DSLs open up the application domain to a larger group of developers	**P675**	Cites **P803**
		C3D	Domain-specific languages increase the ease of use	**P675**	Cites **P803**
		C4D	When using DSLs, less errors are made	**P803**	Empirical Study
	Negative	C80	General-purpose MTLs are not well suited for model migration since there is additional overhead but dedicated migration languages are	**P34**	Examples
Learnability	Negative	C81	The generality of general-purpose MTLs can have the effect of making them less approachable and create a steep learning curve for non-expert users	**P30**	–
		C82	Users have to learn multiple similar, but not always consistent, languages, which requires considerable time to learn	**P52**	–
		C83	Model transformation languages have a steep learning curve	**P58**	–
		C84	One has to learn a completely new language to transform models with MTLs	**P81**	–
Performance	Positive	C85	Model transformation languages are more performant	**P95**	Cites **P676**
		C86	GrGen shows a better performance of transformations than Java	**P676**	Samples
		C5D	DSLs have better performance	**P801**	–

Table 4 continued

Category	Valuation	CID	Claim	Publication	Evidence
	Negative	C87	Declarative MTLs have performance problems	**P45**	–
		C88	The performance of model transformation languages is a shortcoming that may make users feel limited	**P52**	–
		C89	MTLs have worse performance	**P80**	–
Productivity	Positive	C90	Model transformation languages being DSLs improve the productivity	**P29**	–
		C91	Declarative MTLs increase programmer productivity	**P45**	–
		C6D	DSLs increase productivity	**P675**	Cites **P801**, **P803**
		C7D	Using DSLs increases productivity	**P801**	Examples
		C8D	Using DSLs increases productivity	**P803**	Empirical Study
	Negative	C92	The perceived effectiveness of model transformation languages is bad	**P32**	Empirical Study
		C93	Productivity of GPL development might be higher since expert users for GPLs are easier to hire	**P59**	–
Reuse and Maintainability	Positive	C94	Bidirectional model transformations have an advantage in maintainability	**P44**	–
		C95	There exists a plethora of reuse mechanisms for MTLs	**P77**	Literature review
		C9D	Domain-specific languages reduce the maintenance costs	**P675**	Cites **P800**
	Negative	C96	Reuse is sparse, transformations are written from scratch every time because meta-models differ slightly	**P4**	Cites **P77**
		C97	Having written several transformation, we have identified that recurring patterns have to be implemented from scratch every time	**P10**	Experience
		C98	There exists no module concept for model transformation languages that allows programmers to control information hiding and strictly declare model and code dependencies at module interface	**P29**	–
		C99	Model transformation languages lack sophisticated reuse mechanisms	**P33**	–
		C100	Unfortunately, the definition of model transformations is normally a type-centric activity, thus making their reuse for other meta-models difficult	**P41**	–
		C101	Evolving and maintaining MTL requires effort	**P52**	Cites **P674**

Table 4 continued

Category	Valuation	CID	Claim	Publication	Evidence
		C102	The emphasis of MDE on using DSLs has caused a proliferation of meta-models. In this scenario, developing a transformation for a new meta-model is usually performed manually with no reuse, even if comparable transformations for similar meta-models exist	**P60**	–
		C103	There are barriers such as insufficient abstraction of reuse mechanisms from meta-models that hamper reuse	**P77**	Literature review
		C104	There is little support for reusing model transformations in different contexts since they are tightly coupled to the meta-models they are defined upon	**P78**	–
		C105	Reuse of model transformations is hardly established in practice	**P95**	Cites **P77**
		C106	Developing these new languages to a sufficient degree of maturity is an enormous effort which includes for example construction and optimisation of compilers	**P674**	–
Semantics and Verification	Positive	C107	Bidirectional transformation languages have an advantage in verification	**P44**	–
	Negative	C108	For existing relational model transformation approaches, it is usually not really clear under which constraints particular implementations really conform to the formal semantics	**P21**	–
		C109	Comprehensive verification support of model transformations is missing	**P22**	–
		C110	There is a semantic difference between a typical programming language and formalisms that support bidirectionality and change propagation such as TGGs	**P23**	–
		C111	Most transformation languages have no formal semantics to add detailed specifications on the expected behaviour	**P39**	–
		C112	The semantics of many model transformation languages is not formally defined	**P58**	–

Table 4 continued

Category	Valuation	CID	Claim	Publication	Evidence
Tool Support	Positive	C113	Internal MTLs can inherit tool support of general-purpose host language	**P23**	–
		C114	Tool support for external transformation languages is potentially more powerful than for internal MTL or GPL because it can be tailored to the DSL	**P44**	–
		C115	Declarative MTLs provide opportunities for specialized tool support	**P45**	–
	Negative	C116	Model transformation languages lack tool support	**P23**	Cites **P669**,**P670**
		C117	Declarative MTLs lack libraries and tool support	**P45**	–
		C118	Model transformation languages lack tool support	**P52**	–
		C119	Supporting tools for MTLs have not the same level of maturity as for GPLs	**P74**	–
		C120	Model transformation languages have worse tool support	**P80**	Cites **P94**
		C121	Tool support for external MTLs has to be developed which entails extra effort	**P94**	–
		C122	Tool support for model transformations is not as mature as subjects would like	**P669**	Empirical study
		C123	Tool support for model transformations is not great	**P670**	Empirical study
Versatility	Negative	C124	The syntax of model transformation languages is less versatile	**P52**	–
		C125	Model transformation languages are less versatile than GPLs	**P80**	–
		C126	Model transformation languages have less versatile language constructs	**P80**	–
		C127	Model transformation languages constructs are less versatile	**P94**	–
		C10D	DSLs are less general than general-purpose programming languages	**P675**	–

B SLR results

P2 Patzina, Sven and Lars Patzina (2012). "A Case Study Based Comparison of ATL and SDM". In: *Proceedings of the 4th International Conference on Applications of Graph Transformations with Industrial Relevance*. AGTIVE 2011. DOI: https://doi.org/10.1007/978-3-642-34176-2_18.

P3 Stephan, Matthew and Andrew Stevenson (2009). *A Comparative Look at Model Transformation Languages*. Tech. rep. Software Technology Laboratory at Queens University. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.712.2983.

P4 Cuadrado, J. S., E. Guerra, and J. de Lara (2014). "A Component Model for Model Transformations". In: *IEEE Transactions on Software Engineering*. DOI: https://doi.org/10.1109/TSE.2014.2339852.

P9 Agrawal, Aditya, Gabor Karsai, and Feng Shi (2003). "A UML-based graph transformation approach for implementing domain-specific model transformations". In: *International Journal on Software and Systems Modeling*. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.152.1226.

P10 Johannes, Jendrik et al. (2009). "Abstracting Complex Languages through Transformation and Composition". In: *Model Driven Engineering Languages and Systems*. MODELS 2009. DOI: https://doi.org/10.1007/978-3-642-04425-0_41.

P15 Jouault, Frédéric et al. (2008). "ATL: A model transformation tool". In: *Science of Computer Programming*. DOI: https://doi.org/10.1016/j.scico.2007.08.002.

P21 Giese, Holger, Stephan Hildebrandt, and Leen Lambers (2014). "Bridging the gap between formal semantics and implementation of triple graph grammars". In: *Software & Systems Modeling*. DOI: https://doi.org/10.1007/s10270-012-0247-y.

P22 Schoenboeck, Johannes et al. (2010). "Catch Me If You Can - Debugging Support for Model Transformations". In: *Models in Software Engineering*. MODELS 2009. DOI: https://doi.org/10.1007/978-3-642-12261-3_2.

P23 Hinkel, Georg and Erik Burger (2019). "Change propagation and bidirectionality in internal transformation DSLs". In: *Software & Systems Modeling*. DOI: https://doi.org/10.1007/s10270-017-0617-6.

P27 Sottet, J. and A. Vagner (2014). "Defining Domain Specific Transformations in Human-Computer interfaces development". In: *2014 2nd International Conference on Model-Driven Engineering and Software Development*. MODELSWARD '14. URL: https://ieeexplore.ieee.org/abstract/document/7018471.

P28 Acretoaie, Vlad (2013). *Delivering the Next Generation of Model Transformation Languages and Tools*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.708.6612.

P29 Rentschler, Andreas et al. (2014). "Designing Information Hiding Modularity for Model Transformation Languages". In: *Proceedings of the 13th International Conference on Modularity*. MODULARITY '14. DOI: https://doi.org/10.1145/2577080.2577094.

P30 Steel, Jim and Robin Drogemuller (2011). "Domain-Specific Model Transformation in Building Quantity Take-Off". In: *Model Driven Engineering Languages and Systems*. MODELS 2011. DOI: https://doi.org/10.1007/978-3-642-24485-8_15.

P32 Shin, Shin-Shing (2019). "Empirical study on the effectiveness and efficiency of model-driven architecture techniques". In: *Software & Systems Modeling*. DOI: https://doi.org/10.1007/s10270-018-00711-y.

P33 Criado, Javier et al. (2015). "Enabling the Reuse of Stored Model Transformations Through Annotations". In: *Theory and Practice of Model Transformations*. ICMT 2015. DOI: https://doi.org/10.1007/978-3-319-21155-8_4.

P34 Rose, Louis M. et al. (2014). "Epsilon Flock: a model migration language". In: *Software & Systems Modeling*. DOI: https://doi.org/10.1007/s10270-012-0296-2.

P39 Berramla, K., E. A. Deba, and M. Senouci (2015). "Formal validation of model transformation with Coq proof assistant". In: *2015 First International Conference on New Technologies of Information and Communication*. NTIC 2015. DOI: https://doi.org/10.1109/NTIC.2015.7368755.

P40 Legros, Elodie et al. (2009). "Generic and reflective graph transformations for checking and enforcement of modeling guidelines". In: *Journal of Visual Languages & Computing* 4. DOI: https://doi.org/10.1016/j.jvlc.2009.04.005.

P41 Sánchez Cuadrado, Jesús, Esther Guerra, and Juan de Lara (2011). "Generic Model Transformations: Write Once, Reuse Everywhere". In: *Theory and Practice of Model Transformations*. ICMT 2011. DOI: https://doi.org/10.1007/978-3-642-21732-6_5.

P43 Strüber, Daniel et al. (2017). "Henshin: A Usability-Focused Framework for EMF Model Transformation Development". In: *Graph Transformation*. ICGT 2017. DOI: https://doi.org/10.1007/978-3-319-61470-0_12.

P44 Wider, Arif (2014). "Implementing a Bidirectional Model Transformation Language as an Internal DSL in Scala". In: *EDBT/ICDT Workshops*. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.428.9439.

P45 Lawley, Michael and Kerry Raymond (2007). "Implementing a Practical Declarative Logic-based Model Transformation Engine". In: *Proceedings of the 2007 ACM Symposium on Applied Computing*. SAC '07. DOI: https://doi.org/10.1145/1244002.1244216.

P50 Liepiņš, Renārs (2012). "Library for Model Querying: IQuery". In: *Proceedings of the 12th Workshop on OCL and Textual Modelling*. OCL '12. DOI: https://doi.org/10.1145/2428516.2428522.

P52 Krikava, Filip, Philippe Collet, and Robert France (2014). "Manipulating Models Using Internal Domain-Specific Languages". In: *Symposium On Applied Computing*. SAC '14. DOI: https://doi.org/10.1145/2554850.2555127.

P56 Sun, Yu, Jules White, and Jeff Gray (2009). "Model Transformation by Demonstration". In: *Model Driven Engineering Languages and Systems*. MODELS 2009. DOI: https://doi.org/10.1007/978-3-642-04425-0_58.

P58 Irazábal, Jerónimo and Claudia Pons (2010). "Model Transformation Languages Relying on Models as ADTs". In: *Software Language Engineering*. SLE 2009. DOI: https://doi.org/10.1007/978-3-642-12107-4_10.

P59 Hebig, Regina et al. (2018). "Model Transformation Languages Under a Magnifying Glass: A Controlled Experiment with Xtend, ATL, and QVT". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. DOI: https://doi.org/10.1145/3236024.3236046.

P60 Lara, Juan de et al. (2018). "Model Transformation Product Lines". In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '18. DOI: https://doi.org/10.1145/3239372.3239377.

P63 Sendall, S. and W. Kozaczynski (2003). "Model transformation: the heart and soul of model-driven software development". In: *IEEE Software*. DOI: https://doi.org/10.1109/MS.2003.1231150.

P64 Tratt, Laurence (2005). "Model transformations and tool integration". In: *Software & Systems Modeling*. DOI: https://doi.org/10.1007/s10270-004-0070-1.

P66 — (2007). "Model transformations in MT". In: *Science of Computer Programming*. DOI: https://doi.org/10.1016/j.scico.2007.05.003.

P70 Baar, Thomas and Jon Whittle (2007). "On the Usage of Concrete Syntax in Model Transformation Rules". In: *Perspectives of Systems Informatics*. PSI 2006. DOI: https://doi.org/10.1007/978-3-540-70881-0_10.

P74 Sánchez Cuadrado, J., E. Guerra, and J. de Lara (2015). "Quick fixing ATL model transformations". In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems*. MODELS '15. DOI: https://doi.org/10.1109/MODELS.2015.7338245.

P75 Li, Dan, Xiaoshan Li, and Volker Stolz (2011). "QVT-based Model Transformation Using XSLT". In: *SIGSOFT Softw. Eng. Notes*. DOI: https://doi.org/10.1145/1921532.1921563.

P77 Kusel, A. et al. (2015). "Reuse in model-to-model transformation languages: are we there yet?" In: *Software & Systems Modeling*. DOI: https://doi.org/10.1007/s10270-013-0343-7.

P78 Wimmer, Manuel et al. (2011). "Reusing Model Transformations across Heterogeneous Metamodels". In: *ECEASST*. DOI: https://doi.org/10.14279/tuj.eceasst.50.722.

P80 Křikava, Filip, Philippe Collet, and Robert B. France (2014). "SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations". In: *Model-Driven Engineering Languages and Systems*. MODELS 2014. DOI: https://doi.org/10.1007/978-3-319-11653-2_35.

P81 Akehurst, D. H. et al. (2006). "SiTra: Simple Transformations in Java". In: *Model Driven Engineering Languages and Systems*. MODELS 2006. DOI: https://doi.org/10.1007/11880240_25.

P86 Kolovos, Dimitrios S., Richard F. Paige, and Fiona A. C. Polack (2008). "The Epsilon Transformation Language". In: *Theory and Practice of Model Transformations*. ICMT 2008. DOI: https://doi.org/10.1007/978-3-540-69927-9_4.

P90 Sánchez Cuadrado, Jesús, Esther Guerra, and Juan de Lara (2014). "Towards the Systematic Construction of Domain-Specific Transformation Languages". In: *Modelling Foundations and Applications*. ECMFA 2014. DOI: https://doi.org/10.1007/978-3-319-09195-2_13.

P94 George, Lars, Arif Wider, and Markus Scheidgen (2012). "Type-Safe Model Transformation Languages as Internal DSLs in Scala". In: *Theory and Practice of Model Transformations*. ICMT 2012. DOI: https://doi.org/10.1007/978-3-642-30476-7_11.

P95 Hinkel, Georg, Thomas Goldschmidt, et al. (2019). "Using internal domain-specific languages to inherit tool support and modularity for model transformations". In: *Software & Systems Modeling*. DOI: https://doi.org/10.1007/s10270-017-0578-9.

P664 Agrawal, Aditya, Tihamer Levendovszky, et al. (2002). "Generative programming via graph transformations in the model-driven architecture". In: *In OOPSLA 2002 Workshop in Generative Techniques in the context of Model Driven Architecture*. OOPSLA '02.

URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.70.4824.

P665 Afimann, Uwe (1996). "How to uniformly specify program analysis and transformation with graph rewrite systems". In: *Compiler Construction*. CC 1996. DOI: https://doi.org/10.1007/3-540-61053-7_57.

P667 Radermacher, Ansgar (2000). "Support for Design Patterns through Graph Transformation Tools". In: *Applications of Graph Transformations with Industrial Relevance*. AGTIVE 1999. DOI: https://doi.org/10.1007/3-540-45104-8_9.

P669 Mohagheghi, Parastoo et al. (2013). "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases". In: *Empirical Software Engineering*. DOI: https://doi.org/10.1007/s10664-012-9196-x.

P670 Staron, Miroslaw (2006). "Adopting Model Driven Software Development in Industry – A Case Study at Two Companies". In: *Model Driven Engineering Languages and Systems*. MODELS 2006. DOI: https://doi.org/10.1007/11880240_5.

P671 Panach, José Ignacio, Óscar Pastor, and Nathalie Aquino (2011). "A Model for Dealing with Usability in a Holistic MDD Method". In: *User Interface Description Language, Lisbon, Portugal*. UIDL '11.

P672 Amelunxen, Carsten et al. (2008). "Checking and Enforcement of Modeling Guidelines with Graph Transformations". In: *Applications of Graph Transformations with Industrial Relevance*. AGTIVE 2007. DOI: https://doi.org/10.1007/978-3-540-89020-1_22.

P673 Schmidt, Douglas (2006). "Guest Editor's Introduction: Model-Driven Engineering". In: *COMPUTER-IEEE COMPUTER SOCIETY*. DOI: https://doi.org/10.1109/MC.2006.58.

P674 Chafi, Hassan et al. (2010). "Language Virtualization for Heterogeneous Parallel Computing". In: *ACM Sigplan Notices*. DOI: https://doi.org/10.1145/1932682.1869527.

P675 Mernik, Marjan, Jan Heering, and Anthony M. Sloane (2005). "When and How to Develop Domain-specific Languages". In: *ACM computing surveys (CSUR)*. DOI: https://doi.org/10.1145/1118890.1118892.

P676 Gorp, Pieter Van and Louis M. Rose (2013). The Petri-Nets to Statecharts Transformation Case. DOI: https://doi.org/10.4204/EPTCS.135.3.

P677 Mens, Tom and Pieter Van Gorp (2006). "A Taxonomy of Model Transformation". In: *Electronic Notes in Theoretical Computer Science (GraMoT 2005)*. DOI: https://doi.org/10.1016/j.entcs.2005.10.021.

P800 Herndon, R. M. and V. A. Berzins (1988). "The realizable benefits of a language prototyping language". In: *IEEE Transactions on Software Engineering*. DOI: https://doi.org/10.1109/32.6159.

P801 Batory, Don, Jeff Thomas, and Marty Sirkin (1994). "Reengineering a Complex Application Using a Scalable Data Structure Compiler". In: *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT '94. DOI: https://doi.org/10.1145/193173.195299.

P803 Kieburtz, Richard B. et al. (1996). "A Software Engineering Experiment in Software Component Generation". In: *Proceedings of the 18th International Conference on Software Engineering*. ICSE'96. DOI: https://doi.org/10.1109/ICSE.1996.493448.

P804 Gray, J. and G. Karsai (2003). "An examination of DSLs for concisely representing model traversals and transformations". In: *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*. HICSS '03. DOI: https://doi.org/10.1109/HICSS.2003.1174892.

References

1. Alves, R., Nunes, N.J.: Ceiling and threshold of paas tools: the role of learnability in tool adoption. In: International Conference on Human-Centred Software Engineering. HESSD 2016. (2016). https://doi.org/10.1007/978-3-319-44902-9_21
2. van Amstel, M.F., van den Brand, M.G.J.: Model transformation analysis: staying ahead of the maintenance nightmare. In: Theory and practice of model transformations. ICMT 2011. (2011). https://doi.org/10.1007/978-3-642-21732-6_8
3. Arendt, T. et al.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Model Driven Engineering Languages and Systems. MODELS 2010. (2010). https://doi.org/10.1007/978-3-642-16145-2_9
4. Aruoba, S.B., Fernandez-Villaverde, J.: A comparison of programming languages in economics. Technical report National Bureau of Economic Research, Inc. (2014). https://EconPapers.repec.org/RePEc:nbr:nberwo:20263
5. Auer, F., Felderer, M.: Current state of research on continuous experimentation: a systematic mapping study. In: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). (2018). https://doi.org/10.1109/SEAA.2018.00062
6. Badampudi, D., Wohlin, C., Petersen, K.: Experiences from using snowballing and database searches in systematic literature studies. In: Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering. EASE '15. (2015). https://doi.org/10.1145/2745802.2745818
7. Balogh, A., Varro, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: Proceedings of the 2006 ACM Symposium on Applied Computing. SAC '06. (2006). https://doi.org/10.1145/1141277.1141575
8. Barat, S. et al.: A model-based approach to systematic review of research literature. In: Proceedings of the 10th Innovations in Software Engineering Conference. ISEC '17. (2017). https://doi.org/10.1145/3021460.3021462
9. Barb, A.S., et al.: A statistical study of the relevance of lines of code measures in software projects. In: Innovations in Systems and Software Engineering. (2014). https://doi.org/10.1007/s11334-014-0231-5

10. Basili, V., Reiter, R.: An investigation of human factors in software development. In: Computer. (1979). https://doi.org/10.1109/MC.1979.1658573
11. Basili, V.R., Caldiera, G., Dieter R.H.: The goal question metric approach. In: Encyclopedia of Software Engineering (1994)
12. Batory, D., Johnson, C., et al.: Achieving extensibility through product-lines and domain-specific languages: a case study. In: ACM Transactions on Software Engineering and Methodology (2002). https://doi.org/10.1145/505145.505147
13. Boehm, B., et al.: Cost models for future software life cycle processes: COCOMO 2.0. In: Annals of Software Engineering (1995). https://doi.org/10.1007/BF02249046
14. Boot, A., Sutton, A., Papaioannou, D.: Systematic Approaches to a Successful Literature Review. Sage, Thousand Oaks (2016)
15. Burkhardt, J.-M., Detiénne, F., Wiedenbeck, S.: Object-oriented program comprehension: effect of expertise, task and phase. In: Empirical Software Engineering (2002). https://doi.org/10.1023/A:1015297914742
16. Cabot, L., Burgueño, J., Gérard, S.: The future of model transformation languages: an open community discussion. In: Journal of Object Technology (2019). https://doi.org/10.5381/jot.2019.18.3.a7
17. Calvar, T., et al.: Efficient ATL incremental transformations. In: Journal of Object Technology (2019). https://doi.org/10.5381/jot.2019.18.3.a2
18. Cary, J.R., Shasharina, S.G., Cummings, J.C., Reynders, J.V., Hinker, P.J., et al.: Comparison of C++ and Fortran 90 for object-oriented scientific programming. Comput. Phys. Commun. **105**(1), 20–36 (1997)
19. Charmaz, K.: Constructing Grounded Theory. Sage, Thousand Oaks (2014)
20. Ciccozzi, F., Malavolta, I., Selic, B.: Execution of UML models: a systematic review of research and practice. In: Software & Systems Modeling (2019). https://doi.org/10.1007/s10270-018-0675-4
21. Cuadrado, J., Molina, J.G., Tortosa, M.M.: RubyTL: a practical, extensible transformation language. In: Model Driven Architecture–Foundations and Applications. ECMDA-FA 2006 (2006). https://doi.org/10.1007/11787044_13
22. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. In: IBM Systems Journal (2006). https://doi.org/10.1147/sj.453.0621
23. Dieste, O., et al.: Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study. In: Empirical Software Engineering (2017). https://doi.org/10.1007/s10664-016-9471-3
24. Fang, J., Varbanescu, A.L., Sips, H.: A comprehensive performance comparison of CUDA and OpenCL. In: 2011 International Conference on Parallel Processing. ICPP 2011 (2011). https://doi.org/10.1109/ICPP.2011.45
25. Frakes, W.B., Succi, G.: An industrial study of reuse, quality, and productivity. In: Journal of Systems and Software (2001). https://doi.org/10.1016/S0164-1212(00)00121-7
26. Galster, M., et al.: Variability in software systems—a systematic literature review. In: IEEE Transactions on Software Engineering (2014). https://doi.org/10.1109/TSE.2013.56
27. Gherardi, L., Brugali, D., Comotti D.: A Java vs. C++ performance evaluation: a 3D modeling benchmark. In: International Conference on Simulation, Modeling, and Programming for Autonomous Robots. SIMPAR 2012 (2012). https://doi.org/10.1007/978-3-642-34327-8_17
28. Hailpern, B., Tarr, P.: Model-driven development: the good, the bad, and the ugly. In: IBM Systems Journal (2006). https://doi.org/10.1147/sj.453.0451
29. Hebig, R., et al.: Model transformation languages under a magnifying glass: a controlled experiment with Xtend, ATL, and QVT. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2018 (2018). https://doi.org/10.1145/3236024.3236046
30. Henderson, R., Zorn, B.: A comparison of object-oriented programming in four modern languages. In: Software: Practice and Experience (1994). https://doi.org/10.1002/spe.4380241106
31. Hinkel, G.: An approach to maintainable model transformations with an internal DSL. PhD thesis. National Research Center (2013)
32. Hinkel, G., Burger, E.: Change propagation and bidirectionality in internal transformation DSLs. In: Software & Systems Modeling (2019). https://doi.org/10.1007/s10270-017-0617-6
33. Hinkel, G., Goldschmidt, T., et al.: Using internal domain-specific languages to inherit tool support and modularity for model transformations. In: Software & Systems Modeling (2019). https://doi.org/10.1007/s10270-017-0578-9
34. Hutchinson, John, et al.: Empirical assessment of MDE in industry. In: Proceedings of the 33rd International Conference on Software Engineering. ICSE '11 (2011). https://doi.org/10.1145/1985793.1985858
35. ISO/IEC 25010:2011 (2011). ISO/IEC. URL: https://www.iso.org/standard/22749.html
36. Jakumeit, E., et al.: A survey and comparison of transformation tools based on the transformation tool contest. In: Science of Computer Programming (2014). https://doi.org/10.1016/j.scico.2013.10.009
37. Jones, T.C.: Measuring programming quality and productivity. In: IBM Systems Journal (1978). https://doi.org/10.1147/sj.171.0039
38. Jouault, F., et al.: ATL: A QVT-like transformation language. In: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. OOPSLA '06 (2006). https://doi.org/10.1145/1176617.1176691
39. Kahani, N., et al.:. Survey and classification of model transformation tools. In: Software & Systems Modeling (2019). https://doi.org/10.1007/s10270-018-0665-6
40. Kieburtz, R.B., et al.: A software engineering experiment in software component generation. In: Proceedings of the 18th International Conference on Software Engineering. ICSE'96 (1996). https://doi.org/10.1109/ICSE.1996.493448
41. Kitchenham, B., Charters, S.: Guidelines for performing Systematic Literature Reviews in Software Engineering (2007). https://www.researchgate.net/publication/302924724_Guidelines_for_performing_Systematic_Literature_Reviews_in_Software_Engineering
42. Kofod-Petersen, A.: How to do a Structured Literature Review in computer science (2015). https://www.researchgate.net/publication/265158913_How_to_do_a_Structured_Literature_Review_in_computer_science
43. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon transformation language. In: Theory and Practice of Model Transformations, ICMT 2008 (2008). https://doi.org/10.1007/978-3-540-69927-9_4
44. Kosar, T., Bohra, S., Mernik, M.: Domain-specific languages: a systematic mapping study. In: Information and Software Technology (2016). https://doi.org/10.1016/j.infsof.2015.11.001
45. Kosar, T., et al.: Comparing general-purpose and domain-specific languages: an empirical study. In: ComSIS–Computer Science an Information Systems Journal (2010). https://doi.org/10.2298/CSIS1002247K
46. Kfikava, F., Collet, P., France, R.B.: SIGMA: Scala internal domain-specific languages for model manipulations. In: Model-Driven Engineering Languages and Systems. MODELS 2014 (2014). https://doi.org/10.1007/978-3-319-11653-2_35
47. Kurniawan, B., Xue, J.: A comparative study of web application design models using the java technologies. In: Asia-Pacific Web Conference. APWeb 2004 (2004). https://doi.org/10.1007/978-3-540-24655-8_77

48. Loniewski, G., Insfran, E., Abrahão, S.: A systematic review of the use of requirements engineering techniques in model-driven development. In: Model Driven Engineering Languages and Systems (2010). https://doi.org/10.1007/978-3-642-16129-2_16
49. Mens, T., Van Gorp, P.: A taxonomy of model transformation. In: Electronic Notes in Theoretical Computer Science (GraMoT 2005) (2006). https://doi.org/10.1016/j.entcs.2005.10.021
50. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. In: ACM Computing Surveys (CSUR) (2005). https://doi.org/10.1145/1118890.1118892
51. Mohagheghi, P., et al.: An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. In: Empirical Software Engineering (2013). https://doi.org/10.1007/s10664-012-9196-x
52. OMG (2001). Model Driven Architecture (MDA), ormsc/2001-07-01
53. Prechelt, L.: An empirical comparison of seven programming languages. In: Computer (2000). https://doi.org/10.1109/2.876288
54. Rein, P., Taeumel, M., Hirschfeld, R.: Towards empirical evidence on the comprehensibility of natural language versus programming language. In: Design Thinking Research (2019). https://doi.org/10.1007/978-3-030-28960-7_7
55. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. In: IEEE Software (2003). https://doi.org/10.1109/MS.2003.1231150
56. Shaw, M.: Writing good software engineering research papers. In: 25th International Conference on Software Engineering, 2003. Proceedings (2003). https://doi.org/10.1109/ICSE.2003.1201262
57. Shevtsov, S., et al.: Control-theoretical software adaptation: a systematic literature review. In: IEEE Transactions on Software Engineering (2018). https://doi.org/10.1109/TSE.2017.2704579
58. Sjoberg, D.I.K., et al.: Conducting realistic experiments in software engineering. In: Proceedings International Symposium on Empirical Software Engineering. ISESE '02 (2002). https://doi.org/10.1109/ISESE.2002.1166921
59. Somasundaram, R., Karlsbjerg, J.: Research philosophies in the IOS adoption field. In: ECIS 2003 Proceedings, pp. 53 (2003)
60. Tratt, L.: Model transformations and tool integration. In: Software & Systems Modeling (2005). https://doi.org/10.1007/s10270-004-0070-1
61. Van Deursen, A., Klint, P.: Domain-specific language design requires feature descriptions. In: Journal of Computing and Information Technology (2002). https://doi.org/10.2498/cit.2002.01.01
62. Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. In: ACM Sigplan Notices (2000). https://doi.org/10.1145/352029.352035
63. Varro, D., et al.: Termination analysis of model transformations by Petri Nets. In: Graph Transformations. ICGT 2006 (2006). https://doi.org/10.1007/11841883_19
64. Weyns, D., et al.: Claims and supporting evidence for self-adaptive systems: a literature study. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '12 (2012). https://doi.org/10.1109/SEAMS.2012.6224395
65. Whittle, J., et al.: Industrial adoption of model-driven engineering: are the tools really the problem?. In: Model-Driven Engineering Languages and Systems. MODELS 2013 (2013). https://doi.org/10.1007/978-3-642-41533-3_1
66. Wiger, U., Telecom Ab, E.: Four-fold Increase in Productivity and Quality -Industrial-Strength Functional Programming in Telecom-Class Products (2001)
67. Wohlin, C.: Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. EASE '14. Association for Computing Machinery (2014). https://doi.org/10.1145/2601248.2601268

Stefan Götz is a PhD student at the Ulm University. His research is focused on topics surrounding the development and evaluation of model transformation languages. Prior to his work as a PhD student, he was a student of software engineering at the Ulm University where he received his M.Sc. in.

Proof. Dr Mathias Tichy is full professor for software engineering at the Ulm University and director of Institute of Software Engineering and Programming Languages. His main research focus is on model-driven software engineering, particularly for cyber-physical systems. He works on requirements engineering, dependability, and validation and verification complemented by empirical research techniques. He is a regular member of programme committees for conferences and workshops in the area of software engineering and model-driven development. He is a co-author of over 110 peer-reviewed publications.

Raffaela Groner is a PhD student at the Ulm University. Her research is focused on the performance of model transformations. Prior, she studied computer science at the Ulm University.

F.2 Paper B

Advantages and disadvantages of (dedicated) model transformation languages: A Qualitative Interview Study

S. Höppner, Y. Haas, M. Tichy, K. Juhnke

Empirical Software Engineering (EMSE), volume 27, article number 159, 2022
Springer Nature

DOI: 10.1007/s10664-022-10194-7

Empirical Software Engineering (2022) 27:159
https://doi.org/10.1007/s10664-022-10194-7

Advantages and disadvantages of (dedicated) model transformation languages

A qualitative interview study

Stefan Höppner[1] · **Yves Haas**[1] · **Matthias Tichy**[1] · **Katharina Juhnke**[1]

Accepted: 15 June 2022

Abstract

Context Model driven development envisages the use of model transformations to evolve models. Model transformation languages, developed for this task, are touted with many benefits over general purpose programming languages. However, a large number of these claims have not yet been substantiated. They are also made without the context necessary to be able to critically assess their merit or built meaningful empirical studies around them.

Objective The objective of our work is to elicit the reasoning, influences and background knowledge that lead people to assume benefits or drawbacks of model transformation languages.

Method We conducted a large-scale interview study involving 56 participants from research and industry. Interviewees were presented with claims about model transformation languages and were asked to provide reasons for their assessment thereof. We qualitatively analysed the responses to find factors that influence the properties of model transformation languages as well as explanations as to how exactly they do so.

Results Our interviews show, that general purpose expressiveness of GPLs, domain specific capabilities of MTLs as well as tooling all have strong influences on how people view properties of model transformation languages. Moreover, the *Choice of MTL*, the *Use Case* for which a transformation should be developed as well as the *Skill*s of involved stakeholders have a moderating effect on the influences, by changing the context to consider.

Conclusion There is a broad body of experience, that suggests positive and negative influences for properties of MTLs. Our data suggests, that much needs to be done in order to convey the viability of model transformation languages. Efforts to provide more empirical substance need to be undergone and lacklustre language capabilities and tooling need to be improved upon. We suggest several approaches for this that can be based on the results of the presented study.

Communicated by: Alexander Serebrenik

✉ Stefan Höppner
stefan.hoeppner@uni-ulm.de

Extended author information available on the last page of the article.

Keywords Interview · Interview Study · Model Transformation Language · DSL · Model Transformation · MDSE · advantages · disadvantages · Qualitative Analysis

1 Introduction

Model transformations are at the heart of model-driven engineering (MDE) (Sendall and Kozaczynski 2003; Metzger 2005). They provide a way to consistently and automatically derive a multitude of artefacts such as source code, simulation inputs or different views from system models (Schmidt 2006). Model transformations also allow to analyse system aspects on the basis of models (Schmidt 2006) and can provide interoperability between different modelling languages, e.g. architecture description languages like those described by Malavolta et al. (2010). Since the emergence of the MDE paradigm at the beginning of the century numerous dedicated model transformation languages (MTLs) have been developed to support engineers in their endeavours (Jouault et al. 2006; Arendt et al. 2010; OMG 2016b). Their appeal is driven by the promise of many advantages such as increased productivity, comprehensibility and domain specificity associated with using domain specific languages (Hermans et al. 2009; Johannes et al. 2009).

A recent literature study of us revealed, that, while a large number of such advantages and also disadvantages are claimed in literature, there exist only a few studies investigating to what extend these claims actually hold (Götz et al. 2021). The study presents 15 properties of MTLs for which literature claims advantages or disadvantages. In this context, a claimed positive effect on one of the properties means an advantage whereas a negative influence means a disadvantage. The properties identified in the study are: *Analysability*, *Comprehensibility*, *Conciseness*, *Debugging*, *Ease of Writing* (a transformation), *Expressiveness*, *Extendability*, (being) *Just Better*, *Learnability*, *Performance*, *Productivity*, *Reuse & Maintainability*, *Tool Support*, *Semantics & Verification* and *Versatility*.

Our study also revealed, that most claims in literature are made broadly and without much explanation as to where the claim originates from (Götz et al. 2021). Claims such as *"Model transformation languages make it easy to work with models."* (Liepiņš 2012), *"Declarative MTLs increase programmer productivity"* (Lawley and Raymond 2007) or *"Model transformation languages are more concise"* (Hinkel and Goldschmidt 2019) illustrate this. We assume that authors make such claims while having certain context and background in mind, but choose to omit it for unspecified reasons. Some likely reason for omission of the context are, that authors believe it to not be worth mentioning or to preserve space which is often sparse in publications.

Regardless of the concrete reasons, a result of this practice is a lack of cause and effect relations in the context of model transformation languages that explain both why and when certain advantages or disadvantages hold. Claims are thus easily dismissed based on anecdotal evidence. Furthermore, setting up proper evaluation is also difficult because the claims do not provide the necessary background to do so.

To close this gap, we executed a large-scale empirical study using semi-structured interviews. It involved a total of 56 researchers and practitioners in the field of model transformations. The **goal** of our study was to compile a comprehensive list of influences on properties of model transformation languages guided by the following research questions:

RQ 1: What are the factors that influence properties of model transformation languages?

RQ 2: How do the identified factors influence MTL properties?

To concentrate our efforts and best utilize all available resources, we decided to focus on 6 of the 15 properties of model transformation languages identified by us in the preceding SLR (Götz et al. 2021). The 6 properties investigated in this study are: *Comprehensibility*, *Ease of Writing*, *(practical) Expressiveness*, *Productivity*, *Reuse and Maintainability* and *Tool support*. We have chosen these six because they all play a major role in providing reasons for the adoption of model transformation languages.

Interviewees were presented with a number of claims about MTLs from literature and asked to reveal their views on the matter, as well as assumptions and reasons that lead them to agree or disagree with the presented claims. We qualitatively analysed the interviews to understand the participants perceived influence factors and reasons for the advantages or disadvantages stated in the claims. The extracted data was then analysed to find commonalities between interviewees. This was done for single claims as well as for overarching factors and reasons that influence a variety of aspects of MTLs.

We present a comprehensive explanation of factors that, according to experts, play an essential role in the discussion of advantages and disadvantages of model transformation languages for the investigated properties. This is accompanied by a detailed exposition of **how** factors are relevant for the properties given above. Lastly, we discuss the most salient factors and argue actionable results for the community and further research.

As the first study of this type, we make the following contributions:

1. A comprehensive categorisation and listing of factors resulting in advantages or disadvantages of MTLs in the 6 properties studied.
2. A detailed description of why and how each identified factor exerts an influence on different properties.
3. Suggestions for how the presented information can be utilised to empirically investigate MTL properties.
4. Procedural proposals for improving current model transformation languages based on the presented data.

The results of our study show, that there is a large number of factors that influence properties of model transformation languages. There is also a number of factors on which this influence depends on, i.e. factors that have a moderation effect on the influence of other factors. These factors provide a solid basis that allows further studies to be conducted with more focus. They also enable precise decisions on where improvements and adjustments in or for model transformation languages can be made.

The remainder of this paper is structured as follows: Section 2 introduces model-driven engineering and model transformation languages, the context in which our study integrates. Section 3 will detail our methodology for preparing and conducting the interviews and the procedures used to analyse the data accumulated through the interviews. Afterwards Section 4 gives an overview over demographic data of our interview participants while 5 presents our code system and details the findings for each code based on the interviews and analysis thereof. In Section 6 we present overarching findings and in Section 7, we discuss actionable results that can be drawn from our study that indicate avenues to focus on for the research community. Section 8 contains a detailed discussion of the validity threats of this research, and in Section 9 related efforts are presented. Lastly, Section 10 draws a conclusion for our research and proposes future work.

2 Background

This section will provide the necessary background for the context in which our study is integrated in.

2.1 Model-Driven Engineering

The *Model-Driven Architecture* (MDA) paradigm was first introduced by the Object Management Group in 2001 (OMG 2001). It forms the basis for an approach commonly referred to as *Model-driven development* (MDD) (Brown et al. 2005), introduced as means to cope with the ever growing complexity associated with software development. At the core of it lies the notion of using models as the central artefact for development. In essence this means, that models are used both to describe and reason about the problem domain as well as to develop solutions (Brown et al. 2005). An advantage ascribed to this approach that arises from the use of models in this way, is that they can be expressed with concepts closer to the related domain than when using regular programming languages (Selic 2003).

When fully utilized, MDD envisions automatic generation of executable solutions specialized from abstract models (Selic 2003; Schmidt 2006). To be able to achieve this, the structure of models needs to be known. This is achieved through so called meta-models which define the structure of models. The structure of meta-models themselves is then defined through meta-models of their own. For this setup, the OMG developed a modelling standard called *Meta-object Facility* (MOF) (OMG 2016a) on the basis of which a number of modelling frameworks such as the *Eclipse Modelling Framework* (EMF) (Steinberg et al. 2008) and the *.NET Modelling Framework* (Hinkel 2016) have been developed.

2.2 Domain-Specific Languages

Domain-specific languages (DSLs) are languages designed with a notation that is tailored for a specific domain by focusing on relevant features of the domain (Van Deursen and Klint 2002). In doing so DSLs aim to provide domain specific language constructs, that let developers feel like working directly with domain concepts thus increasing speed and ease of development (Sprinkle et al. 2009). Because of these potential advantages, a well defined DSL can provide a promising alternative to using general purpose tools for solving problems in a specific domain. Examples of this include languages such as *shell scripts* in Unix operating systems (Kernighan and Pike 1984), *HTML* (Raggett et al. 1999) for designing web pages or AADL an architecture design language (SAEMobilus 2004).

2.3 Model Transformation Languages

The process of (automatically) transforming one model into another model of the same or different meta-model is called *model transformation* (MT). They are regarded as being at the heart of Model Driven Software Development (Sendall and Kozaczynski 2003; Metzger 2005), thus making the process of developing them an integral part of MDD. Since the introduction of MDE at the beginning of the century, a plethora of domain specific languages for developing model transformations, so called model transformation languages (MTLs), have been developed (Arendt et al. 2010; Balogh and Varró 2006; Jouault et al. 2006; Kolovos et al. 2008; Horn 2013; George et al. 2012; Hinkel and Burger 2019). Model transformation

languages are DSLs designed to support developers in writing model transformations. For this purpose, they provide explicit language constructs for tasks involved in model transformations such as model matching. There are various features, such as directionality or rule organization (Czarnecki and Helsen 2006), by which model transformation languages can be distinguished. For the purpose of this paper, we will only be explaining those features that are relevant to our study and discussion in Sections 2.3.1 to 2.3.7. Table 1 provides an overview over the presented features.

Please refer to Czarnecki and Helsen (2006), Kahani et al. (2019), and Mens and Gorp (2006) for complete classifications.

2.3.1 External and Internal Transformation Languages

Domain specific languages, and MTLs by extension, can be distinguished on whether they are embedded into another language, the so called host language, or whether they are fully independent languages that come with their own compiler or virtual machine.

Languages embedded in a host language are called *internal* languages. Prominent representatives among model transformation languages are *FunnyQT* (Horn 2013) a language embedded in Clojure, *NMF Synchronizations* and the *.NET transformation language* (Hinkel and Burger 2019) embedded in C#, and *RubyTL* (Cuadrado et al. 2006) embedded in Ruby.

Fully independent languages are called *external* languages. Examples of external model transformation languages include one of the most widely known languages such as the *Atlas transformation language* (ATL) (Jouault et al. 2006), the graphical transformation language Henshin (Arendt et al. 2010) as well as a complete model transformation framework called VIATRA (Balogh and Varró 2006).

2.3.2 Transformation Rules

Czarnecki and Helsen (2006) describe rules as being *"understood as a broad term that describes the smallest units of [a] transformation [definition]"*. Examples for transformation rules are the rules that make up transformation modules in ATL, but also functions, methods or procedures that implement a transformation from input elements to output elements.

The fundamental difference between model transformation languages and general-purpose languages that originates in this definition, lies in dedicated constructs that represent rules. The difference between a transformation rule and any other function, method or procedure is not clear cut when looking at GPLs. It can only be made based on the contents thereof. An example of this can be seen in Listing 1, which contains exemplary Java methods. Without detailed inspection of the two methods it is not apparent which method does some form of transformation and which does not.

In a MTL on the other hand transformation rules tend to be dedicated constructs within the language that allow a definition of a *mapping* between input and output (elements). The example rules written in the model transformation language ATL in Listing 2 make this apparent. They define mappings between model elements of type `Member` and model elements of type `Male` as well as between `Member` and `Female` using *rules*, a dedicated language construct for defining transformation mappings. The transformation is a modified version of the well known Families2Persons transformation case (Anjorin et al. 2017).

Table 1 MTL feature overview

Feature	Characteristic	Representative Language
Embeddedness	Internal	FunnyQT (Clojure), RubyTL (Ruby), NMF Synchronizations (C#)
	External	ATL, Henshin, QVT
Rules	Explicit Syntax Construct	ATL, Henshin, QVT
	Repurposed Syntax Construct	NMF Synchronizations (Classes), FunnyQT (Macros)
Location Determination	Automatic Traversal	ATL, QVT
	Pattern Matching	Henshin
Directionality	Unidirectional	ATL, QVT-O
	Bidirectional	QVT-R, NMF Synchronisations
Incrementality	Yes	NMF Synchronizations
	No	QVT-O
Tracing	Automatic	ATL, QVT
	Manual	NMF Synchronizations
Dedicated Model Navigation Syntax	Yes	ATL (OCL), QVT (OCL), Henshin (implicit in rules)
	No	NMF Synchronizations, FunnyQT, RubyTL

```
public void methodExample(Member m) {
  System.out.println(m.getFirstName());
}
public void methodExample2(Member m) {
  Male target = new Male();
  target.setFullName(m.getFirstName() + " Smith");
  REGISTRY.register(target);
}
```

List. 1 Example Java methods

```
rule Member2Male {
  from
    s : Member (not s.isFemale())
  to
    t : Male (
        fullName <- s.firstName + ' Smith'
    )
}

rule Member2Female {
  from
    s : Member (s.isFemale())
  to
    t : Female (
      fullName = s.firstName + ' Smith'
      partner = s.companion
    )
}
```

List. 2 Example ATL rules

2.3.3 Rule Application Control: Location Determination

Location determination describes the strategy that is applied for determining the elements within a model onto which a transformation rule should be applied (Czarnecki and Helsen 2006). Most model transformation languages such as ATL, Henshin, VIATRA or QVT (OMG 2016b), rely on some form of *automatic traversal* strategy to determine where to apply rules.

We differentiate two forms of location determination, based on the kind of matching that takes place during traversal. There is the basic *automatic traversal* in languages such as ATL or QVT, where single elements are matched to which transformation rules are applied. The other form of location determination, used in languages like Henshin, is based on *pattern matching*, meaning a model- or graph-*pattern* is matched to which rules are applied. This does allow developers to define sub-graphs consisting of several model elements and references between them which are then manipulated by a rule.

The *automatic traversal* of ATL applied to the example from Listing 2 will result in the transformation engine automatically executing the `Member2Male` on all model

elements of type `Member` where the function `isFemale()` returns `false` and the `Member2Female` on all other model elements of type `Member`.

The *pattern matching* of Henshin can be demonstrated using Fig. 1, a modified version of the transformation examples by Krause et al. (2014). It describes a transformation that creates a couple connection between two actors that play in two films together. When the transformation is executed the transformation engine will try and find instances of the defined graph pattern and apply the changes on the found matches.

This highlights the main difference between *automatic traversal* and *pattern matching* as the engine will search for a sub graph within the model instead of applying a rule to single elements within the model.

2.3.4 Directionality

The directionality of a model transformation describes whether it can be executed in one direction, called a unidirectional transformation or in multiple directions, called a multidirectional transformation (Czarnecki and Helsen 2006).

For the purpose of our study the distinction between unidirectional and bidirectional transformations is relevant. Some languages allow dedicated support for executing a transformation both ways based on only one transformation definition, while other require users to define transformation rules for both directions. General-purpose languages can not provide bidirectional support and also require both directions to be implemented explicitly.

The ATL transformation from Listing 2 defines a unidirectional transformation. Input and output are defined and the transformation can only be executed in that direction.

The QVT-R relation defined in Listing 3 is an example of a bidirectional transformation definition (For simplicity reasons the transformation omits the condition that males are only created from members that are not female). Instead of a declaration of input and output, it defines how two elements from different domains relate to one another. As a result given a `Member` element its corresponding `Male` elements can be inferred, and vice versa.

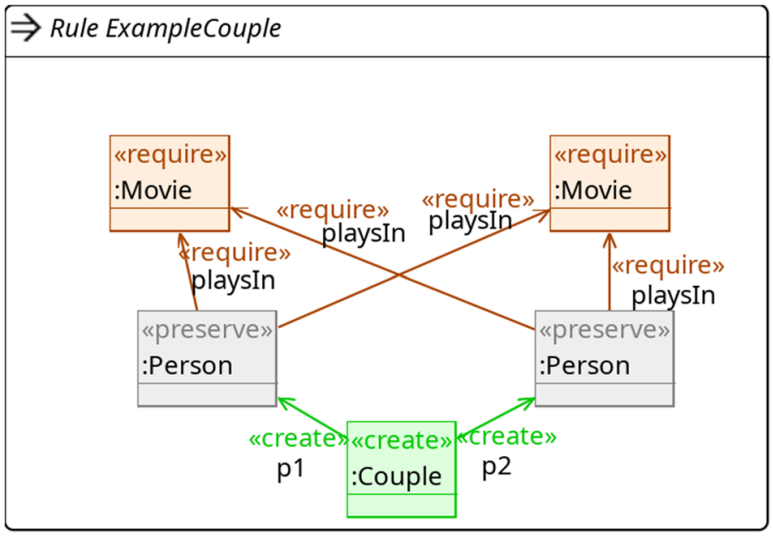


Fig. 1 Example Henshin transformation

```
top relation Member2Male {
  n, fullName : String;
  domain Families s:Member {
    firstName = n };
  domain Persons t:Male {
    fullName = fullName};
  where {
    fullName = n + ' Smith'; };
}
```

List. 3 Example QVT-R relation

2.3.5 Incrementality

Incrementality of a transformation describes whether existing models can be updated based on changes in the source models without rerunning the complete transformation (Czarnecki and Helsen 2006). This feature is sometimes also called model synchronisation.

Providing incrementality for transformations requires active monitoring of input and/or output models as well as information which rules affect what parts of the models. When a change is detected the corresponding rules can then be executed. It can also require additional management tasks to be executed to keep models valid and consistent.

2.3.6 Tracing

According to Czarnecki and Helsen (2006) tracing *"is concerned with the mechanisms for recording different aspects of transformation execution, such as creating and maintaining trace links between source and target model elements"*.

Several model transformation languages, such as ATL and QVT have automated mechanisms for trace management. This means that traces are automatically created during runtime. Some of the trace information can be accessed through special syntax constructs while some of it is automatically resolved to provide seamless access to the target elements based on their sources.

An example of tracing in action can be seen in `line 16` of Listing 2. Here the `partner` attribute of a `Female` element that is being created, is assigned to `s.companion`. The `s.companion` reference points towards a element of type `Member` within the input model. When creating a `Female` or `Male` element from a `Member` element, the ATL engine will resolve this reference into the corresponding element, that was created from the referred `Member` element via either the `Member2Male` or `Member2Female` rule. ATL achieves this by automatically tracing which target model elements are created from which source model elements.

2.3.7 Dedicated Model Navigation Syntax

Languages or syntax constructs for navigating models is not part of any feature classification for model transformation languages. However, it was often discussed in our interviews and thus requires an explanation as to what interviewees refer to.

Languages such as OCL (OMG 2014), which is used in transformation languages like ATL, provide dedicated syntax for querying and navigating models. As such they provide syntactical constructs that aid users in navigation tasks. Different model transformation

languages provide different syntax for this purpose. The aim is to provide specific syntax so users do not have to manually implement queries using loops or other general purpose constructs. OCL provides a functional approach for accumulating and querying data based on collections while Henshin uses graph patterns for expressing the relationship of sought-after model elements.

3 Methodology

To collect data for our research question, we decided on using semi-structured interviews and a subsequent qualitative content analysis that follows the guidelines detailed by Kuckartz (2014). Semi-structured interviews were chosen as a data collection method because they present a versatile approach to eliciting information from experts. They provide a framework that allows to get insights into opinions, thoughts and knowledge of experts (Meyer and Booker 1990; Hove and Anda 2005; Kallio et al. 2016). The qualitative content analysis guidelines by Kuckartz (2014) were chosen because of their detailed descriptions for all steps of the analysis process. As such they provide a more detailed and modern framework compared to the procedures introduced by Mayring (1994), which have long been a gold standard in qualitative content analysis.

An overview over our complete study design can be found in Fig. 2. It shows the order of activities that were planned and executed as well as the artefacts produced and used throughout the study. Each activity is annotated with the section number in which we detail the activity. We split our approach into three main-phases: *Preparation* (detailed in Section 3.1), *Operation* (detailed in Section 3.2) and *Coding & Analysis* (detailed in Section 3.3).

For the preparation phase, we used a subset of the claimed properties of model transformation languages identified by us (Götz et al. 2021) to develop an interview guide. The guide focuses around asking participants **whether** they agree with a claim from one of the properties and then envisages the usage of **why** questions to gain a deeper understanding of their opinions on the matter. After identifying and contacting participants based on the publications considered during our previous literature review (Götz et al. 2021), we conducted 54 interviews with 55 interviewees (at the request of two participants, one interview was

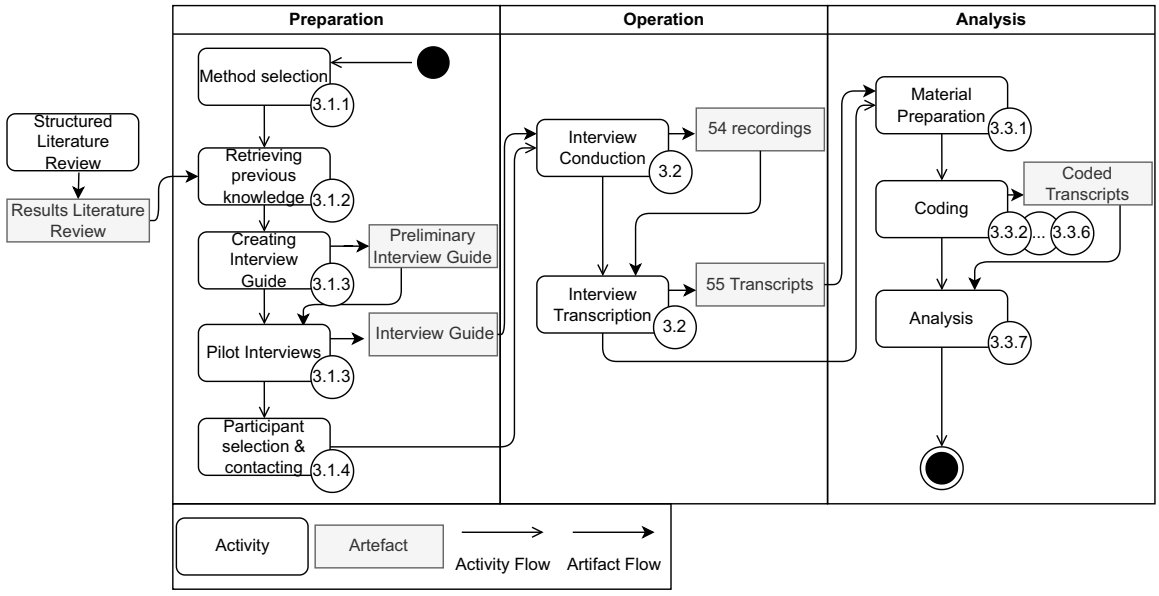


Fig. 2 Overview over the study design

conducted with both of them together) and collected one additional written response. During the *Coding & Analysis* phase, we coded and analysed all 54 transcripts, as well as the written response, guided by the framework detailed by Kuckartz. In doing so, we focused first on factors and reasons for the individual properties and then on common factors and reasons between them.

The remainder of this section will describe in detail how each of the three phases of our study was conducted.

3.1 Interview Preparation

Our interview preparation phase consists of the creation of an interview guide plus selecting and contacting appropriate interview participants. We use the guidelines by Kallio et al. (2016) for the creation of our interview guide and expand the steps detailed there with steps for selecting and contacting participants. In addition, we use the guidance from Newcomer et al. (2015) to construct our study in the best possible way.

According to Kallio et al. (2016) the creation of an interview guide consists of five consecutive steps. First, researchers are urged to evaluate how appropriate semi-structured interviews are as a data collection method for the posed research questions. Then, existing knowledge about the topic should be retrieved by means of a literature review. Based on the knowledge from the knowledge retrieval phase, a preliminary interview guide can then be formulated and in another step be pilot tested. Lastly the complete interview guide can then be presented and used. As previously stated, we enhance these steps with two additional steps for selecting and contacting potential interview participants.

In the following, we detail how the presented steps were executed and the results thereof.

3.1.1 Identifying the Appropriateness of Semi-Structured Interviews

The goal of our study, outlined in our research questions, is to collect and analyse reasons and background information of why people believe claims about model transformation languages to be true. Data such as this is qualitative by nature and hence requires a research method capable of producing qualitative data. According to Hove and Anda (2005) and Meyer and Booker (1990) expert interviews are one of the most widely used research methodologies in the technical field for this purpose. They allow to ascertain qualitative data such as opinions and estimates. Interviews also enable qualitative interpretation of already available data (Meyer and Booker 1990) which perfectly aligns with our goal. Moreover the opportunity to ask further questions about specific statements made by the participants (Newcomer et al. 2015) fits the open ended nature of our research question. For these reasons, we believe semi-structured interviews to be a well suited to answer our research questions.

3.1.2 Retrieving Previous Knowledge

In our previous publication (Götz et al. 2021), we detailed the preparation, execution and results of an extensive structured literature review on the topic of claims about model transformation languages. The literature review resulted in a categorization of 127 claims into 15 different categories (i.e. properties of MTLs) namely *Analysability*, *Comprehensibility*, *Conciseness*, *Debugging*, *Ease of Writing a transformation*, *Expressiveness*, *Extendability*, *Just better*, *Learnability*, *Performance*, *Productivity*, *Reuse & Maintainability*, *Tool Support*, *Semantics and Verification* and lastly *Versatility*. These properties and the claims about them serve as the basis for the design of our interview study presented here.

3.1.3 Interview Guide

The interview guide involves presenting each interview participant with several claims on model transformation languages. We use claims from literature instead of formulating our own statements, to make them more accessible. This also prevents any bias from the authors to be introduced at this step. Participants are first asked to assess their agreement with a claim before transitioning into a discussion on what the reasons for their decision are based on an open-ended question. This style of using close-ended questions as a prerequisite for open-ended or probe questions has been suggested by multiple guides (Newcomer et al. 2015; Hove and Anda 2005).

We focus on a subset of six properties. This is due to the aim of keeping the length of interviews within an acceptable range for participants. According to Newcomer et al. (2015) semi-structured interviews should not exceed a maximum length of one hour. As a result, only a number of properties can be discussed per interview. In order to still talk with enough participants about each property, the number of properties examined must be reduced. The properties we discuss in the interviews and the reasons why they are relevant are as follows:

- *Comprehensibility*: Is an important property when transformations are being developed as part of a team effort or evolve over time.
- *Ease of Writing*: Is a decisive property that influences whether developers want to use a languages to write transformations in.
- *Expressiveness*: Is one of the most cited properties in literature (Götz et al. 2021) and main selling point of domain specific languages in general.
- *Productivity*: Is a property that is highly relevant for industrial adoption.
- *Reuse & Maintainability*: Is another property that enables wider adoption of model transformation languages in project settings.
- *Tools*: High-quality tools can provide huge improvements to the development.

The list consists of the 5 most claimed properties form the previous literature review (Götz et al. 2021) and is supplemented with *Productivity*, because we believe this attribute to be the most relevant for industry adoption.

To maximize the response rate of contacted persons, we aim for an interview length of 30 minutes. This decision is based on experiences from previous interview studies conducted at our research group (Groner et al. 2020; Juhnke et al. 2020) and fits within the maximum interview length suggested by Newcomer et al. (2015).

To best utilize the limited time per interview, the six properties are split into three sets of two properties each. In each interview one of the three sets is discussed.

For each property, one non-specific, one specific and one negative claim is used to structure all interviews involving this property around. A complete overview over all selected claims can be found in Table 2.

We consider non-specific claims to be those that do not provide any rationale as to why the claimed property holds, e.g. *"Model transformation languages ease the writing of model transformations."*. The non-specific claims chosen simply reflect the property itself. They serve the purpose of getting participants to state their assumptions and beliefs for the property without any influence exerted by the discussed claim.

We consider those claims as specific, that provide a rationale or reason for why the claimed property holds, e.g. *"Model transformation languages, being DSLs, improve the productivity."*. And we consider negative claims to be those, that state a negative property of model transformation languages, e.g. *"Model transformation languages lack sophisticated*

Table 2 Properties and Claims

Property	Claim
Comprehensibility	The use of model transformation languages increases the comprehensibility of model transformations.
	Model transformation languages incorporate high-level abstractions that make them more understandable than general purpose languages.
	Most model transformation languages lack convenient facilities for understanding the transformation logic.
Ease of Writing	The use of model transformation languages increases the ease of writing model transformations.
	Model transformation languages ease development efforts by offering succinct syntax to query from and map model elements between different modelling domains.
	Model transformation languages require specific skills to be able to write model transformations.
Expressiveness	The use of model transformation languages increases the expressiveness of model transformations.
	Model transformation languages hide transformation complexity and burden from the user.
	Having written several transformations we have identified that current model transformation languages are too low a level of abstraction for succinctly expressing transformations between DSLs because they demonstrate several recurring patterns that have to be reimplemented each time.
Productivity	The use of model transformation languages increases the productivity of writing model transformations.
	Model transformation languages, being DSLs, improve the productivity.
	Productivity of GPL development might be higher since expert users for general purpose languages are easier to hire.
Reuse & Maintainability	The use of model transformation languages increases the reusability and maintainability of model transformations.
	Bidirectional model transformations have an advantage in maintainability.
	Model transformation languages lack sophisticated reuse mechanisms.
Tool Support	There is sufficient tool support for the use of model transformation languages for writing model transformations.
	Tool support for external transformation languages is potentially more powerful than for internal MTL or GPL because it can be tailored to the DSL.
	Model transformation languages lack tool support.

reuse mechanisms.”. Generally, we use claims where we believe the discussions about the reasons to provide useful insights.

There exist several reasons why we believe this setup of using the same three none-specific, specific and negative claims for each property to be appropriate. First, the non-specific claim allows participants to provide any and all factors or reasons that they believe influence a claimed property. The specific claim then allows us to introduce a reason, that participants might not have thought about. It also prompts a discussion about a particular reason or factor that is shared between all participants. This ensures at least one area for cross comparison between answers. The negative claim forces participants to also

deliberate negative aspects, providing a counterbalance that counteracts bias. Furthermore, the non-specific claim provides an easy introduction into the discussion about a specific MTL property that can present the interviewer with an overview of the participants thoughts on the matter. It also allows participants to provide other influence factors not specifically covered through the discussed claims or even new factors and reasons not present in the collection of claims from our literature review (Götz et al. 2021).

The complete interview guide resulting from the aforementioned considerations can be seen in Fig. 3. After introductory pleasantries we start all interviews of with demographic questions. Although some sources discourage asking demographic questions early in the interview due to their sensitive nature (Newcomer et al. 2015), we use them to break the ice between the interviewer and interviewee because our demographic questions do not probe any sensitive information.

After this initial get-to-know each other phase, the interviewer then proceeds to explain the research intentions, goals and the procedure of the remaining interview. Depending on the property-set selected for the interview, participants are then presented with a claim about a property. They are asked to rate their agreement with the claim based on a 5-point likert scale (5: completely agree, 4: agree, 3: neither agree nor disagree, 2: disagree, 1: completely disagree). The likert scale is used to allow the interviewer to better assess the participants tendency compared to a simple yes or no question. This part of the interview is intended solely to get a first impression of the view of the participant and not for a quantitative analysis. It also creates a casual point of entry for the interviewee to think about the topic under consideration. We communicate this to all participants to reduce any pressure they might feel to answer the question correctly. Afterwards an open-ended question inquiring about the reasons for the interviewees assessment is asked.

Some terms used within the discussed claims have ambiguous definitions. We tried to ask participants to explain their understanding of such terms, to prevent errors in analysis due to interviewees having different interpretations thereof. This allows for better assessment during analysis. The terms we have deemed to be ambiguous are: *'succinct syntax'*,

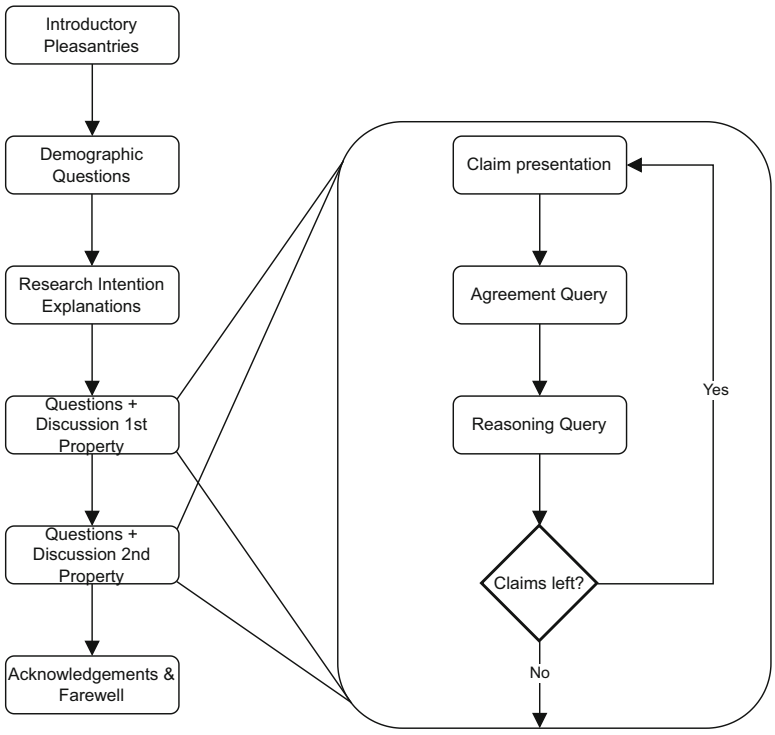


Fig. 3 Interview guide

'mapping', *'specific skills'*, *'high-level abstractions'*, *'convenient facilities'*, *'sufficient tool support'*, *'powerful tool support'*, *'sophisticated reuse mechanisms'* and *'expressiveness'*. We provide a definition for the term *expressiveness*. This is, because we are only interested in a specific type of *expressiveness*, i.e. how concisely and readily developers can express something. We are not interested in expressiveness in a theoretical sense, i.e. the closeness to Turing completeness.

This process of presenting a claim, querying the participants agreement before discussing their reasons for the assessment is repeated for all 3 claims about both properties. After discussing all claims, it is explained to the participants that the formal part of the interview is finished and that they are allowed to make final remarks about all discussed topics or other properties they want to address. After this phase of the interview acknowledgements on the part of the interviewer are expressed before saying goodbye. The complete question catalogue for the interviews can be found in Appendix A.

The interview guide was tested in a pilot study by the main author with one co-author that was not involved in its creation. After pilot testing, we changed the question about agreement with a claim from a yes-no question to one that uses a likert scale. We also extended the question sets with non-specific claims that do not contain any reasoning. Before adding the non specific claim, discussions focused too much on the narrow view within the presented claims.

3.1.4 Selecting & Contacting Participants

The target population for our study consists of all users of model transformation languages. To select potential participants for our study we rely on data from our previous literature review (Götz et al. 2021). The literature review produced a list of publications that address the topic of model transformations and model transformation languages. Because search terms such as 'model to text' and the like were not used in the study, using this list limits our results to model to model transformation languages. We discuss this limitation more thoroughly in Section 8.2.

All authors of the resulting publications are deemed to be potential interview participants. We assume, that people using MTLs in industry do have some research background and thus have published work in the field. There is also no other systematic way to find industry users. We also assume that people who are still active in the field have published within the last 5 years. This limits outreach but makes the set of potential participants more manageable. For this reason, the list was shortened to publications more recent than 2015 before the authors of all publications was compiled. This resulted in a total of 645 potential participants.

After selection, the authors were contacted via mail. First, everyone was contacted once and then, after a week, everyone who had not responded by then was contacted again. The texts we use for both mails can be found in Appendix B. Ten potential participants, from the list of potential participants, were not contacted through this channel but via personalised emails, as they are personal contacts of the authors.

Within the contact mails, potential participants are asked to select a suitable date for the interview and fill out a data consent form allowing us to record and transcribe the interviews.

Overall of the 645 contacted authors, 55 agreed to participate in our interview study resulting in a response rate of 8.53%[1].

[1] when including the written response in this statistic, the resulting response rate is 8.68%.

3.2 Interview Conduction and Transcription

All but one interview were conducted by the first author using the online conferencing tool WebEx and lasted between 20 and 80 minutes. Due to scheduling issues, one interview had to be conducted by the second author, who had a preparatory mock interview with the main interviewer. Additionally, at the request of two participants, one interview was conducted with both of them together. Since our main focus for all interviews is on discussions, we do not believe this to have any effect on its results. WebEx is the chosen conferencing tool, due to its availability to the authors and its integrated recording tool which is used to record all interviews. For data privacy reasons and for easier in-depth analysis later on, all recordings are transcribed by two authors. To increase the readability of heavily fragmented sentences they are shortened to only contain the actual content without interruptions. In case of audibility issues the transcribing authors consulted with each other to try and resolve the issue. Altogether the interviews produced just over 32 hours of audio and about 162.100 words of transcriptions.

Each day, the main author decided on which question sets to use for all participants that had agreed to partake in the interviews. The question sets had to be chosen daily, as many participants only responded to the invitation after interviews had already taken place.

The goal of the decision process was, to ensure an even spread of participants over the question sets based on relevant demographic backgrounds, namely *research*, *industry*, *MTL developer* and *MTL user*. We consider those relevant because each group has a different view point on model transformation languages and their usage for writing transformations. It is therefore important to have answers from each group for each set of questions, to reduce the risk of missing relevant opinions.

We were able to ensure that at least one representative for each demographic group provided answers for each question set. A complete uniform distribution was not possible due to overlaps in the demographic groups.

3.3 Coding & Analysis

Coding and analysing the interview transcripts is done in accordance with the guidelines for *content structuring content analysis* suggested by Kuckartz (2014). The guideline recommends a seven step process (depicted in Fig. 4) for coding and analysing qualitative data. All steps are carried out with tool support provided by the MAXQDA[2] software. In the following, we explain how each process step is conducted in detail. We will use the following statement as a running example to show how codes and sub-codes are assigned and how the coding of text segments evolved throughout the process: *"Of course some MTLs use explicit traceability for instance. But even then you have a mechanism to access it. And if you have a MTL with implicit traceability where the trace links are created automatically then of course you gain a lot of expressivity because you don't have to write something that you would otherwise have to write for almost every rule."* (**P30**)

3.3.1 Initial Text Work

The initial text work step initiates our qualitative analysis. Kuckartz (2014) suggests to read through all the material and highlight important segments as well as to write notes for the

[2] https://www.maxqda.com/

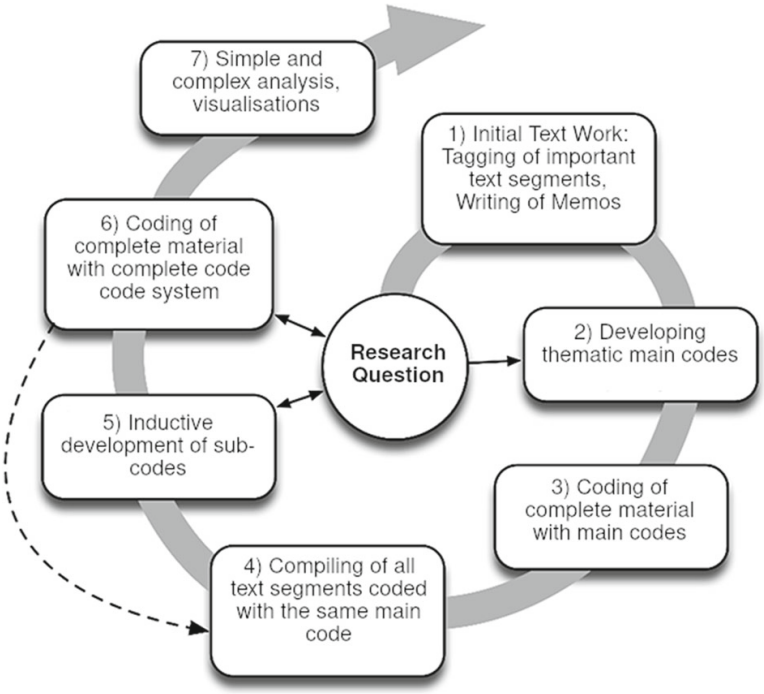


Fig. 4 Process of a content structuring content analysis as presented by Kuckartz (2014)

transcripts using memos. Following these suggestions, we apply *initial* coding from constructivist grounded theory (Charmaz 2014; Vollstedt and Rezat 2019; Stol et al. 2016) to mark and summarize all text segments where interviewees reason about their beliefs on influence factors about the discussed properties. To do so, the two authors, which conducted and transcribed the interviews, read through all transcripts and mark all relevant text segments with codes that preferably represented the segment word for word. The codes allow for easier reference in later steps and, due to tooling, we are still able to quickly read the underlying text segment if necessary.

During this step, the example statement was labelled with the code `automatic tracing increases expressiveness because no manual overhead`.

3.3.2 Developing Thematic Main Codes

For developing the thematic main codes for our study we follow the common practice of inferring them from our research questions as suggested by Kuckartz (2014). Since the goal of our research is to investigate implicit assumptions, and factors that influence the assessment of experts about properties of model transformation languages three main codes arise:

- **Properties**: Denoting which property is being discussed (e.g. *Comprehensibility*).
- **Factors**: Denoting what influences a discussed property according to an interviewee (e.g. *Bidirectional functionality of a MTL*).
- **Factor assessment**: Denoting an evaluation of how a factor influences a property (e.g. *positive* or *negative* or *mixed depending on other factors*).

The sub-codes for the property code can be directly defined based on the six properties from our previous literature review (Götz et al. 2021). As such they are deductive

(a-priori) codes that are intended to mark text segments based on the properties that are being discussed in them.

3.3.3 Coding of All the Material with Main Codes

In order to code of all the material with the main codes one author analyses all interview transcripts. While doing so, the conversations about a discussed claim are marked with the code that is based on the property stated in the claim. To exemplify this, all discussions on the claim *"The use of MTLs increases the comprehensibility of model transformations."* are coded with the main code *comprehensibility*.

This realisation of the process step breaks with Kuckartz's specifications in multiple ways. First, we do not code the material with the main codes **Factors** and **Factor assessment**, because all factors and factor assessments are already coded with the summarising *initial* codes. These will be refined into actual sub-codes of **Factors** and **Factor assessment** in a later step. Second, we directly code segments with the sub-codes for the **Property** main code, because the differentiation comes naturally with the structure of the interviews and delaying this refinement makes no sense. And third, this way of coding makes it possible that unimportant segments are also coded, something that Kuckartz suggests not to do. However, we actively decided in favour of this, because it accelerates the coding process enormously. Furthermore, only overlaps of the property codes with the other codes are considered, in later steps, thus automatically excluding unimportant text segments from consideration.

During this step, the coding for the example text segment was extended with the code `Expressiveness`. While this does not look like much of an enhancement on the surface, it is paramount to allow for systematic analysis in later steps.

After this step the example segment had its *initial code*, summarising the essence of the statement, and the explicit *property sub-code* `Expressiveness`, providing the first systematic categorisation of the segment.

3.3.4 Compilation of All Text Passages Coded with the Same Main Code

This step forms the basis for the subsequent iterative process of inductively developing sub-codes for each main code. Due to the use of the MAXQDA tool, this step is purely technical and does not require any special procedure outside of the selection of the main code that is being considered in the tool.

3.3.5 Inductive Development of Sub-Codes

The inductive development of sub-codes forms the most important coding step in our study. Inductive development here means that the sub-codes are developed based on the transcripts contents.

Kuckartz suggests to read through all segments coded with a main code to iteratively refine the code into several sub-codes that define the main category more precisely (Kuckartz 2014). We optimize this step by analysing all the *initial* codes from the *Initial Text Work* step, to construct concise and comprehensive codes for similar *initial* codes that could be used as sub-codes for the **Factor** or **Factor assessment** main codes. In doing so we follow the *focused* coding procedure of constructivist grounded theory to refine the initial code system.

All sub-codes of the **Factor** main code, that are refined using this process, are thematic codes, meaning they denote a specific topic or argument made within the transcripts. As a result, the sub-codes represent factors explicitly named by interviewees that influence the

different properties. In contrast, all sub-codes of the **Factor assessment** main code, that are refined using this process, are evaluative codes, meaning they represent an evaluation, made by the authors, about an effect. More specifically, the codes represent an evaluation of how participants believe factors influence various properties.

Because of the importance of this coding step, the sub code refinement is created in a joint effort by three of the authors. First, over a period of three meetings, the authors develop comprehensive codes based on the *initial* codes of 18 interviews through discussions. Then the main author complements the resulting code system by analysing the remaining set of interview transcripts, while the two other authors each analyse half of them. In a final meeting any new sub code, devised by one of the authors, is discussed and a consensus for the complete code system is established.

During this step no code segment is extended with additional codes. Instead new codes derived from the *initial codes* are saved for usage in the following steps.

From the example code segment and its *initial code*, a sub-code *automatic tracing* for the **Factors** code was derived. The finalised sub-code `Traceability` was decided upon based on the combination with other derived codes of similar meaning, like *traces*.

3.3.6 Coding of All the Material with Complete Code System

After the final code system is established, the main author processes all transcripts to replace the *initial* codes with codes from the final code system. For this, each coded statement is re-coded with codes indicating the influence factors expressed by the interviewees as well as a factor assessment, if possible. This final coding step is done by the main author while all three co-authors each check 10 coded transcripts to validate the correct and consistent use of all codes and to make sure all relevant statements are considered. The results from the reviews are discussed in pairwise meetings between the main author and the reviewing co-author before being incorporated in a final coding approved by all authors.

During this step, the *initial code* for the example segment was dropped and replaced by the codes `MTL advantage` and `Traceability`.

The final codes assigned to the example text segment thus were: *Expressiveness*, *Traceability* and *MTL advantage*. The reasoning given within the statement as to why automatic tracing provides an expressiveness advantage, are manually extracted during analysis using tooling provided by MAXQDA.

3.3.7 Simple and Complex Analysis and Visualisation

The resulting coding and the coded text segments are then used as the basis for our analysis which, in accordance with our research question, focuses on identifying and evaluating factors that influence the properties of MTLs. As recommended by Kuckartz (2014), this is first done for each **Property** individually before analysis across all properties is conducted (as shown in Fig. 5).

For analysing the influence factors of an individual property, we use the MAXQDA tooling to find segments coded with both a factor and the considered property. Using this approach we first compile a list of all factors relevant for a property, before then doing an in-depth analysis of all the gathered statements for each factor. Here the goal is to elicit commonalities and contradictions between the opinions of our interviewees that can be used to establish a theory on how each factor influences each property individually.

In terms of our example text segment, the segment and all other segments coded with `Expressiveness` and `Traceability` were read and analysed. The goal was to see

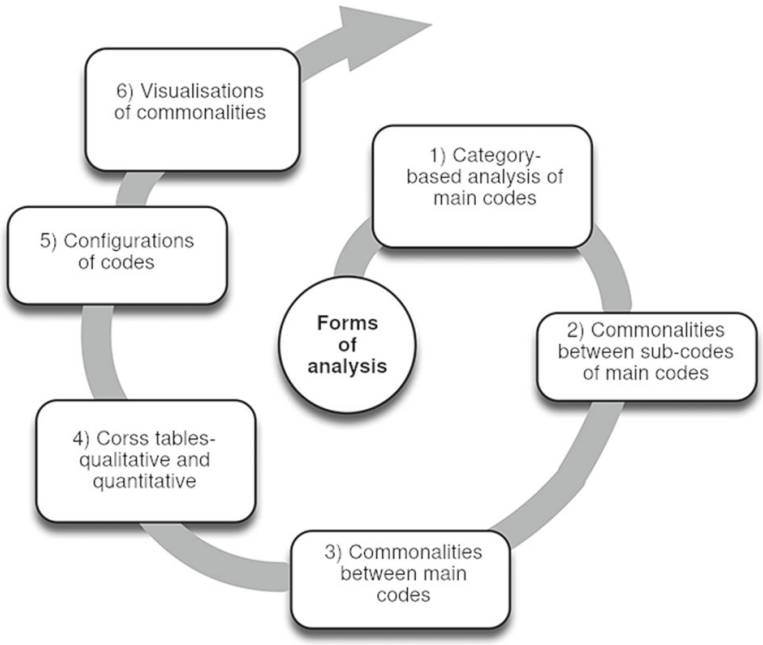


Fig. 5 Analysis forms in a content structuring content analysis as presented by Kuckartz (2014)

if reduced overhead from implicit trace capabilities played a role in the argumentation of other participants and to gather all the other mentioned reasons.

For the analysis over all properties combined we apply the *theoretical* coding process of constructivist grounded theory (Charmaz 2014; Stol et al. 2016) to develop a model of influences. To do so, the **Factor assessment**s are used to examine how the factors influence the respective properties, what the commonalities between properties are and where the differences lie. The goal here is to develop a cohesive theory which explains the influences of factors on the individual properties but also on the properties as a whole and potential influences between the factors themselves.

In terms of our example text segment, the results from analysing `Expressiveness` and `Traceability` segments were compared to results from analysing segments coded with other property codes and `Traceability`. The goal was to find commonalities and differences between the analysed groups.

3.3.8 Privacy and Ethical concerns

All interview participants were informed of the data collection procedure, the handling of the data and their rights surrounding the process, prior to the interview.

During selection of potential participants the following data was collected and processed.

- First & last name.
- E-Mail address.

For participants that agreed to the partake in the interview study the following additional data was collected and processed during the course of the study.

- Non anonymised audio recording of the interview.
- Transcripts of the audio recordings.

All data collected during the study was not shared with any person outside of the group of authors. Audio recordings were handled only by the first and second author.

The complete information and consent form can be found in Appendix D. All participants have consented to having their interview recorded, transcribed and analysed based on this information. All interview recordings were stored on a single device with hardware encryption and deleted as soon as transcriptions were finalised. The interview transcripts were processed to prevent identification of participants. For this, identifying statements and names were removed.

Apart from the voice recordings and names, no sensitive information about the interviewees was collected.

The study design was not presented to an ethical board. The basis for this decision are the rules of the German Research Foundation (DFG) on when to use a ethical board in humanities and social sciences[3]. We refer to these guidelines because there are none specifically for software engineering research and humanities and social sciences are the closest related branch of science for our research.

4 Demographics

We interviewed, and got one written response, from a total of 56 experts from 16 different countries with varied backgrounds and experience levels and collected one comprehensive written response. Table 4 in Appendix C presents an overview of the demographic data about all interview participants. Experts and their statements are distinguished via an anonymous ID (**P1** to **P56**).

4.1 Background

As evident from Fig. 6 participants with a research background constitute the largest portion of our interviewees. Overall there is an even split between participants solely from research and those that have at least some degree of industrial contact (either through industry projects or by working in industry). Only 3 participants stated to have used model transformations solely in an industrial context. This is in part offset by the fact that 25 of interviewees have executed research projects in cooperation with industry or have worked both in research and industry (22 and 3 respectively). While there is a definitive lack of industry practitioners present in our study, a large portion of interviewees are still able to provide insights into model transformations and model transformation languages with an industry view.

Lastly, 10 of our participants are, in some capacity, involved in the development of model transformation languages. They can provide a different angle on advantages or disadvantages of MTLs compared to the 46 participants that use them solely for transformation purposes.

4.2 Experience

50 interviewees expressed to have 5 or more years of experience in using model transformations. Moreover, 24 of the participants have over 10 years of experience in the field. Lastly there was a single participant that had only used model transformations for a brief amount of time during their masters thesis.

[3] https://www.dfg.de/foerderung/faq/geistes_sozialwissenschaften/

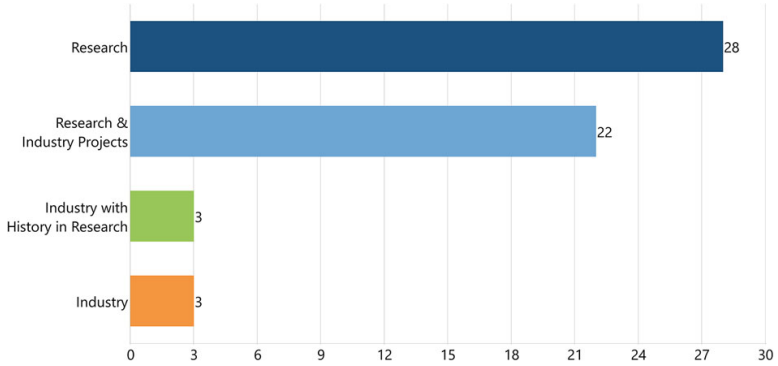


Fig. 6 Distribution of participants background

4.3 Used Languages for Transformation Development

To better assess our participants and to qualify their answers with respect to their background we asked all interviewees to list languages they used to develop model transformations. Figure 7 summarises the answers given by participants while categorizing languages in one of three categories namely *dedicated MTL*, *internal MTL* and *GPL*. This differentiation is based on the classifications from Czarnecki and Helsen (2006) and Kahani et al. (2019).

The distinction between GPL and dedicated/internal MTL is made, to gain an overview over how large the portion of users of general purpose languages for the development is, compared to the users of model transformation languages. Furthermore, it also allows for comprehending the viewpoint participants will take when answering questions throughout the interview, i.e. do they compare general purpose languages with model transformation languages based on their experience with both or do they give specific insights into their

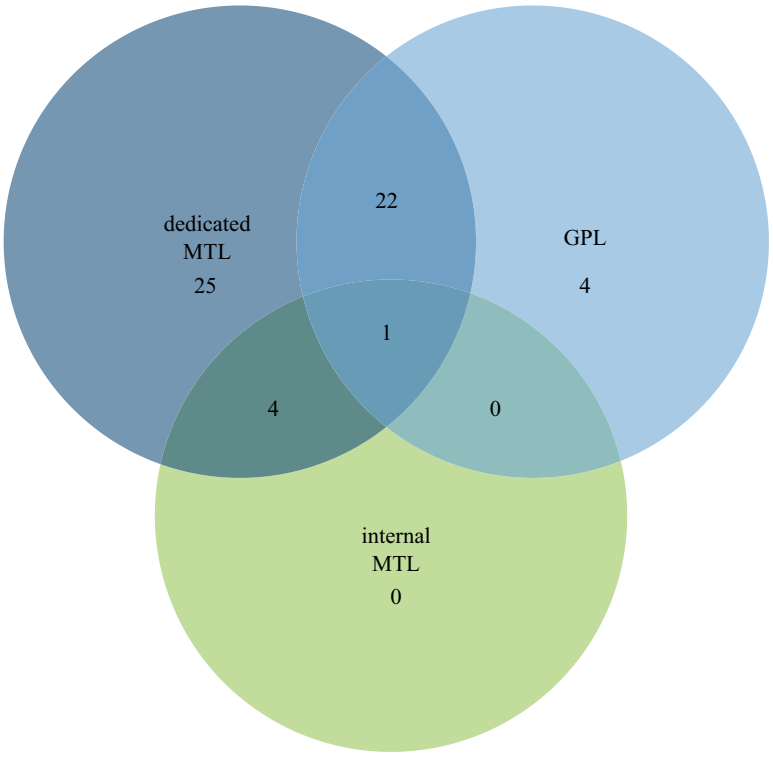


Fig. 7 Venn diagram depicting the language usage of participants

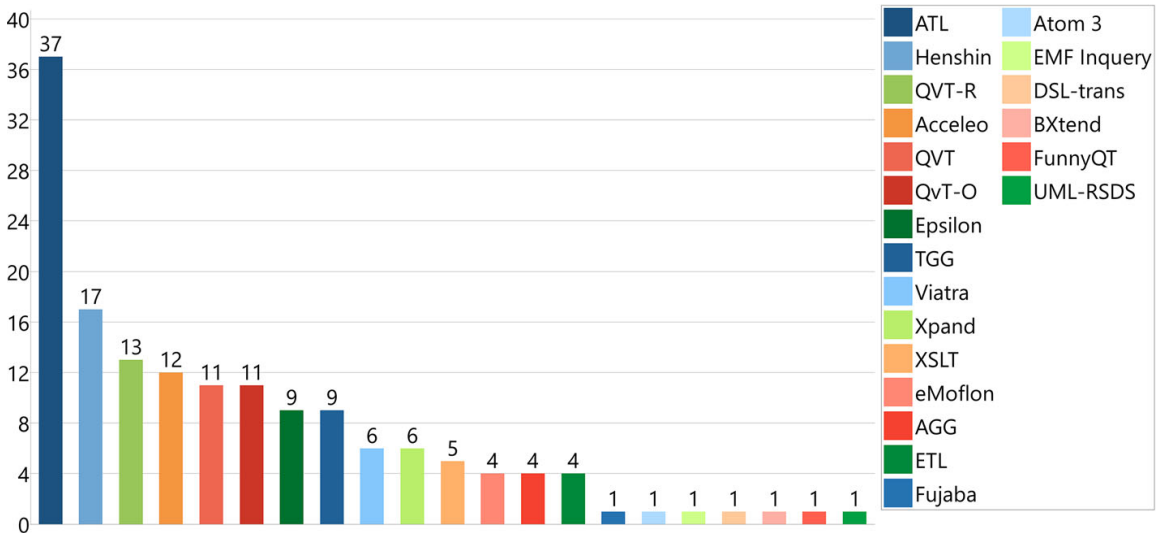


Fig. 8 Number of participants using a specific dedicated MTL

experiences with one of the two approaches. Internal MTL is separated from dedicated MTL because one claim within the interview protocol specifically explores the topic of internal model transformation languages.

52 participants have used dedicated model transformation languages such as ATL, Henshin or Viatra for transforming models. Only half as many (27) stated to have used general purpose languages for this goal. Lastly, only 5 indicated the use of internal MTLs.

When looking at the specific dedicated MTLs used ATL is by far the most prominent one used by interviewees. A total of 37 participants mention having used ATL. This is more than double the amount of the second most used language namely Henshin which is only mentioned by 17 interviewees. The QVT family then follows in third place with QvT-R having been used by 13 participants, QvT-O by 11. A complete overview over all dedicated model transformation languages used by our interviewees can be found in Fig. 8. Note that several interviewees mentioned using more than one language, making the total number of data points in this figure larger than 52.

In the group of GPL languages used for model transformation (summarised in Fig. 9), Java is the most used language with 14 participants stating so. Note that several interviewees mentioned using more than one language, making the total number of data points in

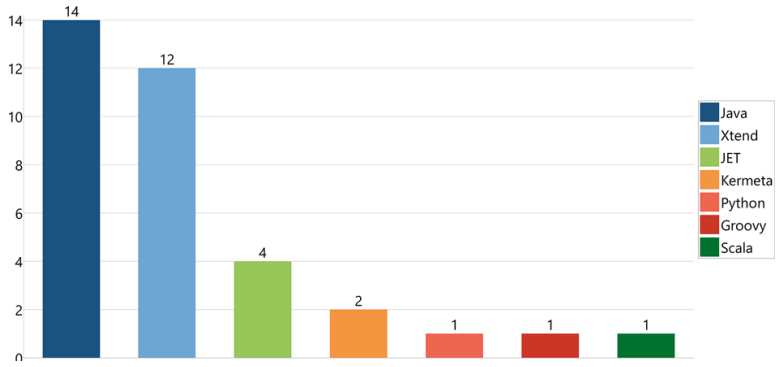


Fig. 9 Number of participants using a specific GPL

this figure larger than 27. Java is closely followed by Xtend which is mentioned by 12 interviewees. Then follows a steep drop of in popularity with Java Emitter Templates having been used by only four participants.

Lastly, only four internal model transformation languages, namely RubyTL, NTL, NMF Synchronizations and FunnyQT, are mentioned. This shows a lack of prominence thereof. Moreover none of the languages is used by more than two interviewees.

5 Findings

Based on the responses of our interviewees and our analysis, we developed a framework to classify influence factors. It allows us to categorize how factors influence properties of MLTs and each other according to our interviewees. Note that we split the property *Reuse & Maintainability* into two properties for the purpose of reporting. This is done because interviewees chose to consider them separately. Thus reporting on them separately allows for presenting more nuanced results.

The factors themselves are split into six top-level factors namely *GPL Capabilities*, *MTL Capabilities*, *Tooling*, *Choice of MTL*, *Skills* and *Use Case*. The first factor, *GPL Capabilities*, encompasses sub-factors related to writing model transformations in general purpose languages. *MTL Capabilities* encompasses sub-factors that originate from transformation specific features of model transformation languages. *Tooling* contains factors surrounding tool support for MTLs. *Choice of MTL* details how the choice of language asserts its influence. The factor *Skills* encompasses sub-factors associated with skills. Lastly, the *Use Cases* factor contains sub-factors that relate to the involved use case an its influences.

Within the framework we differentiate between two kinds of factors. The first kind are factors, that have a positive or negative impact on properties of MTLs. These include the factors *GPL Capabilities*, *MTL Capabilities* and *Tooling* as well as their sub-factors. The second kind are factors that, depending on their characteristic, moderate how other factors influence properties, e.g. depending on the language, its syntax might have a positive or negative influence on the comprehensibility of written code. We call such factors *moderating* factors. These include the factors *Choice of MTL*, *Skills* and *Use Case* and their sub-factors.

Table 3 provides an overview over the answers given by our interviewees. The table shows factors on its rows and MTL properties on its columns. A `+` in a cell denotes, that interviewees mentioned the factor to have a positive effect on their view of the MTL property. A `-` means interviewees saw a negative influence and `+/-` describes that there have been mentions of both positive and negative influences. Lastly, a `M` in a cell denotes, that the factor represents a moderating factor for the MTL property, according to some interviewees. The detailed extent of the influence of each factor is described throughout Sections 5 and 6.

In the following we present all top-level factors and their sub-factors and describe their place within our framework. For each factor we detail its influence on properties of model transformation languages or on other factors, based on the statements made by our interviewees. All statements referred to in this section can be found verbatim in Table 5 in Appendix E.

5.1 GPL Capabilities

Using general purpose languages for developing model transformations, as an alternative to using dedicated languages was extensively discussed in our interviews. Interviewees mentioned both advantages and disadvantages that GPLs have compared to MTLs that made them view MTLs more or less favourably.

Table 3 Overview over quality attribute influences per factor

Top-level Factor	Sub-Factor	Comprehensibility	Ease of Writing	Expressiveness	Maintainability	Productivity	Reuseability	Tool Support
GPL Capabilities		+/-	+/-	-	-	+/-	+	+/-
MTL Capabilities	Domain Focus	+	+	+/-	+	+/-		+
	Bidirectionality	+/-	+/-	+	+/-	+		
	Incrementality	+	+/-	+				
	Mappings	+	+/-	+	+		+/-	
	Traceability	+	+/-	+/-		+		
	Model Traversal	+	+	+		+		
	Pattern Matching	+		+		+		
	Model Navigation	+	+	+				
	Model Management	+	+	+				
	Reuse Mechanisms						+/-	
	Learnability		-					
Tooling	Analysis Tooling	+				+		+/-
	Code Repositories						-	-
	Debugging Tooling	+/-						+/-
	Ecosystem				-	-		-
	IDE Tooling		+/-		-			-
	Interoperability							-
	Tooling Awareness							-
	Tool Creation Effort							-
	Tool Learnability		-					-
	Tool Usability		-			-		-
	Tool Maturity							-
	Validation Tooling							-
Choice of MTL		M	M	M	M	M	M	M
Skills	Language Skills	M	M		M		M	
	User Experience/Knowledge		M		M	M		
	(Meta-) Models		M					
Use Case	I/O Semantic gap	M	M			M		
	Size	M	M					

The disadvantages of GPLs compared to MTLs stem from additional features and abstractions that MTLs bring with them and will be discussed later in Section 5.2. The advantages of GPLs on the other hand can not be placed within the *MTL Capability* factors. These will instead be presented separately in this section.

According to our interviewees, advantages of GPLs are a relevant factor for all properties of MTLs.

General purpose languages are better suited for *writing* transformations that require lots of computations. This is because they were streamlined for these kinds of activities and designed for this task, with language features like streams, generics and lambdas. As a result, general purpose languages are far more advanced for such situations compared to model transformation languages, which sacrifice this for more domain expressiveness [$Q_{gpl}1$].

Much like the language design for GPLs, their tools and ecosystems are mature and designed to integrate well with each other. Moreover, according to several interviewees, their tools are of high quality making developers feel more *Productive* [$Q_{gpl}2$].

Lastly, multiple participants noted, that there are much more GPL developers readily available for companies to hire, thus making GPLs more attractive for them. This helps the *Maintainability* of existing code as such experts are more likely to *Comprehend* GPL code [$Q_{gpl}3$]. Whether this aspect also improves the overall *Productivity* of transformation development in a GPL was disagreed upon, because it might be that developers trained in a MTL could produce similar results with less resources.

It was also mentioned, that much more training resources are available for GPL development, making it easier to start learning and using a new GPL compared to a MTL.

5.2 MTL Capabilities

The capabilities that model transformation languages provide that are not present in GPLs, are important factors that influence properties of the languages. This view is shared by our interviewees that raised many different aspects and abstractions present in model transformation languages.

The influence of capabilities specifically introduced in MTLs is diverse and depends on the concrete implementation in a specific language, the skills of the developers using the MTL and the use case in which the MTL is to be applied. We will discuss all the implications raised by our interviewees regarding the transformation specific capabilities of MTLs for the properties attributed to MTLs in detail, in this section.

5.2.1 Domain Focus

Domain Focus describes the fact that model transformation languages provide transformation specific constructs, abstractions or workflows. Interviewees remarked the domain focus, provided by MTLs, as influencing *Comprehensibility*, *Ease of Writing*, *Expressiveness*, *Maintainability*, *Productivity* and *Tool Support*. But the effects can differ depending on the specific MTL in question.

There exists a consensus that MTLs can provide better domain specificity than GPLs by introducing domain specific language constructs and abstractions. This increases *Expressiveness* by lifting the problem onto the same level as the involved models allowing developers to express more while writing less. MTLs allow developers to treat the transformation problem on the same abstraction level as the involved modelling languages [$Q_{df}1$]. This also improves the *ease of development*.

Several interviewees argued, that when moving towards domain specific concepts the *Comprehensibility* of written code is greatly increased. The reason for this is, that because transformation logic is written in terms of domain elements, unnecessary parts are omitted (compared to GPLs) and one can focus solely on the transformation aspect [$Q_{df}2$].

Having domain specific constructs was also raised as facilitating better *Maintainability*. Co-evolving transformations written in MTLs together with hardware-, technology-, platform- or model changes is said to be easier than in GPLs because *"Once you have things like rules and helpers and things like left hand side and right hand side and all these patterns then [it is] easier to create things like meta-rules to take rules from one version to another version [...]"* (**P23**).

Domain focus also enforces a stricter code structure on model transformations. This reduces the amount of variety in which they can be expressed in MTLs. As a result, developing *Tool Support* for analysing transformation scripts gets easier. Achieving similarly powerful tool support for general purpose languages, and even for internal MTLs, can be a lot harder or even impossible because much less is known solely based on the structure of the code. Analysis of GPL transformations has to deal with the complete array of functionality of general purpose language constructs [$Q_{df}4$]. While MTLs can be Turing complete too, they tend to limit this capability to specific sections of the transformation code. They also make more information about the transformation explicit compared to GPLs. This allows for easier analysis of properties of the transformation scripts which reduces the amount of work required to develop analysis tooling.

The influence of domain abstractions on *Productivity* was heavily discussed in our interviews. Interviewees agreed that, depending on the used language, *Productivity* gains are likely, due to their domain focus. However, one interviewee explained that precisely because of *Productivity* concerns companies in the industry might use general purpose languages. The reason for this boils down to the *Use Case* and project context. Infrastructure for general purpose languages might already be set up and developers do not need to be trained in other languages [$Q_{df}5$]. Moreover, different tasks might require different model transformation languages to fully utilise their benefits, which, from an organisational standpoint, does not make sense for a company. So instead one GPL is used for all tasks.

5.2.2 Bidirectionality

According to our interviewees bidirectional functionality in a model transformation language influences its *Comprehensibility*, *Ease of Writing*, *Expressiveness* and *Maintainability* and *Productivity*. Its effects on these properties then depends on the concrete implementation of the functionality in a *MTL*. It also depends on the *Skills* of the developers and the concrete *Use Case*.

Our interviewees mentioned that the problem of bidirectional transformations is inherently difficult and that high level formalisms are required to concisely define all aspects of such transformations. Many believe that because of this solutions using general purpose languages can never be sufficient. Statements in the vein of *"in a general purpose programming language you would have to add a bit of clutter, a bit of distraction, from the real heart of the matter"* (**P42**) were made several times. This, combined with having less optimal querying syntax, then shifts focus away from the actual transformation and decreases both the *Comprehensibility* and *Maintainability* of the written code.

Maintainability is also hampered because GPL solutions scatter the implementation throughout two or more rules (or methods or files) that have to be adapted in case of

changes [$Q_{bx}2$]. Expressive and high level syntax in MTLs helps alleviate these problems and increases the *ease* at which developers can *write* bidirectional model transformations.

Interviewees also commented on the fact that, thanks to bidirectional functionalities, consistency definitions and synchronisations between both sides of the transformation can be achieved easier. This improves the *Maintainability* of modelling projects as a whole and allows for more *Productive* workflows. Manual attempts to do so have been stated to be error-prone and labour-intensive.

It was also pointed out that the inherent complexity of bidirectionality leads to several problems that have to be considered. MTLs that offer syntax for defining bidirectional transformations are mentioned to be more complex to use as their unidirectional counterparts. They should thus only be used in *cases* where bidirectionality is a requirement. Moreover, one interviewee mentioned that developers are not generally used to thinking in bidirectional way [$Q_{bx}3$].

Lastly, the models involved in bidirectional transformations also play a role regardless of the language used to define the transformation. Often the models are not equally powerful making it hard to actually achieve bidirectionality between them, because of information loss from one side to the other [$Q_{bx}4$].

5.2.3 Incrementality

Dedicated functionality in MTLs for executing incremental transformations has been discussed as influencing *Comprehensibility*, *Ease of Writing* and *Expressiveness*. Similar to bidirectionality its influence is again heavily dependent on the *Use Case* in which incremental languages are applied as well as the *Skills* of the involved developers.

Declarative languages have been mentioned to facilitate incrementality because the execution semantics are already hidden and thus completely up to the execution engine. This increases the *Expressiveness* of language constructs. It can, however, hamper the *Comprehensibility* of transformation scripts for developers inexperienced with the language because there is no direct way of knowing in which order transformation steps are executed [$Q_{inc}1$].

On the other hand interviewees also explained that writing incremental transformations in a GPL is unfeasible. Manual implementations are error-prone because too many kinds of changes have to be considered and chances are high that developers miss some specific kind. Due to the high level of complexity that the problem of incrementality inherently posses interviewees argued that *writing* such transformations in MTLs is much *easier* [$Q_{inc}2$].

The same argumentation also applied for the *Comprehensibility* of transformations. All the additional code required to introduce incrementality to GPL transformations is argued to clutter the code so much that developers "*[will be] in way over their head[s]*" (**P13**).

As with bidirectionality interviewees agreed, that the *Use Case* needs to be carefully considered when arguing over incremental functionality. Only when ad-hoc incrementality is really needed should developers consider using incremental languages. In cases where transformations are executed in batches, maybe even over night, no actual incrementality is necessary and then "*general purpose programming languages are very much contenders for implementing model transformations*" (**P42**). It was also explained that using general purpose languages for simple transformations is common practice in industry as they are "*very good in expressing [the required] control flow*" (**P42**) and because none of the aforementioned problems for GPLs have a strong impact in these cases.

5.2.4 Mappings

The ability of a MTL to define mappings influences that languages *Comprehensibility*, *Ease of Writing*, *Expressiveness*, *Maintainability* and *Reuse* of model transformations. Developer *Skills*, the used *Language* and concrete *Use Case* also play an important role in the kind of influence.

Interviewees agreed, that the *Expressiveness* of transformation languages utilising syntax for mapping is increased due to them hiding low level operations [$Q_{map}1$]. However, as remarked by one participant, the semantic complexity of transformations can not be hidden by mappings, only the computational complexity.

According our interviewees mappings form a natural way of how people think about transformations. They impose a strict structure on how transformations need to be defined, making it easy for developers to start of *writing* transformations. The structure also aids general development, because all elements of a transformation have a predetermined place within a mapping. Being this restrictive has the advantage of directing ones thoughts and focus solely on the elements that should be transformed [$Q_{map}2$]. To transform an element, developers only need to write down the element and what it should be mapped to.

The simple structure expressed by mappings also benefits the *Comprehensibility* of transformations. It allows to easily grasp which elements are involved in a transformation, even by people that are not experienced in the used language. Trying to understand the same transformation in GPLs would be much harder because *"[one] would not recognize [the involved elements] in Java code any more"* (**P32**). Instead, they are hidden in between all the other instructions necessary to perform transformations in the language. Interviewees also mentioned that, due to the natural fit of mappings for transformations, it is much easier to find entry points from where to start and understand a transformation and to reconstruct relationships between input and output. This is aided by the fact that the order of mappings within a transformation does not need to conform with its execution sequence and thus enables developers to order them in a comprehensible way [$Q_{map}3$].

One interviewee explained that, from their experience, mappings lead to less code being written which makes the transformations both easier to *comprehend* and to *maintain*. However, they conceded that the competence of the involved developers is a crucial factor as well. According to them, language features alone do not make code maintainable. Developers need to have language engineering skills and intricate domain knowledge to be able to design well maintainable transformations [$Q_{map}4$]. Both are skills that too little developers posses.

Moreover, several interviewees raised the concern, that complex *Use Cases* can hamper the *Comprehensibility* of transformations. Understanding which elements are being mapped can be hard to grasp if several auxiliary functions are used for selecting the elements. Here one interviewee suggested that a standardized way of annotating such selections could help alleviate the problem.

It was also mentioned that, while mappings and other MTL features increase the *Expressiveness* of the language, they might make it harder for developers to start learning the languages. Because a lot of semantics are hidden behind keywords, developers need to first understand the hidden concepts to be able to utilise them correctly [$Q_{map}5$].

Other features that highlight how much *Expressiveness* is gained from mappings have also been mentioned. Mappings hide how relations between input and output are defined. This creates a formal and predictable correspondence between them and thus enables

Tracing. Moreover, the correspondence between elements allows languages to provide functionality such as *Bidirectionality* and *Incrementality* [$Q_{map}6$].

Because many languages that utilise mappings can forgo definitions of explicit control flow, mappings allow transformation engines to do execution optimisations. However, one interviewee explained that they encountered *Use Cases* where developers want to influence the execution order, forcing them to introduce imperative elements into their code effectively hampering this advantage. It has also been mentioned that in *complex cases* the code within mappings can get complicated to the point where non experts are unable to *comprehend* the transformation again. This problem also exists for *writing* transformations as well. According to one interviewee mappings are great for linear transformations and are thus very dependent on the involved (meta-)models. Also in *cases* where complex interactions needs to be defined mappings do not present any advantage over GPL syntax and sometimes it can even be easier to define such logic in GPLs [$Q_{map}7$].

Lastly, mappings enable more modular code to be written. This in turn facilitates *reuse*, because reusing and changing code results in local changes instead of several changes throughout different parts of GPL code [$Q_{map}8$].

5.2.5 Traceability

The ability in model transformation languages to automatically create and handle trace information about the transformation has been discussed by our interviewees to influence *Comprehensibility*, *Ease of writing*, *Expressiveness* and *Productivity*. However, the concrete effect depends on the *MTL* and the *skill* of users.

All interviewees talking about automatic tracing agreed that it increases the *Expressiveness* of the language utilising it. In GPLs this functionality would need to be manually implemented using structures like hash maps. Code to set up traces would then also need to be added to all transformation rules [$Q_{trc}1$].

However, interviewees disagreed on how much this actually impacts the overall transformation development. Most interviewees felt like automatic trace handling *Eases Writing* transformations and even increases *Productivity* since no manual workarounds need to be implemented. This is because manual implementation requires developers to think about when and in which cases traces need to be created and how to access them correctly. It also enables languages that allow developers to define rules independent from the execution sequence. One interviewee however felt like this was not as effort intensive as commonly claimed and thus automatic trace handling to them is more of a nice to have feature than a requirement for writing transformations effectively. Moreover, for complex *Use Cases* of tracing such as QvTs late resolve, the *Users* are required to understand the principle of tracing [$Q_{trc}2$]. And according to another interviewee teaching how tracing and trace models work is hard.

Comprehending written transformations can also be aided by automatic trace management. Manual implementations introduce unnecessary clutter into transformation code that needs to be understood to be able to understand a whole transformation. This is especially true if effort has been put into making tracing work efficiently, according to one interviewee. Understanding a transformation is much more straight forward when only the established relationships between the input and output domains need to be considered, without any additional code to setup and use traces [$Q_{trc}3$].

Lastly, one interviewee raised the issue that manual trace handling might be necessary to write complex transformations involving multiple source and target models, as current engines are not intended for such *Use Cases*.

5.2.6 Automatic Model Traversal

According to our interviewees, the automatic traversal of the input model to apply transformations influences *Ease of Writing*, *Expressiveness*, *Comprehensibility* and *Productivity*. They also explain that depending on the implementation in a concrete *MTL* the effects can differ. *Use Cases* are also mentioned to be relevant to the influence of automatic traversal.

Automatic model traversal hides the traversal of the input model and how and when transformations are applied to the input model elements. Because of this many interviewees expressed that this feature in MTLs increases their *Expressiveness*. The reduced code clutter also helps with *Comprehensibility*.

It also *Eases the Writing* of transformations because developers do not need to worry about traversing the input and finding all relevant elements, a task that has been described as complicated by interviewees. This can be of significant help to developers. One interviewee explained, that they ran an experiment with several participants where they observed model traversal to be *"one of the biggest problems for test persons"* (**P49**).

Not having to manually define traversal reduces the amount of code that needs to be written and thus increases the overall *Productivity* of development, according to one interviewee. However, there can also be drawbacks from this practice. Hiding the traversal automatically leads to the context of model elements to be hidden from the developer. In cases where the context contains relevant information this can be detrimental and even mask errors that are hard to track down [$Q_{trv}1$].

Lastly, automatic input traversal enables transformation engines to optimize the order of execution in declarative MTLs. And MTLs where no automatic execution ordering can be performed have been described as being "*close to plain GPLs*" (**P52**).

5.2.7 Pattern-Matching

Some model transformation languages, such as Henshin, allow developers to define subgraphs of the model graph, often using a graphical syntax, to be matched and transformed. This pattern-matching functionality influences the *Comprehensibility*, *Expressiveness* and *Productivity*, according to our interviewees. It is, however, strongly dependent on the specific *language* and *Use Case*. The feature is only present in a small portion of MTLs and brings with it its own set of restrictions depending on the concrete implementation in the language.

Pattern-matching functionality greatly increases the *Expressiveness* of MTLs. Similar to the basic model traversal no extra code has to be written to implement this semantic. However, the complexity of the abstracted functionality is even higher, since it is required to perform sub-graph matching to find all the relevant elements in a model. These patterns can also become arbitrarily complex and thus all interviewees talking about pattern-matching agreed that manual implementations are nearly impossible. Nevertheless one interviewee mentioned, that all languages they used that provided pattern-matching functionality (Henshin and TGG) had the drawback of providing no abstractions for resolving traces which takes away from its overall usefulness for certain *Use Cases* [$Q_{pm}1$].

Not having to implement complex pattern-matching algorithms manually is also mentioned to increase the *Productivity* of writing transformations because this task is labour-intensive and error-prone.

Improvements for the *Comprehensibility* of transformations have also been recognized by some interviewees. They explained that the, often times graphical, syntax of languages with pattern-matching functionality allows to directly see the connection between involved

elements. In GPLs this would be hidden behind all the code required to find and collect the elements. As such MTL code is *"less polluted"* (**P52**) than GPL code. Moreover, the *Comprehensibility* is also promoted by the fact that in some languages the graphical syntax shows the involved elements as they would be represented in the abstract syntax of the model.

5.2.8 Model Navigation

Dedicated syntax for expressing model navigation has influence on the *Comprehensibility* and *Ease of Writing* of model transformations as well as on the *Expressiveness* of the *MTL* that utilises it.

Having dedicated syntax for model navigation helps to *ease development* as it allows transformation engineers to simply express which elements or data they want to query from a model while the engine takes care of everything else. Furthermore, it has been mentioned that this has a positive effect on transformation development because developers do not need to consider the efficiency of the query compared to when defining such queries using nested loops in general purpose languages [$Q_{nav}1$].

Because languages like OCL abstract from how a model is navigated to compute the results of a query, interviewees attributed a higher *Expressiveness* to them than GPL solutions and described code written in these languages as more concise. Several interviewees attribute a better *Comprehensibility* to OCL as a result of this conciseness, arguing that well designed conditions and queries written in OCL are easy to read [$Q_{nav}2$].

OCL has however also been criticised by an interviewee. According to them, the language is too practically oriented, misses a good theoretical foundation and lacks elegance to properly express ones intent. They explain that because of this, the worth of learning such a language compared to using a more common language is uncertain.

5.2.9 Model Management

The impact of having to read and write models from and to files, i.e., model management, has been discussed by several interviewees. Automatic model management was discussed in our interviews as influencing the *Comprehensibility*, *Ease of Writing* and *Expressiveness* of model transformations in MTLs.

The argument for all three properties boils down to developers not having to write code for reading input models or writing output models, as well as the automatic creation of output elements and the order thereof. Interviewees agreed that implicit language functionality for these aspects raised the *Expressiveness* of languages. It reduces clutter when reading a written transformation and thus improving the *Comprehensibility*. Finally, developers do not have to deal with model management tasks, e.g. using the right format, that are not relevant to the actual transformation which helps with *writing* transformations [$Q_{man}1$].

5.2.10 Reuse Mechanism

Mechanisms to reuse model transformations mostly influence the *Reusability* of model transformations in MTLs. Their concrete influence depends on the used *Language* and how reuse is handled in it. Interviewees also reported on cases where the users *Skills* with the language was relevant because novices might not be familiar with how the provided facilities can be utilised to achieve reuse.

There exists discourse between the interviewees about reuse mechanisms and their usefulness in model transformation languages. Several interviewees argued that MTLs do not have any reuse mechanisms that go beyond what is already present in general purpose languages. They believe that most, if not all, the reuse mechanisms that exist in MTLs are already present in GPLs and as such MTLs do not provide any reuse advantages [$Q_{rm}1$]. According to them such reuse mechanisms include things like rule inheritance from languages like ATL or modules and libraries.

Other interviewees on the other hand suggested that while the aforementioned mechanisms stem from general purpose languages, they are still more transformation specific than their GPL counterpart. This is, because the mechanisms work on transformation specific parts in MTLs rather than generic constructs in GPLs [$Q_{rm}2$, $Q_{rm}3$]. Because of this focus, interviewees argue that they are more straight forward to use and thus improve *Reusability* in MTLs.

Interviewees also explained that there exist many languages that do not provide any useful reuse or modularisation mechanisms and that even in those that do it can be hard to achieve *Reusability* in a useful manner. However, one participant acknowledged that in their case, the reason for this might also relate to the inability of the *Users* to properly utilize the available mechanisms.

It has also been mentioned that reuse in model transformations is an inherently complex problem to solve. Transferring needs between two transformations which apply on different meta-models is difficult to do. As such, model transformation are often tightly tied to the domain in which they are used which makes reuse hard to achieve and most reuse between domains is currently done via copy & paste. This argument can present a reason why, as criticised by several interviewees, no advanced reuse mechanisms are broadly available.

The desire for advanced mechanisms has been expressed several times. One interviewee would like to see a mechanism that allows to define transformations to adapt to different input and output models to really feel like MTLs provide better reusability than GPLs. Another mentioned, that all reuse mechanisms conferred from object orientation rely on the tree like structure present in class structures while models are often more graph like and cyclic in nature. They believe that mechanisms that address this difference could be useful in MTLs.

Another disadvantage in some MTLs that was raised, is the granularity on which reuse can be defined. In languages like Henshin, for example, reuse is defined on a much coarser level than what is possible in GPLs.

Not having a central catalogue, similar to maven for Java, from which transformations or libraries can be reused, has also been critiqued as hindering reuse in model transformation languages.

5.2.11 Learnability

The learnability of model transformation languages has been discussed as influencing the *Ease of Writing* model transformations.

It has been criticised by several interviewees, that the learning curve for MTLs is steep. This is, in part, due to the fact that users not only need to learn the languages themselves, but also accompanying concepts from MDE which are often required to be able to fully utilise model transformation languages. The learning curve makes it difficult for users to get started and therefore hampers the *Ease of Writing* transformations [$Q_{ler}1$]. This effect could be observed among computer science students at several of the universities of our interviewees.

The students were described to having difficulties adapting to the vastly different approach to development compared to more traditional methods. A potential reason for this could be that people come into contact with MDE principles too late, as noted by an interviewee [$Q_{ler}2$].

5.3 Tooling

While *Tool Support* is a MTL property that was investigated in our study, the tooling provided for MTLs, as well as several functional properties thereof, have been raised many times as factors that influence other properties attributed to model transformation languages as well. Most of the time this influence is negative, as tooling is seen as one of the greatest weak points of model transformation languages by our interviewees.

Many interviewees explained, that the most common languages do in fact have tools. The problem, however, lies in the fact that some helpful tools only exist for one language while others only exist for another language. As a result there is always some tool missing for any specific language. This leads people to feel like *Tool Support* for MTLs is bad compared to GPLs. Though there was one interviewee that explained that for their *Use Cases*, all tools required to be productive were present.

In the following, we will present several functional properties and tools that interviewees expressed as influential for *Tool Support* as well as other properties of MTLs.

5.3.1 Analysis Tooling

Analysis tools are seen as a strong suit of MTLs. Their existence in MTLs is said to impact *Productivity*, *Comprehensibility* and perceived *Tool Support*.

According to the interviewees, some analyses can only be carried out on MTLs, as the abstraction in transformations in GPLs is not high enough and too much information is represented by the program logic and not in analysable constructs. As one interviewee explained, this comes from the fact that for complex analysis, such as validating correctness, languages need to be more structured. Nevertheless, participants mainly mentioned analyses they would like to see, which is an indication that, while the potential for analysis tools for MTLs is high, they do not yet see usable solutions for it, or are unaware of it. This is highlighted by one interviewee that explained that they are missing ways to check properties of model transformations, even though such solutions exist for certain MTLs [$Q_{db}1$].

A desired analysis tool mentioned in the interviews is rule dependency analysis and visualisation. They believe that such a tool would provide valuable insights into the structure of written transformations and help to better *comprehend* them and their intent. *"What I would need for QVT-R, for example, in more complex cases, would be a kind of dependency graph."* (P32). Moreover two interviewees expressed the desire for tools to verify that transformations uphold certain properties or preserve certain properties of the involved models.

5.3.2 Code Repositories

A gap in *Tool Support* that has been brought up several times, is a central platform to share transformation libraries, much like maven-central for Java or npm for JavaScript. This tool influences *Tool Support* and the *Reusability* of MTLs.

According to two interviewees, not having a central repository where transformations, written by other developers, can be browsed or downloaded, greatly hinders their view

on the *Reusability* of model transformation languages. This is because it creates a barrier for reuse. For one thing, it is difficult to find model transformations that can be reused. Secondly, mechanisms that would simplify such reuse are then also missing. *"I think what is currently missing is a catalogue or a tool like maven for having repositories for transformations so you can possibly find transformations to reuse."* (**P14**)

5.3.3 Debugging Tooling

Debuggers have been raised as essential tools that help with the *Comprehensibility* of written model transformations. The existence of a debugger for a given language therefore influences its *Tool Support* as well as its *Comprehensibility*.

One interviewee explained that, especially for declarative languages, where the execution deviates greatly from the definition, debugging tools would be a tremendous help in understanding what is going on. In this context, opinions were also expressed that more information is needed for debugging model transformations than for traditional programming and that the tools should therefore be able to do more. Interviewees mentioned the desire to be able to see how matchings arise or why expected matches are not produced as well as the ability to challenge their transformations with assertions to see when and why expressions evaluate to certain values. *"Demonstrate to me that this is true, show me the section of the transformation in which this OCL constraint is true or false."* (**P28**).

Valuable debugging of model transformations is mainly possible in dedicated MTLs, according to one interviewee. They argue that debugging model transformations in GPLs is cumbersome because much of the code does not relate to the actual transformation thus hampering a developers ability to grasp problems in their code.

5.3.4 Ecosystem

The ecosystems around a language, as well as existing ecosystems, in which model transformations languages would have to be incorporated into, were remarked as mostly limiting factors for *Productivity*, *Maintainability* as well as the perceived amount of *Tool Support*.

One interviewee explained, that for many companies, adopting a model transformation language for their modelling concerns is out of the question because it would negatively impact their *Productivity*. The reason for this are existing ecosystems, which are designed for GPL usage. Moreover, it was noted that, to fully utilise the benefits of dedicated languages many companies would need to adopt several languages to properly utilise their benefits. This is seen as hard to implement as *"people from industry have a hard time when they are required to use multiple languages"* (**P49**) making it hard for them to *maintain* code in such ecosystems.

Ecosystems surrounding MTLs have also been criticised in hampering *Productivity* and perceived *Tool Support*. Several interviewees mentioned, that developers tend to favour ecosystems where many activities can be done in one place, something they see as lacking in MTL ecosystems. One interviewee even referred to this problem as the reason why they turned away from using model transformation languages completely [$Q_{eco}2$].

This issue somewhat contrasts a concern raised by a different group of interviewees. They felt that the coupling of much of MDE tooling to Eclipse is a problem that hampers the adoption of MTLs and MDE. This coupling allows many tools to be available within the Eclipse IDE but, according to them, the problem lies in the fact that Eclipse is developed at a faster pace than what tool developers are able to keep up with, leaving much of the *Tool Support* for MTLs in an outdated state, limiting their exposure and usability [$Q_{eco}3$].

5.3.5 IDE Tooling

One essential tool for *Tool Support*, *Ease of Writing* and *Maintainability* of MTLs are language specific editors in IDEs.

Several interviewees mentioned, that languages without basic IDE support are likely to be unusable, because developers are used to all the quality-of-life improvements, with autocompletion and syntax highlighting being the two most important features offered by such tools. Refactoring capabilities in IDEs, like renaming, have also been raised as crucial, especially for easing the *Maintainability* of transformations.

5.3.6 Interoperability

How well tools can be integrated with each other has been raised as a concern for the *Tool Support* of MTLs by several interviewees.

Interviewees see a clear advantage for GPLs when talking about interoperability between different MTL tools. They believe, that due to the majority of tools being research projects, little effort is spent into standardizing those in a way that allows for interoperability on the level that is currently provided for GPLs. *"But the technologies, to combine them, it is difficult [...]"* (**P36**). One interviewee described their first hand experience with this. They could not get a MTL to process models they generated with a software architecture tool because it produced non standard UML models which could not be used by the MTL. This problem has been echoed by another interviewee who explained that many MTLs do not work with non EMF compatible models.

5.3.7 Tooling Awareness

A few interviewees talked about the availability of information about tools and the general awareness of which tools for MTLs exist. According to them, this strongly influences the perceived lack of *Tool Support* for model transformation languages in general.

When starting out with model transformations it can be hard to find out which tools one should use or even which tools are available at all. Two interview participants mention experiencing this first hand. They further explain that there exists no central starting point when looking for tools and tools are generally not well communicated to potential users outside of research [$Q_{awa}1$].

Another interviewee suspected that the same problem also happens the other way around. They believe that some well designed MTL tools are completely unknown outside of the companies that developed them for internal use.

5.3.8 Tool Creation Effort

The amount of effort, that is required to be put into the development of MTL tools, has been raised by many interviewees as a reason why *Tool Support* for MTLs is seen as lacking.

All interviewees talking about the effort involved in creating tools for MTLs agree that there is a lot of effort involved in developing tools. This is not a problem in and of itself but, when comparing tooling with GPLs interviewees felt like MTLs being at a disadvantage. The disadvantage stems from the community for MTLs being much smaller and thus having less man power to develop tools which limits the amount of tools that can be developed.

Several interviewees noted, that the only solution they see for this problem is industrial backing or commercial tool vendors because *"I am keenly aware of the cost to being able to develop a good programming language, the cost of maintaining it and the cost of adding debuggers and refactoring engines. It is enormous."* (**P1**).

When comparing the actual effort for creating transformation specific tools, some interviewees explained that their experience suggests easier tool development for MTLs than for GPLs. They explained that, extracting the transformation specific information necessary for such tools out of GPL code complicates the whole process, whereas dedicated MTLs with their small and focused language core provide much easier access to such information [$Q_{tce}2$].

5.3.9 Tool Learnability

The learning curve for someone starting off with MTLs and MTL tools is discussed as a heavy burden to the perceived effectiveness of *Tools* and even influences *Ease of Writing*.

Several interviewees criticised the fact that when starting off with a new MTL and its accompanying tools there is little support for users. Many tools lack basic documentation on how to set them up properly and how to use them. As a result users feel lost and find it difficult to start off *writing* transformations [$Q_{tle}1$].

5.3.10 Tool Usability

Related to the topic of learnability, the usability of tools for model transformation languages is discussed as influencing the quality of *Tool Support* for the languages as well as the *Ease of Writing* and *Productivity*.

To fully utilise the potential of MTLs useable tools are essential. Due to their higher level of abstraction, high quality tools are necessary to properly work with them and *Write* well rounded transformations [$Q_{use}1$].

This is currently not the case when looking at the opinions of our interviewees talking about the topic of tool usability. There are tools available for people to start off with developing transformations but they are not well rounded and thus not ready for professional use, according to one interviewee. This is supported by several other interviewees opinions, many tools are faulty, which hinders the workflow and reduces *Productivity* [$Q_{use}2$]. It has also been stated that if there were high quality useable tools available, they would be used. The reality for many users is, however, more in line with the experience of one interviewee who stated that they were unable to get many tools (for bidirectional languages) to even work at all.

5.3.11 Tool Maturity

A reason given for many of the criticised points surrounding MTL tools is their maturity. It is said to be a pivotal factor for everything related to *Tool Support*.

The maturity of tools for model transformation languages was commented on a lot. Tools need to be refined more in order to raze many of their current faults. The fact that this is not currently done relates back to the effort that is involved with it and the limited personnel available to do so. This is highlighted in an argument made by one of the interviewees who feels, that the community should not be hiding behind the argument of maturity [$Q_{mat}1$].

5.3.12 Validation Tooling

Tools or frameworks to support the validation and testing of transformations written in MTLs have been discussed to influence the perceived *Tool Support* for nearly all MTLs.

Too much of the available tool support focuses solely on the writing phase of transformation development. There is little tool support for testing developed transformations, which has been raised as an area where much progress can, and has to be, made. Especially when comparing the current state of the art with GPLs, MTLs are seen as lacking [$Q_{val}1$]. Not only are there little to no tools like unit testing frameworks, there is also too few transformation specific support such as tools to specifically verify binding or helper code in ATL.

5.4 Choice of MTL

The choice of MTL is an obvious factor that influences how other factors, such as the *MTL Capabilities*, influence the properties of model transformation languages. However, it should be explicitly mentioned, because it has been brought up countless times by interviewees while not often being considered in literature. Depending on the chosen model transformation language its capabilities and whole makeup changes, which has strong implications on all aspects of model transformation development.

A large number of the interviewees have commented on this. They either directly raised the concern, by prefacing a discussion with a statement such as *"[...] it depends on the MTL"*, or indirectly raised the concern, when comparing specific languages that do or do not exhibit certain capabilities and properties.

5.5 Skills

Skills of involved stakeholders is another group of factors that does not have a direct influence on how MTLs are perceived but instead plays a passive role. Many interviewees cited skills as a limiting factor to other influence factors. They argue that insufficient user skills could hinder advantages that MTLs can provide and might even create disadvantages compared to the more well-known and commonly used GPLs.

In this section we present the different types of skills mentioned by our interviewees as being relevant to the discussion of properties of model transformation languages.

5.5.1 Language Skills

The skill of developers in a specific model transformation language was raised by several interviewees as critical in facilitating many of the advantages provided through the languages capabilities. So much so that, according to them, the ability of developers to use and read a language can make or break any and all advantages and disadvantages of MTLs related to *Comprehensibility*, *Ease of Writing*, *Maintainability* and *Reuseability*.

Basic skills in any language are a prerequisite to being able to use it. They are also necessary to understand written code. There is no difference between GPLs and MTLs. It was however mentioned, that developers are generally more used to the development style in general purpose languages. Thus users need to learn how to solve a problem with the functionality of the model transformation language to be able to successfully develop transformations in a MTL [$Q_{skl}1$]. This is especially relevant for complex transformations, where users are required to know of abstractions such as tracing or automatic traversal.

Following on on this, one interviewee explained, that while learning the language is a requirement, using a new library, e.g. one for developing model transformations, in a GPL also entails learning and as such this must not be regarded as a disadvantage.

For *reuse* it is also paramount for users to know what elements of a transformation can be reused through language functionality. As a result the *Reuseability* is again limited by the knowledge of users in the specific language.

Lastly, being able to *maintain* a transformation written in an MTL also requires users to know the language to be able to understand where changes need to be made [$Q_{skl}2$].

5.5.2 User Experience/Knowledge

Apart from mastering a used language, the amount of experience users have with said languages and techniques also play a vital role in bringing out the full potential of said languages. Our interviewees discussed this for *Ease of Writing*, *Maintainability* and *Productivity*.

One interviewee explained that, from their experience, the amount of experience developers have with a language greatly impacts their *Productivity* when using said language. The problem for MTLs that results from this is the fact that there is little incentive for a person that is trying to build up their CV to spend much time on dedicated languages such as MTLs [$Q_{exp}1$]. Developers are more inclined to learning and accumulating experience in languages that are commonly used in different companies to improve their chances of landing jobs. As a result people tend to have little to no experience in using MTLs. This in turn results in them having a harder time *developing* transformations in these languages, and the final product being of lower quality than what they could achieve using a GPL in which they have more experience in.

The problem is further exacerbated in teaching. *"Many MDSE courses are just given too late, when people are too acquainted with GPLs, and then its really hard for students to see the point of using models, modelling and MTLs, because it's comparable with languages and stuff they have already learned and worked with."* (**P6**).

5.6 Use Case

Similar as the *MTL* itself and stakeholder *Skills*, the concrete *Use Case* in which model transformations are being developed is another factor that does not directly influence how properties of MTLs are being assessed. Instead, interviewees often mention that, depending on the *Use Case*, other influence factors could either have a positive or negative effect.

Use cases are distinguished along three dimensions. The complexity of involved models based on their structure, the complexity of the transformation based on the semantic gap between source and target, and the size of the transformation based on the use case. Depending on which differentiation is referred to by the interviewees, the considerations look differently.

5.6.1 Involved (Meta-) Models

The involved models and meta-models can have a large impact on the transformation and can hence heavily influence the advantages or disadvantages that MTLs exhibit.

Writing transformations for well behaved models, meaning models that are well structured and documented, can be immensely productive in a MTL while 'badly' behaved models bring out problems that require well trained experts to properly solve in a MTL. The

UML meta-model was put forth as an example for such a badly behaved meta-model by one interviewee. According to them, transformations involving UML models can be problematic due to templates, which are model elements that are parametrized by other model elements, and `eGenericType`. The problem with these complex model elements is often worsened by low-quality documentation [$Q_{mod}1$]. In cases where these badly behaved models are involved, many of the advantages from advanced features of MTLs can not be properly utilised without powerful tooling.

5.6.2 Semantic Gap Between Input and Output

Many interviewees formulate considerations based on the differentiation between 'simple' and 'complex' transformations in terms of the semantic gap that needs to be overcome. Transformations are considered simple when there is little semantic difference between the source and target models. Common comparisons read like: *"transforming boxes into circles"* (**P32**).

For simple transformations, model transformation languages are regarded as taking a lot of work off of the developers through the different language features discussed in Section 5.2. In more complex cases, transformations will get more complex and the developers experience gets more and more relevant, as more advanced language features need to be utilised, which can favour GPLs [$Q_{gap}1$].

Others argue that the advantages of MTLs only really come into play in more complex cases or when high level features, such as bidirectionality or incrementality, are required. The reasoning for this argument is, that in simple cases the overhead of GPLs is not that prominent. Moreover, for writing complex transformations, dedicated query languages in MTLs are regarded by some to be much better than having to manually define complex conditions and loops in a GPL.

5.6.3 Size

The Size of the transformation based on the *Use Case* is considered by some interviewees to be a relevant factor as well. In cases with many rules that depend on each other, MTLs are seen as having advantages [$Q_{siz}1$]. The size of transformations has been said to be a limiting factor for the use of graphical languages as enormous transformations would make graphical notations confusing. Modularisation mechanisms of languages also become a relevant feature in these cases.

6 Cross-Factor Findings

Based on interview responses, we developed a structure model from structural equation modeling (Weiber and Mühlhaus 2021) that models interactions between the presented influence factors and the properties of model transformation languages.

Structure models depict assumed relationships between variables (Weiber and Mühlhaus 2021). They divide their components into endogenous and exogenous variables. The endogenous variables are explained by the causal influences assumed in the model. The exogenous variables serve as explanatory variables, but are not themselves explained by the causal model. Exogenous variables either directly influence a endogenous variable or they moderate an influence of another exogenous variable on an endogenous variable.

Structure models are therefore well suited to provide a theoretical framework for the findings of our work. Factors identified during analysis constitute exogenous variables while MTL properties constitute endogenous variables. Moderating factors also constitute exogenous variables, with the caveat of only having moderating influences on other influences.

A graphical overview over the influences identified by us can be found in Fig. 10. The detailed structure model is depicted in Fig. 11.

The structure model depicts which MTL properties are influenced by which of the identified factors. For each MTL property the model also illustrates which factors moderate the influence on the property. Rectangles represent factors, rounded rectangles represent MTL properties. Below each MTL property the moderating factors for the property are displayed. Arrows between a factor and a MTL property represent the factor having an influence on the MTL property. Each influence on a MTL property is moderated by its moderating factors. The graphical representation deviates from standard presentation due to its size.

The capabilities of model transformation languages based on domain specific abstractions are aimed at providing advantages over general purpose languages. Whether these advantages are realised or whether disadvantages emerge is moderated by the *Skills* of the users, the concrete *MTL chosen* as well as the *Use Case* for which transformations are applied. Depending on these organisational factors the versatility provided by general purpose languages may overshadow advantages provided by MTLs.

Tooling can aid the usage of advanced features of MTLs by supporting developers in their endeavours beyond simple syntax highlighting. As a result, tools can further promote the advantages that stem from the domain specific abstractions. The biggest problem that tools for MTLs face is their availability and quality.

In the following we will present thorough discussions of the most salient observations based on our interviews and the presented structure model. Note that, as will be thoroughly discussed in Section 8.2, the observations have a limited applicability for industry use cases due to the lack of interviewees that use MTLs in an industry setting.

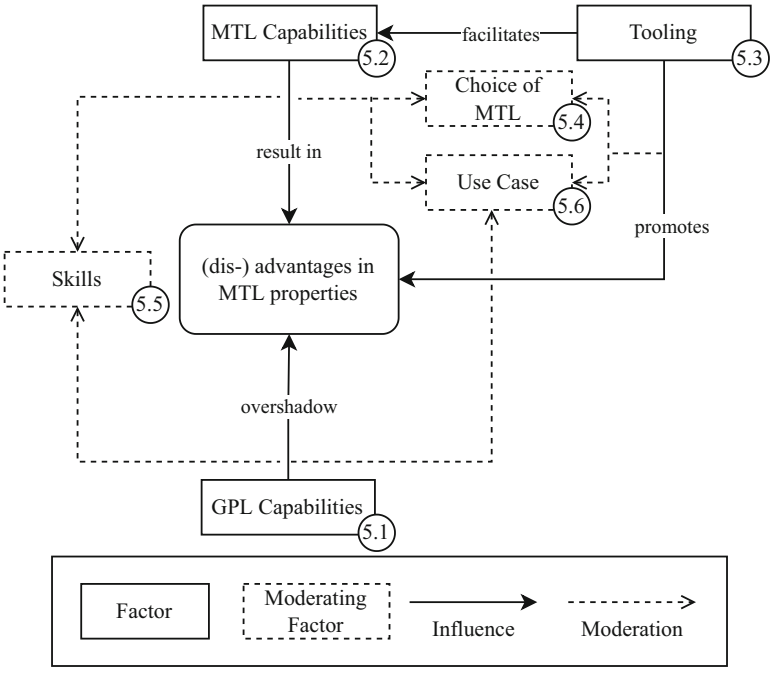


Fig. 10 Graphical overview over factor influences and moderations

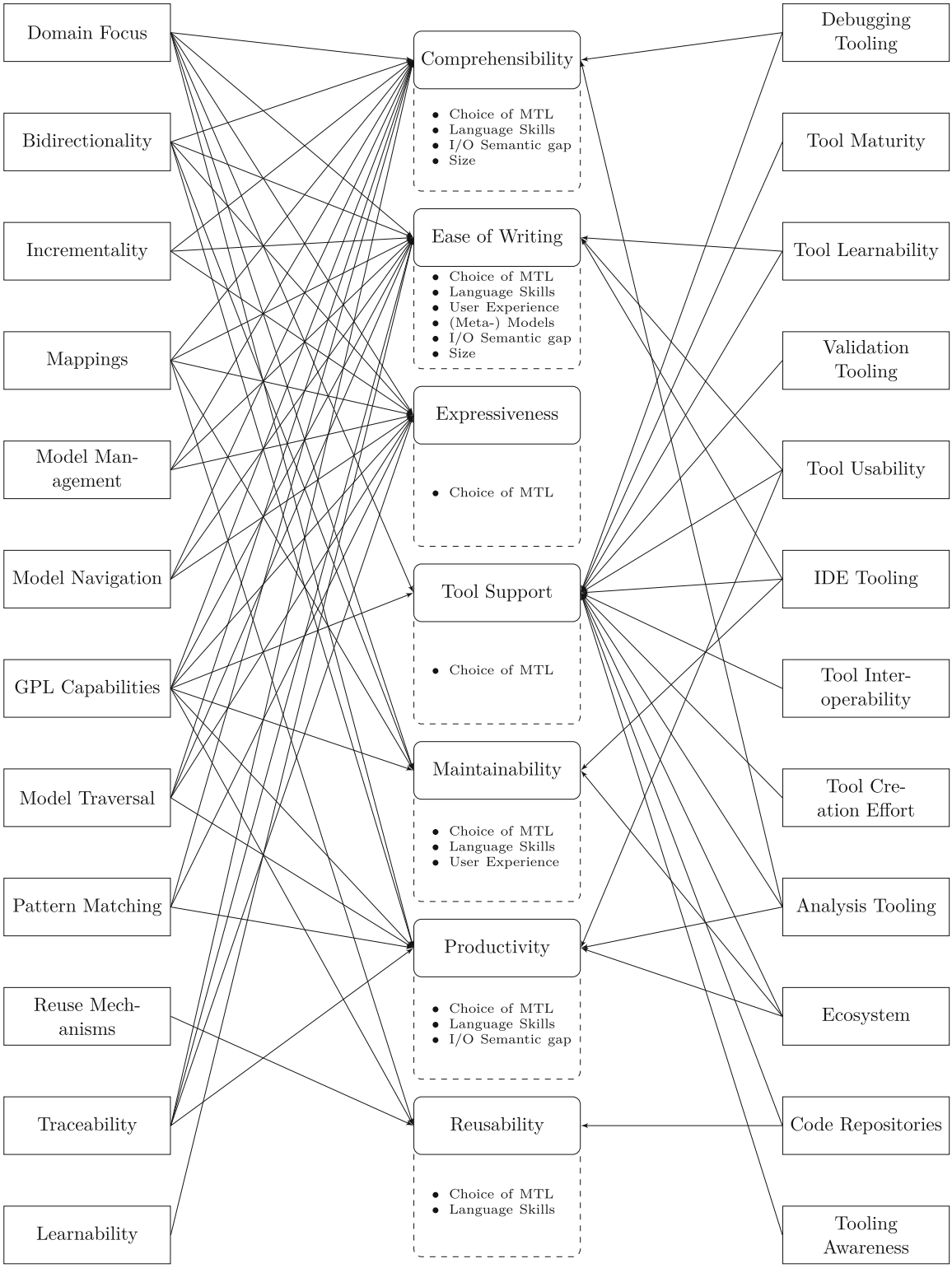


Fig. 11 Structure model of influence and moderation effects of factors on MTL properties (Due to its size, the model has been made interactive. Please refer to the supplementary materials for the interactive version.)

6.1 The Effects of MTL Capabilities

Capabilities of model transformation languages that go beyond what general purpose languages can offer, are regarded as opportunities for better support in development of transformations. The advantage often boils down to not having to manually implement the functionality in question when it is required. It also helps reduce clutter in transformation code, putting the mapping of input and output at the centre of attention. Moreover, they aid developers in handling problems specific to the transformation domain, such as synchronisations and the relationship of input and output values.

This does however come with its own set of limitations. Model transformation languages favour a different way of problem solving that is well suited to the problem at hand, but is unfamiliar for the common programmer. This is amplified by an education that is heavily focused on imperative programming and lacks deeper exposure to logical and functional programming. Knowledge and understanding of functional concepts would help developers when using query languages such as OCL, while logical concepts often find application in graph based transformation languages. The domain specific mechanisms in model transformation languages also make generalisations harder. This is highlighted in the discussions regarding reusability. Interviewees commonly referred to transformations as conceptionally hard to reuse because of their specificity that makes them applicable only to the use case for which they were developed.

6.2 Tooling Impact on Properties Other than Tool Support

Tooling, or the lack thereof, is a main factor that influences how people perceive the quality and availability of usable model transformation languages. However, our interviews show that tooling also facilitates many other properties. This is because tools are not developed as an end in themselves. Tools are intended to support developers in their efforts to develop and maintain their code.

As a result, the quality of available tools is a major factor that impacts all aspects of a MTL. *"Basically all the good aids you see in a Java environment should be there even better in a MTL tool, because model transformation is so much more abstract and more relevant that you should be having tools that are again more abstract and more relevant."* (**P28**). Problems in the area of *Usability*, *Maturity* and *Interoperability* of tools have also been reported on in empirical studies on MDE in general (Whittle et al. 2013; Mohagheghi et al. 2013).

Herein also lies the biggest problem for model transformation languages. The quality of tools is inadequate. While there do exist good and useable tools, they are far and between, only exist for certain languages and are not integrated with each other. This greatly diminishes the potential of model transformation languages because, compared to general purpose languages, developing with them can often be scattered over multiple separate workflows and tools. There do exist many tools, but most of them are prototypical in nature and only available for individual languages. This makes it hard to fully utilise the capabilities of a MTL when suitable tools only exist in theory.

The lack of good tools can be attributed mainly to the amount of work required to develop them and the comparatively small community. Moreover, there are no large commercial vendors, that could put in the required resources to develop tools of a commercially viable quality.

6.3 The Importance of Moderating Factors

The saying *"Use the right tool for the job"* also applies to the context of model transformations. One of the most important things to note is that depending on the context, such as *Use Case* and *developer Skills*, the right language to use can differ greatly. This was highlighted time and time again in our interviews.

Interviewees insisted that the combination of use case and the concrete implementation of a language feature significantly change how well a feature supports properties such as *Comprehensibility* or *Productivity*, i.e. the influence of factors is moderated by 'contextual' factors. For one, the implementation of a feature in a language might not fit well for the problem that needs to be solved. Or the feature is not required at all and thus could impose effort on developers that is seen as unnecessary. In our opinion, this stems from the use cases language developers intended the language for. For example, a language such as Henshin is intended for cases where patterns of model elements need to be matched and manipulated. In such cases, the features provided by Henshin can bring significant advantages over implementing the intended transformation in general purpose languages. Other use cases, where these features are not required, bring no advantage. They can even have negative effects as the language design around them might hinder users from developing a straight forward solution.

The skills and background knowledge of users is relevant, as it can greatly influence how comfortable people are in using a language. This in turn reflects on how well they can perform. This is problematic for the adoption of model transformation languages, as programmers tend to be trained in imperative, general purpose languages. As a result, a gentle learning curve is essential and the initial costs of learning need to bear fruit in adequate time. The choice of using an MTL is therefore a long term investment that is not necessarily suited for only a single project.

The considerations around *Use Case*, *Skills* and the *Choice of MTL* are not novel, but they are rarely discussed explicitly. This is concerning because almost any decision process will come back to these three factors and their sub-factors, as seen in the fact that the influence on each MTL property in Fig. 11 is moderated by at least one of them. They provide organisational considerations that come into play before transformation development begins. Moreover, organisational concerns have already been identified as relevant factors for general MDE adoption (Whittle et al. 2013; Hutchinson et al. 2011; Hutchinson et al. 2011). As such they have to be at the centre of attention of researchers and language developers too.

7 Actionable Results

In this section we present and discuss actionable results that arise from the responses made by our interviewees and analysis thereof. Results will largely focus on actions that can be taken by researchers, because they make up the largest portion of our interview participants.

7.1 Evaluation and Development of MTL Capabilities

Our interviewees mentioned a large number of model transformation language capabilities and reasoned about their implications for the investigated properties of MTL. We believe

the detailed results of the interviews can form a basis for further research into two key aspects:

(I) backing up the expert opinions with empirical data
(II) improving existing model transformation languages

7.1.1 Evaluation of MTL Capabilities and Properties

In our interviews, experts voiced many opinions on how and why factors influence the various MTL aspects examined in our study. The opinions were always based on personal experiences, experiences of colleagues and reasoning. We therefore believe, that our results provide a good insight into the communities sentiment and show that there exists consensus between the experts in many aspects. Model transformation language capabilities are considered largely beneficial, except for certain edge cases. However, empirical data to support this consensus is still missing. The lack of empirical studies into the topic of model transformation languages has already been highlighted in our preceding study (Götz et al. 2021).

We are firmly convinced that researchers within the community need to carry out extensive empirical studies, to back up the expert opinions and to explore the exact limits that interviewees hinted at.

We envision two main types of studies, experiments and case studies. Setting up experiments that consider real-world examples with a large number of suitable participants would be optimal but is hard to achieve. Introducing large transformation examples in experiments is very time-consuming and requires that all participants are experts in the languages used. In addition, recruiting appropriate participants is a generally difficult in software engineering studies (Rainer and Wohlin 2021). We recommend using existing transformations as the object of study in experiments instead. This enables the analysis of complex systems to generate quantitative data without involving human subjects.

If the assessments and experiences of developers are to be the central object of study, we recommend to set up case studies. This allows researchers to study effects in complex, real-world settings over a longer period of time. This is important because the exact effects of, for example, the use of a certain language feature often only become apparent to developers after a long period of use. Case studies of research projects or even industrial transformation systems can thus be used to obtain detailed information on the impact of the applied technologies.

To design such studies, our results can form an important basis.

(a) Empirical factor evaluation. How and under which circumstances the factors we have identified affect MTL properties needs to be comprehensively evaluated. Here we envision both qualitative and quantitative studies that focus on the impact of a single factor or a group of related factors. These could, for example, make comparisons between cases where a factor does or does not apply. The results of such studies can help language developers make decisions about features to include in their MTLs.

Our interviews provide extensive context that should be taken into account in the study design and interpretation of results. For example, our interviews show that the semantic gap between input and output defines a relevant context that needs to be considered. For this reason, when investigating the advantages and disadvantages of mappings, transformations involving models with different levels of semantic gaps between input and output have to be

used, to be able to fully evaluate all relevant use cases. Some transformations need to contain complicated selection conditions or complex calculations for attributes while others need to have less complicated expressions. Researchers can then evaluate how well mappings in a language fit the different scenarios to aid in providing a clear picture of their advantages and disadvantages.

(b) Empirical MTL property evaluation. What advantages or disadvantages MTLs really have is still up for debate. We believe that the credibility of research efforts on MTL can be greatly improved with studies that provide empirical substantiation to the speculated properties. Advances like those made by Hebig et al. (2018) are rare and further ones, based on real world examples, must be carried out.

Our results can also make a valuable contribution to such studies. The factors we have identified as influencing a property can be taken into account in studies from the outset. They can be used to formulate null hypotheses on why a MTL is superior or inferior to a GPL when considering one specific property.

For example, a study that is interested in investigating the *Comprehensibility* of MTLs compared to GPLs can find a number of factors in our results that need to be taken into account. Such factors include tracing mechanisms, mappings or pattern matching capabilities. Researchers can consciously decide which of them are relevant for the transformations used in the study and what impact their presence or absence has on the study results. Based on these considerations hypotheses can be formed.

A recent study we conducted provides an example of how these considerations can be used to expand the body of empirical studies on this topic (Höppner et al. 2021). By focusing the investigation on *Mappings*, *Model Navigation* and *Tracing* we were able to present clear and focused results for comparing and explaining differences in the expressiveness of transformation code written in ATL and Java. We concentrated our analysis on these factors because they all influence Expressiveness according to our interviews.

Such considerations should of course be part of any proper study, but our results provide a basis that can be useful in ensuring that no relevant factors are overlooked.

(c) Influence Quantification. Lastly, the results of this study should be quantified. The design of the reported study makes quantification of the importance of factors and their influence strengths impossible. However, such quantification is necessary to prioritise which factors to focus on first, both for assessment and for improvement. We intend to design and execute such a study as future work to this study.

We can use structural equation modelling methods (Weiber and Mühlhaus 2021) to quantify the factors and their influences because we already have a structure model. We plan to use an online survey to query users of MTLs from research and industry about the amount they use different language features, their perception of qualitative properties of their transformations and demographic data surrounding use-case, skills & experience and used languages. The responses are used as input for universal structure modelling (USM) (Buckler and Hennig-Thurau 2008) based on the structural equation model developed from the interview responses.

USM is used to estimate the influence and moderation weights of all variables within the structure model. We can therefore use it to produce quantified data on the influence and moderation effects of identified factors.

We are confident that the approach of using a survey to quantify interview results, can complement the current results, because several of the authors have had positive experiences applying it (Liebel et al. 2018; Juhnke et al. 2020).

7.1.2 Improving MTL Capabilities

To improve current model transformation languages the criticisms articulated by interviewees can be used as starting points for enhancements and innovation. There are several aspects that are considered to be problematic by our interviewees.

(d) Improve reuse mechanism adoption. Reuse mechanisms in model transformation languages are one aspect where interviewees saw potential for improvement (see Section 5.2.10). Languages that do not currently possess mature reuse mechanisms can adopt them to become more usable. For the adoption of mature reuse mechanisms in MTLs we see the languages developers as responsible.

(e) Reuse mechanism innovation. Innovation towards transformation specific reuse mechanisms, as has been requested by some participants (see Section 5.2.10), should also be advanced. This topic was discussed at length during the interviews on the statement *"Having written several transformations, we have identified that current MTLs are too low a level of abstraction for succinctly expressing transformations between DSLs, because they demonstrate several recurring patterns that have to be reimplemented each time."* in Question Set 3.

Interviewees pointed out a need for reuse mechanisms that allow transformations to adapt to differing inputs and outputs. It would be conceivable to define transformation rules, or parts of them, independently of concrete model types, similar to generics in GPLs. This would allow development of generic transformation 'templates' of common transformation patterns. One pattern, for example, could be finding and manipulating specific model structures, like cliques, independent of the *concrete* model elements involved. Such templates can then be reused and adapted in all transformations where the pattern is required.

We believe, that innovating such new transformation specific reuse mechanisms is a community wide effort that needs to be taken on in order to make them more widely usable.

(f) Improving MDSE education. The *Learnability* of MTLs has also been a point of criticism. We believe, that more effort needs to be put into the transfer of knowledge for MDSE and its techniques like model transformations and MTLs. This believe is supported by the findings of Hutchinson et al. (2011). They also identified the lack of MDSE knowledge as a limiting factor for the adoption of the approach.

People need to come into contact with the principles earlier so that the inhibition threshold to apply them is lower. This was also remarked by interviewees when discussing the *Learnability* (see Section 5.2.11). More focus needs to be given to modelling and modelling techniques in software engineering courses. This is especially important since the skill of users has been said to be a largely impactful factor upon which many of the advantages from other MTL capabilities rely. Furthermore, there exist studies such as the one by Dieste et al. (2017), which detected a connection between the experience of developers with a language and their productivity as well as the code quality of the resulting programs.

To achieve this, we believe, that the researchers from the community, in their role as higher education teachers and university staff, need to become active. They should advocate for teaching the concepts of MDSE and the advantages/disadvantages in undergraduate studies in computer science study programmes. This view is shared by Samiee et al. (2018). Particularly, it should be taught that models can be used for more than documentation purposes, e.g., code generation, simulations early in the development cycle, test case

generation. These other uses are widely and successfully employed in the domain of cyber-physical systems according to Bucchiarone et al. (2021). Hence, it might be beneficial to include industrial modelling tools like Matlab/Simulink/Stateflow from this domain in addition to standard UML tools in undergraduate courses. Furthermore, we successfully used simulation frameworks for autonomous cars, like Carla (Dosovitskiy et al. 2017), in the past as targets for student projects when teaching courses on the development of modeling languages and model transformations. For example, the students devised a state machine language and code generator targeting the simulation framework to develop an automatic parking functionality. Model transformations were developed to flatten hierarchical state machines to non-hierarchical state machines prior to code generation.

(g) Increase knowledge retention. It is also difficult to get to grips with the subject matter in general, as information on it is much harder to obtain than on general purpose programming (see Section 5.3.7). This starts with the fact that, we found websites on MTLs to often be outdated or unappealing and lack good tutorials and comprehensible documentation. These points need to be fixed, by the language developers, to provide potential users with better resources to combat the perceived steepness of the learning curve. More active community involvement is also conceivable here. Users of MTLs could invest time in creating documentation and keeping it up-to-date. The possibility of this working and producing good results can be seen in examples such as the arch-linux wiki[4].

(h) Improve community outwards presentation. The model transformation community is small. In our opinion this leads to less innovation and poses the danger of entrenched practices. The problem is not limited to small communities as seen by, for example, the risk averse movie industry or low innovation automotive industry. An improved outwards presentation of the technology of model transformations can help alleviate the problem of limited human resources. The current hype surrounding low-code-platforms can be used to inspire young and aspiring researchers to contribute to its underlying concepts such as model transformations.

(i) Improve industry outreach and cooperation. We think it is also paramount to pursue industry cooperation to gauge industrial needs in order to facilitate more industrial adoption of MTLs. Here ambitious studies are required that attempt to provide the community with clear requirements specific domains of industry have for MDE and transformation languages, as well as to show for which domains application is reasonable at all. There exist some field studies by Staron (2006), Mohagheghi and Dehlen (2008), and Mohagheghi et al. (2013) but they are far and in between and do not focus on the transformation languages involved. The research community can attempt to organize solutions for these requirements based on such field study and industry research. However, for such industry cooperation to be possible, a focused community outreach is required. There are notable advancements in this direction e.g. MDENet[5], but they are still in their infancy and require more involvement by the research community.

(j) Provide *representative* model transformation languages. To provide reasonable evidence that model transformation languages can be competitive against GPLs there also

[4] wiki.archlinux.org

[5] community.mde-network.org

needs to be heavy focus on providing less prototypical and more pragmatic and useable transformation languages (see Section 6.2). To that end only a few selected languages should be attempted to be made production ready, potentially through further industry cooperation. MTLs could be integrated into commercial modelling tools in order to be able to process models programmatically in the tool.

Alternatively, few modern standardised MTLs could be promoted by the community. Since such a decision has far-reaching effects, a central, community wide respected body is needed. The OMG could possibly take action for this as they are already deciding on community impacting standards.

The QVT standard was an ambitious push in this direction. However, we believe that the initiative needs a fresh take, given the findings of the last 20 years of research. This idea is supported by several interviewees who considered QVT to be bloated and outdated. Especially in the areas of bidirectional and incremental transformations we see huge potential. Furthermore, relying more on declarative approaches for defining uni-directional transformations should also be considered. This trend can also be observed in the field of GPLs with the introduction of more and more functional concepts into them.

Innovation in prototypical languages should then be thoroughly evaluated for its usefulness before adoption into one of the flagship languages. It is not the task of research to produce industry ready languages, but setting up the environment and using these languages should not be more complicated than for any general purpose programming language.

(k) Research legacy integration. The integration of MTLs into existing legacy systems has been remarked as a huge entry barrier for industry adoption (see Sections 5.2.1 and 5.3.4). We believe this stems from a lack of techniques that facilitate gradual integration of modelling technology into existing systems and infrastructure. This is highlighted by the fact that basic literature such as that by Brambilla et al. (2017) does not contain any suggestions to this end. To combat this, we propose a dedicated branch of MDE research focused on developing tools and processes to integrate model driven techniques into legacy systems.

We envision distinct guidelines and processes on how to integrate transformations and transformation concepts into existing systems. There should be terms of reference as to which types of system components lend themselves to the use of model transformations. Furthermore, descriptions of which transformations and which transformation languages are suitable for which type of use case are also required. Having such guides can reduce the barrier of entry, because they provide a clear course of action when trying to (gradually) adopt the paradigm.

This also includes accessible GPL bindings for applying model transformation concepts. They can be used to gradually replace system components that can benefit from the use of transformations. This can be done without the overhead of integrating a new language and intermediate models. One example for this is DresdenOCL, a OCL dialect that can be used on Java code (Demuth and Wilke 2009).

7.2 Steps Towards Solving the Tooling Problem

From our interviews, we have to conclude that the biggest weak point of model transformation languages is their *Tool Support*.

The two biggest tooling gaps that we were able to identify are:

(I) many necessary tools do not exist

(II) existing tools lack user-friendliness and are not compatible with each other

We hope that our work can be a starting point in counteracting these drawbacks.

(l) Provide essential tooling. In our view, tooling of flagship model transformation languages needs to be extended to include all the essential tools mentioned in the interviews to make MTLs production and industry ready. This includes useable *Editors*, *Debuggers* and *Validation* or *Analysis* tools. At best all such tools for a language should be useable within one IDE. One way language developers can help with this task is by implementing the *Language Server Protocol* (LSP) (Microsoft) or its graphical counterpart GLSP (Eclipse Foundation) for their MTL. This would greatly improve the ability of tool developers to create and distribute tooling.

(m) Develop transformation specific debugging. As mentioned by our interviewees, for debuggers there is a need for model transformation specific techniques. Troya et al. (2022) showed that there are numerous advances in this area like those by Wimmer et al. (2009), Hibberd et al. (2007), and Ege and Tichy (2019) but none of them have led to well rounded debuggers yet. Further effort by researchers active in this area is therefore required. They should strive to develop their approaches to a point where they can be productively used to demonstrate their usefulness for a productive transformation development.

(n) Improve tool usability. Most importantly, a lot of effort needs to be put into improving the usability of MTL tools. Our interviews have shown, that unusable tools are the most off putting factor that hampers wider adoption. To combat this, we believe usability studies to be essential. Studies to identify usability issues in the likeness of what is proposed by Pietron et al. (2018) can be used to gain insights into where problems originate from and how to improve them. Such studies have already been successfully utilised for other MDE related tooling (Stegmaier et al. 2019). We therefore need more researchers from the community to get involved in designing and conducting usability studies for tooling surrounding MTLs.

We think the results of usability studies can also provide useful lessons learned for tool developers to make tools more usable from the beginning. The overall goal must be to find out what needs to be changed or improved in MTL tools to make their adoption significant. Industrial efforts to provide proper tool support can then be based on these results and the existing, usable, tools. This adoption is necessary because, in our view, the human resources required for providing adequate long-term support for the tools can only be provided by commercially operating companies. Such long term support is necessary so that model transformation languages, and their accompanying tools, can gain a foothold in the fast-moving industrial world. The industrialisation of MTL tooling was also proposed during an open community discussion detailed by Burgueño et al. (2019).

The goal should be to provide well rounded, all-in-one solutions that integrate all necessary tooling in one place, to make development as seamless as possible. The appropriateness of this has been shown by Jonkers et al. (2006).

(o) Limit-test internal MTLs. A different approach that should be further explored is the attempt to thoroughly embed an internal model transformation language in a main stream GPL as done by Hinkel and Goldschmidt (2019). The advantage of this approach is the ability to inherit tooling of the host language (Hinkel and Goldschmidt 2019) and it allows general purpose developers to apply their rich pool of experience. However, there are some drawbacks to this approach, as discussed in Section 5. The amount of tooling that can be

properly integrated is limited and it is more difficult to develop transformation specific tooling for internal languages as it is hard to extract the required information from the code. For this reason, we think, the required tools should be known at design time and the language has to be designed to expose all the required information while not imposing this as an additional burden on developers. Researchers that plan to develop an internal model transformation language should therefore thoroughly asses the tool requirements for the use case for which they intend to develop their language.

8 Threats to Validity

Our interview study was carefully designed, and followed reputable guidelines for preparation, conduction and analysis. Nonetheless there are some threats to validity that need to be discussed to provide a complete picture of our study and its results.

8.1 Internal Validity

Internal validity describes the extent to which a casual conclusion based on the study is warranted. The validity is threatened by manual errors and biases of the involved researchers throughout the study process.

Errors could have been introduced during the transcription phase and during the analysis of the data since both steps were conducted by a single author at a time.

To prevent transcription errors, all transcripts were re-examined after completion to ensure consistency between the transcripts and audio recordings.

To minimize possible confirmation biases introduced during analysis and categorisation of interviewee statements, random samples were checked by other authors to find possible discrepancies between the authors assessments on statements. In cases where such discrepancies were encountered, thorough discussions between all authors were conducted to find a consensus that was then applied to all transcripts containing similar considerations.

Lastly there is the potential of misinterpretation of interviewees responses during analysis. While we carefully stuck to interpret statements literally during coding, there are words and phrases that have overloaded meanings. During the interviews, it would always be necessary to ask exactly what meaning interviewees used, but this was not always possible. Therefore the threat could not be mitigated completely as contextual information was required to interpret interviewees responses in some cases.

8.2 External Validity

External validity describes the extent to which the results of a study can be generalised. In our interview study this validity is threatened by our interview participant assortment ,which is a result of our sampling and selection method.

We utilise convenience sampling interviewing any and all people that respond to our emails. This can limit how representative the final group of interviewees is of the target population. The issue here is that we do not know much about the makeup of the target population. It is therefore difficult to assess how much the group of participants deviates from a representative set.

Using research publications as the starting point for participant selection also introduces a bias towards users from research. This can be clearly seen in Fig. 6. There is an apparent lack of participants from industry which limits the applicability of our results to industrial

cases. This threat is somewhat mitigated by the fact that half of all participants do have at least some contact with industry, either through research projects in conjunction with industry or by having worked in industry.

Another threat to external validity relates to model to text (M2T) transformations. Only a few of our participants stated to have experience in applying M2T transformations. This is a result of how the initial set of potential participants was constructed. The search terms used in the SLR miss terms that relate to M2T such as 'code generation' or 'model to text'. This limitation was opted into to avoid having to differentiate between the two transformation approaches during analysis. Moreover, the consensus during discussions was that we were talking about model to model transformations. As such, our results can not be applied to the field of model to text languages.

Lastly, there is the threat of participation bias. Participants may disproportionately posses a trait that reduces the generalisability of their responses. People that view model transformation languages positively might be more inclined to participate than critics. We can not preclude this threat, but, the amount of critique we were able to elicit from the interviews suggests the effects from this bias to be weak. Other impacts of this bias are discussed in Section 8.4.

8.3 Construct Validity

Construct validity describes the extent to which the right method was applied to find answers for the research question. This validity is threatened by an inappropriate method that allows for errors.

Prior to conducting our research much work went into designing a proper framework to use. Here we relied on reputable existing guidelines for both the interview and analysis parts of this work. We used open ended questions to facilitate an open space for participants to bring forth any and all their opinions and considerations for the topic at hand. The statements used as guidance can however present a potential threat since their wording could introduce an unconscious bias in our interviewees. To combat this we selected broad statements as well as used both a negative and a positive statement for each discussed property. However, there is a chance that these measures were not fully sufficient.

Lastly, it can not be excluded that some relevant factors have not been raised during our interviews. We have interviewed a large number of people, but this threat cannot be overcome because of the study design and the open nature of our research question.

8.4 Conclusion Validity

Conclusion validity describes the extent to which our results stem from the investigated variables and are reproducible. Here, the biases of our participants represent the biggest threat.

It is safe to assume that people who do research on a subject are more likely to see it in a positive light and less likely to find anything negative about it. As such there is the possibility that too little negative impact factors were considered and presented. However, we found that the people we interviewed were also able to deal with the topic in a very critical way. We therefore conclude that the statements may have been somewhat more positively loaded, but that the results themselves are meaningful.

9 Related Work

To the best of our knowledge, there exists no other interview study that focuses on influence factors on the advantages and disadvantages of model transformation languages. Nonetheless there exist several works that can be related to our study. The related work is divided into empirical studies on model transformation languages, empirical studies on model transformations in general and interview studies on MDE.

9.1 Empirical Studies on Model Transformation Languages

A structured literature review we conducted (Götz et al. 2021) forms the basis for the work presented in this paper. The goal of the reported literature review was to extract and categorize claims about the advantages and disadvantages of model transformation languages as well as to learn and report on the current state of evaluation thereof. The authors searched over 4000 publications to extract a total of 58 publications that directly claim properties of model transformation languages. In total the authors found 137 claims and categorized them into 15 properties. From their work the authors conclude that while many advantages and disadvantages are claimed little to no studies have been executed to verify them. They also point out a general lack of context and background information on the claimed properties that hinders evaluation and prompts scepticism.

Burgueño et al. report on a online survey, as well as a subsequent open discussion, at the 12th edition of the International Conference on Model Transformations (ICMT'2019) about the future of model transformation languages (Burgueño et al. 2019). Their goal for the survey was to identify reasons as to why developers decided for or against the use of model transformation languages and what their opinion on the future of these languages was. At ICMT'2019 where the results of the survey were presented they then moderated an open discussion on the same topic. The results of the study indicate that MTLs have fallen in popularity compared to at the beginning of the decade which they attribute to technical issues, tooling issues, social issues and the fact that general purpose languages have assimilated ideas from MTLs making GPLs a more viable option for defining model transformations. While their methodology differed from our interview study, the results of both studies support each other. However the results of our study are more detailed and provide a larger body of background knowledge that is relevant for future studies on the subject.

The notion of general purpose programming languages as alternatives to MTLs for writing model transformations has been explored by Hebig et al. (2018) and by us (Götz et al. 2021). Hebig et al. (2018) report on a controlled experiment where student participants had to complete three tasks involved in the development of model transformations. One task was to comprehend an existing transformation, one task involved modifying an existing transformation and one task required the participants to develop a transformation from scratch. The authors compare how the use of ATL, QVT-O and the general purpose language Xtend affect the outcome of the three tasks. Their results show no clear evidence of an advantage when using a MTL compared to a GPL but concede the narrow conditions under which the observation was made. The study provides a rare example of empirical evaluation of MTLs of which we suggest that more be made. The narrow conditions the authors struggled with could be alleviated by follow-up studies that draw from our results for defining their boundaries.

In a recent study by us (Götz et al. 2021) we put the value of model transformation language into a historical perspective and drew from the preliminary results of the interview study for the study setup. We compare the complexity of a set of 10 model transformations

written in ATL with their counterparts written in Java SE5, which was current around 2006 when ATL was first introduced, and Java SE14. The Java transformations were translated from the ATL modules based on a predefined translation schema. The findings support the assumptions from Burgueño et al. (2019) in part. While we found that newer Java features such as Streams allow for a significant reduction in cyclomatic complexity and lines of code the relative amount of complexity of aspects that ATL can hide stays the same between the two Java versions.

Gerpheide et al. use an exploratory study with expert interviews, a literature review and introspection to formalize a quality model for the QVTo model transformation standard by the OMG (Gerpheide et al. 2016). They validate their quality model using a survey and afterwards use the quality model to identify tool support need of transformation developers. In a final step the authors design and evaluate a code test coverage tool for QVTo. Their study is similar to ours in that they also relied on expert interviews for their goal. The end goal of the study however differs from ours as they used the interviews to design a quality model for QvTo while we used it to formulate influence factors on quality attributes of model transformation languages

Lastly there are two study templates for evaluating model transformation languages which have yet to be used for executing actual studies. Kramer et al. present a template for a controlled experiment to evaluate the comprehensibility of model transformation languages (Kramer et al. 2016). Their approach suggests the use of a paper-based questionnaire to let participants prove their ability to understand what a transformation code snippet does. The influence of the language in which the code is written on comprehension speed and quality is then measured by comparing the average number of correct answers and the average time spent to fill out the questionnaires. Strüber and Anjorin propose a controlled experiment for comparing the benefits and drawbacks of the reusability mechanisms *rule refinement* and *variability-based rules* (Strüber and Anjorin 2016). They suggest that the value of the reusability of an approach can be measured by looking at the comprehensibility of the two mechanisms as well their changeability, which is measured through bug-fixing and modification tasks. The results of studies executed based on both study templates could draw from our results for their final design and would provide valuable empirical data, a gap we identified in this and the preceding literature review.

9.2 Empirical Studies on Model Transformations

Tehrani et al. executed an interview based study on requirements engineering for model transformation development (Tehrani et al. 2016). Their goal was to identify and understand the contexts and manner in which model transformations are applied as well as how requirements for them are established. To this end they interviewed 5 industry experts. From the interviews the authors found that out of 7 transformation projects only a single project was developed in an already existing project while all other projects were created from scratch. Their findings are relevant to our work since participants in our study agreed that it is hard to integrate MTLs in existing infrastructures. Whether the fact that MTLs are hard to integrate was an influence factor for the projects considered in the interview study by them is however not clear.

Groner et al. utilize an exploratory mixed method study consisting of a survey and subsequent interviews with a selection of the survey participants to try and evaluate how developers deal with performance issues in their model transformations (Groner et al. 2020). They also asses the causes and solutions that developers experienced. The survey results show that over half of all developers have experienced performance issues in their transformations. While the interviews allowed the authors to identify and categorize performance

causes and solutions into 3 categories: *Engine related*, *Transformation definition related* and *Model related*. From the interviews they were also able to identify that tools such as useable profilers and static analyses would help developers in managing performance issues. The results of their study highlight that some of the factors identified by us are also relevant for other MTL properties not directly investigated in our study.

9.3 Interview Studies on Model Driven Software Engineering

There are numerous publications and several groups of researchers that have carried out large scale, in-depth empirical studies on model driven engineering as a whole. We focus on a selection of those that have relation to our study in terms of findings.

Whittle, Hutchinson, Rouncefield et al. used questionnaires (Hutchinson et al. 2011; Hutchinson et al. 2014) and interviews (Whittle et al. 2013; Hutchinson et al. 2011; Hutchinson et al. 2011; Hutchinson et al. 2014) to elicit positive and negative consequences of the usage of MDE in industrial settings. Apart from technical factors related to tooling they also found organisational and social factors that impact the adoption and efficacy of MDE. Several of their findings for MDE in general coincide with results from our study. Related to tooling they too found the factors of *Interoperability*, *Maturity* and *Usability* to be influential. Moreover, on the organisational side, the small amount of people that are knowledgable in MDE techniques and the problem of integrating into existing infrastructure are also results Whittle et al. found. Lastly, developers being more interested in using techniques that help build their CV was identified by them as a limiting factor too.

Staron analyses data collected from a case study of MDE adoption at two companies where one company withdrew from adopting MDE while the other was in the process of adoption (Staron 2006). Their findings suggest that legacy code was a main influence factor on whether a cost efficient MDE adoption was possible. This observation is consistent with our findings that integrating MTLs into existing infrastructures has a negative impact on the *Productivity* that can be achieved with MTLs.

The research group surrounding Mohagheghi also carried out multiple empirical studies on MDE, focusing on factors for and consequences of adoption thereof. They use surveys and interviews at several companies (Mohagheghi et al. 2013; Mohagheghi et al. 2013) as well as a literature review (Mohagheghi and Dehlen 2008) for this purpose. In addition to mature tooling, factors identified by the authors are usefulness, ease of use and compatibility with existing tools. Similar to statements by our interviewees, they also found that MDE is seen as a long term investment. It is not well suited for single projects.

Lastly, Akdur et al. report on a large online survey of people from the domain of embedded systems industry (Akdur et al. 2018). They too found tools surrounding MDE to be a major factor. Another interesting finding by them was that UML models are by far the most commonly used models. This is of relevance to our results since one of our interviewees pointed out, that the makeup of some UML models can have detrimental effects on the usefulness of MTLs.

The results of all presented research groups show, that many of the factors we identified for MTLs also apply to MDE in general which provides additional confidence in our results and shows that advancements in these areas would have a high impact.

10 Conclusion

There are many claims about the advantages and disadvantages of model transformation languages. In this paper, we presented and argued the detailed factors that play a role for

such claims. Based on interviews with 56 participants from research and industry we present a **structure model** of relevant factors for the *Ease of writing*, *Expressiveness*, *Comprehensibility*, *Tool Support*, *Productivity*, *Reuse* and *Maintainability* of model transformation languages. For each factor we detail which properties they influence and **how** they influence them. We have identified two types of factors. There are factors that have a **direct impact** on said properties, e.g. different capabilities of model transformation languages like automatic trace handling. And there are factors that define a **context** whose characteristics *moderate* the the impact of the former factors, e.g. the *Skills* of developers.

Based on the interview results we suggest a number of tangible actions that need to be taken in order to convey the viability of model transformation languages and MDSE. For one, empirical studies need to be executed to provide proper substantiation to claimed properties. We also need to see more innovation for transformation specific reuse, legacy integration and need to improve outreach and presentation to both industry and academia. Lastly, efforts must be made to improve tool support and especially tool usability for MTLs.

For all of the suggested actions, our results can provide detailed data to draw from.

Appendix A: Interview Questions

Demographic Questions

- In what context have you used model transformation languages? Research, industrial projects or other?
- How much experience do you have in using model transformation languages? Rough estimate in years is sufficient.
- What model transformation languages have you used to date?

Question Set 1

Ease of Writing

The use of MTLs increases the ease of writing model transformations.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Model transformation languages ease development efforts by offering succinct syntax to query from and map model elements between different modelling domains.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Model transformation languages require specific skills to be able to write model transformations.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Comprehensibility

The use of MTLs increases the comprehensibility of model transformations.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Model transformation languages incorporate high-level abstractions that make them more understandable than GPLs.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Most MTLs lack convenient facilities for understanding the transformation logic.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Question Set 2

Tool Support

There is sufficient tool support for the use of MTLs for writing model transformations.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Tool support for external transformation languages is potentially more powerful than for internal MTL or GPL because it can be tailored to the DSL.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Model transformation languages lack tool support.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Productivity

The use of MTLs increases the productivity of writing model transformations.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Model transformation languages, being DSLs, improve the productivity.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?

- What is the reasoning behind your answer?

Productivity of GPL development might be higher since expert users for GPLs are easier to hire.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Question Set 3

Reuseability & Maintainability

The use of MTLs increases the reusability and maintainability of model transformations.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Bidirectional model transformations have an advantage in maintainability.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Model transformation languages lack sophisticated reuse mechanisms.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Expressiveness

The use of MTLs increases the expressiveness of model transformations.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Model transformation languages hide transformation complexity and burden from the user.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Having written several transformations we have identified that current MTLs are too low a level of abstraction for succinctly expressing transformations between DSLs because they demonstrate several recurring patterns that have to be reimplemented each time.

- On a scale of 1 to 5, with 1 being strongly disagree and 5 being strongly agree, how would you rate your agreement with the statement?
- What is the reasoning behind your answer?

Appendix B: Mail Templates

Mail Template

Dear ${Author Name},

I'm a PhD student with Matthias Tichy at Ulm University. We recently conducted an SLR about the advantages and disadvantages of model transformation languages as claimed in literature. Our results have been published in the software and systems modelling journal here http://dx.doi.org/10.1007/s10270-020-00815-4. One of our main takeaways from the study was that a large portion of claims about model transformation languages is never substantiated. One main reason for this, we believe, is implicit knowledge authors tend to omit for different reasons.

Since you are an author of one of the publications we considered during our SLR it would be great to talk to you about your experiences and stance with regard to model transformation languages and the claims we extracted from literature. We would need max. 30 minutes of your time. The interview would be conducted by me via an online conferencing system.

In order to organize the interview dates I would like to ask you to chose a suitable date, under the following link https://terminplaner4.dfn.de/F1mIEEwSSkTwh8XA. Please note that the times are given in UTC. The password for the poll is "claims". Your response will not be visible to anyone other than myself. If none of the dates is suitable for you, you are welcome to contact me to find another date for the interview.

Before your interview I would like to ask you to agree to the data protection agreement under the following link https://pmx.informatik.uni-ulm.de/limesurvey/index.php/924713?lang=en. I have also attached a copy of how we handle the interview data to this mail.

Best regards
Stefan Götz

Reminder Mail Template

Dear ${Author Name},

If you already filled out our organization poll please ignore this mail.

I wanted to remind you to maybe take part in our interview study about the implicit knowledge of users with regards to advantages and disadvantages of model transformation languages. It would be great to talk to you about your experiences and stance with regard to model transformation languages and the claims we extracted from literature. We would need max. 30 minutes of your time.

In order to organize the interview dates I would like to ask you to chose a suitable date, under the following link https://terminplaner4.dfn.de/F1mIEEwSSkTwh8XA. Please note that the times are given in UTC. Please also note that you need to press the SAVE button at the right hand side of the poll. The password for the poll is "claims". Your response will not be visible to anyone other than myself. If none of the dates is suitable for you, you are welcome to contact me to find another date for the interview.

Before your interview I would like to ask you to agree to the data protection agreement under the following link https://pmx.informatik.uni-ulm.de/limesurvey/index.php/924713?lang=en. I have also attached a copy of how we handle the interview data to this mail.

Best regards
Stefan Götz

Appendix C: Demographics

See Table 4.

Table 4 Overview over the interviewee demographic data

PID	Background	Experience in years	Language types used for writing transformations
P1	Research	> 10	GPLs
P2	Research	10-15	dedicated MTLs
P3	Research	8	dedicated MTLs
P4	Research	7	dedicated MTLs & internal MTLs
P5	Research	> 5	dedicated MTLs & GPLs
P6	Research & Industry Projects	13	dedicated MTLs & GPLs
P7	Research & Industry Projects	10	dedicated MTLs & GPLs
P8	Research & Industry Projects	18	dedicated MTLs & GPLs
P9	Industry	20	dedicated MTLs
P10	Research	4	dedicated MTLs & GPLs
P11	Research	5-6	dedicated MTLs
P12	Research & Industry Projects	8	dedicated MTLs
P13	Industry with History in Research	6	dedicated MTLs & internal MTLs
P14	Research & Industry Projects	15	dedicated MTLs & internal MTLs & GPLs
P15	Research & Industry Projects	5	dedicated MTLs & GPLs
P16	Research	7	dedicated MTLs & GPLs
P17	Research & Industry Projects	18	dedicated MTLs & GPLs
P18	Research & Industry Projects	10	dedicated MTLs & GPLs
P19	Research	7	dedicated MTLs
P20	Research & Industry Projects	3	dedicated MTLs & GPLs
P21	Research & Industry Projects	15	dedicated MTLs & GPLs
P22	Research & Industry Projects	8	dedicated MTLs & GPLs
P23	Research	13	dedicated MTLs
P24	Research & Industry Projects	15	dedicated MTLs & GPLs
P25	Research	8	dedicated MTLs
P26	Industry	> 10	dedicated MTLs
P27	Industry with History in Research	10-12	dedicated MTLs & GPLs
P28	Research	15	dedicated MTLs & GPLs
P29	Research & Industry Projects	12	dedicated MTLs
P30	Research & Industry Projects	17	dedicated MTLs & GPLs
P31	Research	8	dedicated MTLs
P32	Research & Industry Projects	15	dedicated MTLs & GPLs
P33	Research	5-6	dedicated MTLs & GPLs
P34	Research	5-6	GPLs
P35	Research	10	dedicated MTLs
P36	Research	10	dedicated MTLs & GPLs

Table 4 (continued)

PID	Background	Experience in years	Language types used for writing transformations
P37	Research & Industry Projects	10-11	dedicated MTLs
P38	Research	4-5	dedicated MTLs
P39	Industry	28	dedicated MTLs & GPLs
P40	Research	9	dedicated MTLs
P41	Research	7-8	dedicated MTLs
P42	Industry with History in Research	13	dedicated MTLs & internal MTLs & GPLs
P43	Research & Industry Projects	8-10	dedicated MTLs
P44	Research & Industry Projects	10	dedicated MTLs
P45	Research	1-2	dedicated MTLs
P46	Research & Industry Projects	9	dedicated MTLs
P47	Research	4	dedicated MTLs
P48	Research	7-8	dedicated MTLs & internal MTLs
P49	Research & Industry Projects	10	dedicated MTLs & GPLs
P50	Research	20	dedicated MTLs
P51	Research & Industry Projects	3	dedicated MTLs
P52	Research	13-14	dedicated MTLs
P53	Research	12	dedicated MTLs & GPLs
P54	Research	7	dedicated MTLs
P55	Research & Industry Projects	16	dedicated MTLs & GPLs
P56	Research	16	dedicated MTLs

Appendix D: Data Privacy Agreement

General information, declaration of consent

General information about the interviews about claims about model transformation languages of the Institute for Software Engineering and Programming Languages of Ulm University.

At the Institute for Software Engineering and Programming Languages of Ulm University model transformation languages are being examined. This includes claims about the advantages and disadvantages and evidence thereof. Within one work package of the doctoral thesis of Stefan Götz, researchers and practitioners are being surveyed about their opinions on certain claims.

The goal of this work package is to gain a deeper understanding of the reasoning people use for believing certain claims about model transformation languages.

1 Procedure

- If you decide to participate in our interview study, please fill out the poll at https://terminplaner4.dfn.de/F1mIEEwSSkTwh8XA so we can set up a date for the interview.
- We will contact you about one week in advance of the chosen date to arrange the interview.
- The interview will take place using an online conferencing tool hosted at Ulm University or via Skype depending on your preference.

- Please fill out the consent form at https://pmx.informatik.uni-ulm.de/limesurvey/index.php/924713 before the interview.
- At the beginning of the interview we will ask you questions about your experience level with regards to model transformation languages such as the languages you have used.
- During the interview we will show you different claims from literature about model transformation languages for which we would like you to tell us if you agree with them or not and your reasoning behind the decision.
- An audio recording of the interview will be made if you consent to it.
- The audio recordings will be transcribed after the interview and deleted as soon as transcribing has been completed.

2 Conditions of participation:

- You have some for of experience with model transformation languages.

3 Handling the data in the research project

1. Your interview will be recorded on audio and notes will be taken. The procedure described in 2-5 is followed for anonymizing your interview.
2. The non-anonymous raw data (first name, last name, e-mail address, notes and the audio recording of your interview) is only shared between the project partners (Stefan Götz, Yves Haas and Matthias Tichy from Ulm University) for transcribing and analyzing the answers.
3. We will anonymize your interview by transcribing it and delete the information about your first name, last name, e-mail and the audio recording of your interview as soon as possible and not later than 30 October 2020. Which means that individuals cannot be deduced from their interview.
4. For scientific publication, anonymized answers to this interview (transcribed interview and notes) will be further processed, e.g., coded or aggregated. Special risks for your person are not apparent with the processed results because individual persons cannot be inferred.
5. We strongly believe in open data to allow replication of our results as well as enabling further research. The anonymized answers (unprocessed and processed) to this interview (transcribed interview and notes) will be made available online to the public to accompany publications and stored in open data repositories. Please note that such published data can no longer be completely deleted and can be accessed and used by any person. Special risks for your person are not apparent with the anonymized answers, because individual persons cannot be inferred.

Contact person

If you have any questions, concerns or doubts, please do not hesitate to contact us:

Stefan Götz Ulm University Institute of Software Engineering and Programming Languages 89081 Ulm E-Mail: stefan.goetz@uni-ulm.de

Declaration of Consent

(Name, Surname of the participating person)

I have read the general information on the interview study and agree to participate in the research project and the associated data processing.

(You can also give the following consent:)

- ☐ I have read the general information on the interview, which is conducted as part of a research project. I consent to participate in the interview and I consent to the related data processing as described in 1-4.
- ☐ I hereby consent to the publication of my anonymized answers (unprocessed and processed) to this interview (transcribed interview, notes and linked questionnaire) as described in 5.

I am aware that the consents are voluntary and can be refused without disadvantages (even individually) or revoked at any time without giving reasons. I am aware that in case of revocation, the legality of the processing carried out on the basis of the consent until revocation is not affected. I understand that I can simply contact the contact person named in the information for a revocation and that no disadvantages arise from the refusal of consent or its revocation.

I was informed and provided with the information on the collection of personal data during the interview study. I have also received a copy of this consent form.

Appendix E: Quotations

Table 5 Selection of quotations from interview participants for specific factors

Factor	QID	PID	Quotation
GPL Capabilities	$Q_{gpl}1$	**P42**	*"[General purpose languages are] very good at constructing objects and filling in their fields [...] and computing 'simple' expressions."*
	$Q_{gpl}2$	**P14**	*"I think that in the end you have more tools for development. And I feel more productive."*
	$Q_{gpl}3$	**P8**	*"[...] when you reach the maintenance phase, maybe the [original] developers are gone. And you have an [MTL] program that might be more difficult to understand for [new] developers"*
Domain Focus	$Q_{df}1$	**P6**	*"What is better by using MTLs instead of GPLs is the fact that you are on the same abstraction level of the modelling language. You are basically treating apples with apples."*
	$Q_{df}2$	**P19**	*"[...] you are gonna cut away all those unneeded code and complexity and focus on your problem."*
	$Q_{df}3$	**P23**	*"Once you have things like rules and helpers and things like left hand side and right hand side and all these patterns then [it is] easier to create things like meta-rules to take rules from one version to another version [...]"*
	$Q_{df}4$	**P13**	*"To do this [tool support for analysing rule dependencies] [...] you have to resolve parameter dependencies and I immediately run into Turing completeness. And I don't have that with an external language [...]"*
	$Q_{df}5$	**P6**	*"They have existing infrastructure and people and everything that is based on established languages which is hard to change."*

Table 5 (continued)

Factor	QID	PID	Quotation
Bidirectionality	$Q_{bx}1$	**P42**	*"in a general purpose programming language you would have to add a bit of clutter, a bit of distraction, from the real heart of the matter"*
	$Q_{bx}2$	**P40**	*"So either you write your own program to create unidirectional transformations in both directions or you write both directions by hand and that has the disadvantage that if, in the future, something changes in the transformation, then you have to rework both directions"*
	$Q_{bx}3$	**P41**	"*[...] That makes it harder for them to see whether something is correct or not and to master the complexity of these transformations.*"
	$Q_{bx}4$	**P11**	*"And as soon as I am at bidirectional transformations and there is somehow a loss of information. [...] And then [the question is] how difficult it is to access e.g. context elements that I have already created and need again later, because I want to refer to them."*
Incrementality	$Q_{inc}1$	**P56**	*"Declarative MTLs may have different computation paradigms which may be unfamiliar for developers used to imperative languages"*
	$Q_{inc}2$	**P42**	*"[...] do not try to do it manually, because you will definitely have bugs,[...] because there will be some specific kind of change trajectory that you have missed, [...] this is a super hard problem."*
	$Q_{trc}2$	**P32**	*"You have to know what a trace is. [...] And at some point, at the latest when you do something more complex, you need this stuff. "*
	$Q_{trc}3$	**P31**	*"[...] a model transformation rule only [contains] the domain transformation, so which domain object of the source domain is mapped to an object and how the object is mapped to the target domain. And that is what someone who tries to understand the model transformation is trying to get [...] out of the source code."*
Automatic Traversal	$Q_{trv}1$	**P49**	*"That means abstracting away from the order of traversal and then also knowing in which context this thing came up, that is a bit of a double-edged sword for me, [...] it has the potential to mask serious errors."*
Pattern-Matching	$Q_{pm}1$	**P14**	*"[...] all the complexity of pattern matching is in the engine, but if you try to implement a mapping then all the complexity of keeping the traces you have to do that manually."*
Model Navigation	$Q_{nav}1$	**P41**	*"[...] you don't have to worry about the efficiency of the procedure, just figures out the optimal way of kind of traversing it. For me that is the biggest thing actually, they go and get me the data, if we can hide that from the user, that is great."*
	$Q_{nav}2$	**P11**	*"[...] I do not have to iterate over the model. I only say, I need this or that."*
Model Management	$Q_{man}1$	**P2**	*"[...] this technical level, how I access a model, [...]I get the elements out. That gets abstracted away."*
Reuse Mechanism	$Q_{rm}1$	**P51**	*"[...] we usually use object oriented programming languages and those already have some pretty strong tools for reusability in the appropriate contexts. So i think the bar here, that would we want model transformation languages to jump over, is to provide something more targeted towards modeling [...]"*

Table 5 (continued)

Factor	QID	PID	Quotation
Mappings	$Q_{map}1$	**P24**	"*They hide those dimensions that reflect how graph-wise it would be computationally complex to interpret the problem to transform one model into another*"
	$Q_{map}2$	**P25**	*"So it restricts you in the way you can work and that makes it easier because that is what you need to do."*
	$Q_{map}3$	**P55**	*"This means that you can write the rules independently of the execution sequence, you can define them more declaratively and, at least in my experience, you can still manage to define these rule blocks in a comprehensible way for large transformations."*
	$Q_{map}4$	**P5**	*"I mentioned language engineering because a lot of the transformation difficulties are understanding the syntactical and semantic differences between two domain specific languages."*
	$Q_{map}5$	**P30**	*"[...] Hidden mechanisms or built in mechanisms may be more difficult to understand [thus] learning the language may be a bit more difficult."*
	$Q_{map}6$	**P38**	*"[...] you have a formal correspondence between the two models. And if you can transform in both directions, then you can practically keep both models, between which you want to transform back and forth, synchronous."*
	$Q_{map}7$	**P16**	*"Whereas when you need to do some more elaborate business logic or when you need to hook some external services or other sources of information into your transformation then I am saying that MTLs can start to be a little bit of a limit"*
	$Q_{map}8$	**P3**	*"If I want to reuse this model transformation just changing 2 words in ATL [is enough], if I wanted to do the same in Java instead of changing something in 2 places I have to do it in 5 or 6."*
Tooling Awareness	$Q_{awa}1$	**P35**	*"And, I think, it is hard for new users to see, for example, what, which tool to use. Or which technology you should work with."*
Tool Creation Effort	$Q_{tce}1$	**P1**	*"I am keenly aware of the cost to being able to develop a good programming language, the cost of maintaining it and the cost of adding debuggers and refactoring engines. It is enormous."*
	$Q_{tce}2$	**P6**	*"But it is definitely easier and faster to build the tool support and it allows you to do more advanced stuff. You can play around with your domain specific concepts in a lot of different ways."*
Tool Learnability	$Q_{tle}1$	**P34**	*"Because when i started to work with model transformation languages and to hear about them, [...] I do not think that [...] there was like initial go-to documentation."*
Tool Usability	$Q_{use}1$	**P22**	*"Basically all the good aids you see in a Java environment should be there even better in a MTL tool because model transformation is so much more abstract and more relevant that you should be having tools that are again more abstract and more relevant."*
	$Q_{use}2$	**P48**	*"There are quite a few corner cases, which are often not quite fixed and especially the usability is often very bad."*
Tool Maturity	$Q_{mat}1$	**P23**	*"Because [MTLs] have been around for like 30 years. And other languages and frameworks, they are created in 2-3 years, and they are good to go. And MTLs have been around for so long. And I think its mostly because industry has not taken it in. And it's just a problem of manpower put into the languages."*
Validation Tooling	$Q_{val}1$	**P8**	*"For example I can not remember any tool that offers reasonable support for testing. In Java you have JUnit and other. In ATL there is nothing."*

Table 5 (continued)

Factor	QID	PID	Quotation
Traceability	$Q_{trc}1$	**P22**	*"So that is something you often have to do manually in a GPL. So you have to maintain the trace information yourself and kind of re-implement that."*
	$Q_{rm}2$	**P30**	*"[...] for ATL there are things like module superimposition, and other kinds, we have helper libraries."*
	$Q_{rm}3$	**P27**	*"[...] in the case of VIATRA one of the main goals of the pattern language we are using there is to allow reusing previously defined patterns. Basically any pattern can be included. So there is a lot of stuff you can do to reuse the element."*
Learnability	$Q_{ler}1$	**P23**	*"So the learning curve is pretty steep when trying to use MTLs. You need to learn a lot of stuff before you can use them properly."*
	$Q_{ler}2$	**P6**	*"You can take 10 Java developers and out of them probably 2 would understand what a MTL is. They don't have experience in modelling. Not because they are dumb, because they are not used to that."*
Debugging Tooling	$Q_{db}1$	**P51**	*"Well I think one of the other important points would be to [be] able to prove properties of transformations or check properties of transformations, [...] but we don't really have that for model transformations."*
Ecosystem	$Q_{eco}1$	**P49**	*"people from industry have a hard time when they are required to use multiple languages."*
	$Q_{eco}2$	**P49**	*"It is often on a technical level that the integration into the overall ecosystem of tools you have is not so great."*
	$Q_{eco}3$	**P31**	*"Something I see as a problem with some model transformation languages, which limit the applicability, is the coupling to Eclipse. This is what will cause us as a research community big problems some day [...]."*
Interoperability	$Q_{int}1$	**P36**	*"But the technologies, to combine them, it is difficult [...]"*
Language Skills	$Q_{skl}1$	**P1**	*"[...] this is the way you have to think in terms of formulating your problem"*
	$Q_{skl}2$	**P12**	*"And then you [need to] learn a language, the MTL."*
User Experience/ Knowledge	$Q_{exp}1$	**P21**	*"One of the reasons why Ada is virtually extinct is that developers preferred to have C++ on their CVs. Simply because there were more job postings with C++. And that develops a momentum of its own, which of course makes languages suffer. That also applies to DSLs."*
	$Q_{exp}2$	**P6**	*"Many MDSE courses are just given too late, when people are too acquainted with GPLs, and then its really hard for students to see the point of using models, modelling and MTLs, because it's comparable with languages and stuff they have already learned and worked with."*

Table 5 (continued)

Factor	QID	PID	Quotation
Involved (meta-) models	$Q_{mod}1$	**P28**	*"As soon as you venture into eGenericType there is a lot of pain to be had and there is poor documentation."*
I/O Semantic gap	$Q_{gap}1$	**P22**	*"[...] as soon as I wanted to do something a bit more complex, I have often found that I was not able to express what I wanted to do easily and I had to resort to advanced features of the language in order to achieve what I want to do."*
Size	$Q_{siz}1$	**P55**	*"The size is a good point. I would reduce that now to rules. But if I have several rules that then build on each other, then it will probably be easier with an MTL. Especially if you have a lot of dependencies between the rules."*

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s10664-022-10194-7.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Conflict of Interests The authors have no competing interests to declare that are relevant to the content of this article.

References

Akdur D, Garousi V, Demirörs O (2018) A survey on modeling and model-driven engineering practices in the embedded software industry. J Syst Architect 91:62–82. https://doi.org/10.1016/j.sysarc.2018.09.007. https://www.sciencedirect.com/science/article/pii/S1383762118302455

Anjorin A, Buchmann T, Westfechtel B (2017) The families to persons case. In: Transformation Tool Contest 2017, CEUR-WS, pp. 15–30

Arendt T, Biermann E, Jurack S, Krause C, Taentzer G (2010) Henshin: Advanced concepts and tools for in-place EMF model transformations. 10.1007/978-3-642-16145-2_9

Balogh A, Varró D (2006) Advanced model transformation language constructs in the VIATRA2 framework. In: Proceedings of the 2006 ACM Symposium on Applied Computing, SAC '06. 10.1145/1141277.1141575

Brambilla M, Cabot J, Wimmer M (2017) Model-driven software engineering in practice. Synthesis Lect Soft Eng 3(1):1–207

Brown AW, Conallen J, Tropeano D (2005) Introduction: Models, modeling, and model-driven architecture (MDA). In: Model-Driven Software Development, Springer, pp 1–16. 10.1007/3-540-28554-7_1

Bucchiarone A, Ciccozzi F, Lambers L, Pierantonio A, Tichy M, Tisi M, Wortmann A, Zaytsev V (2021) What is the future of modeling? IEEE Softw 38(2):119–127. https://doi.org/10.1109/MS.2020.3041522, https://doi.org/10.1109/MS.2020.3041522

Buckler F, Hennig-Thurau T (2008) Identifying hidden structures in marketings structural models through universal structure modeling. In: Marketing ZFP 30.JRM 2, pp. 47?66

Burgueño L, Cabot J, Gérard S (2019) The future of model transformation languages: An open community discussion. https://doi.org/10.5381/jot.2019.18.3.a7

Charmaz K (2014) Constructing grounded theory. Sage, ISBN: 9780857029140

Cuadrado JS, Molina JG, Tortosa MM (2006) Rubytl: a practical, extensible transformation language. 10.1007/11787044_13

Czarnecki K, Helsen S (2006) Feature-based survey of model transformation approaches. 10.1147/sj.453.0621

Demuth B, Wilke C (2009) Model and object verification by using dresden OCL. In: Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa. Citeseer, Russia, pp 687-690

Dieste O, Aranda AM, Uyaguari F, Turhan B, Tosun A, Fucci D, Oivo M, Juristo N (2017) Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study. https://doi.org/10.1007/s10664-016-9471-3

Dosovitskiy A, Ros G, Codevilla F, Lopez A, Koltun V (2017) CARLA: An open urban driving simulator. In: Proceedings of the 1st Annual Conference on Robot Learning, Proceedings of Machine Learning Research 78:1–16 Available from https://proceedings.mlr.press/v78/dosovitskiy17a.html

Eclipse Foundation Eclipse graphical language server platform (GLSP). https://www.eclipse.org/glsp/

Ege F, Tichy M (2019) A proposal of features to support analysis and debugging of declarative model transformations with graphical syntax by embedded visualizations. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp 326–330. https://doi.org/10.1109/MODELS-C.2019.00051

George L, Wider A, Scheidgen M (2012) Type-safe model transformation languages as internal DSLs in scala. In: Theory and Practice of Model Transformations, ICMT 2012. https://doi.org/10.1007/978-3-642-30476-7_11

Gerpheide CM, Schiffelers RRH, Serebrenik A (2016) Assessing and improving quality of QVTo model transformations. Softw Qual J 24(3):797–834. https://doi.org/10.1007/s11219-015-9280-8. https://doi.org/10.1007/s11219-015-9280-8

Götz S, Tichy M, Kehrer T (2021) Dedicated model transformation languages vs. general-purpose languages: a historical perspective on ATL vs Java. In: *MODELSWARD* (pp. 122–135)

Groner R, Beaucamp L, Tichy M, Becker S (2020) An exploratory study on performance engineering in model transformations. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, Association for Computing Machinery, New York, NY, USA, MODELS '20, p 308–319. https://doi.org/10.1145/3365438.3410950

Götz S, Tichy M, Groner R (2021) Claimed advantages and disadvantages of (dedicated) model transformation languages: a systematic literature review. Softw Syst Model 20(2):469–503. https://doi.org/10.1007/s10270-020-00815-4. https://doi.org/10.1007/s10270-020-00815-4

Hebig R, Seidl C, Berger T, Pedersen JK, Wąsowski A (2018) Model transformation languages under a magnifying glass: A controlled experiment with Xtend, ATL, and QVT. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018. https://doi.org/10.1145/3236024.3236046

Hermans F, Pinzger M, van Deursen A (2009) Domain-specific languages in practice: A user study on the success factors. In: Model Driven Engineering Languages and Systems, MODELS 2009. https://doi.org/10.1007/978-3-642-04425-0_33

Hibberd M, Lawley M, Raymond K (2007) Forensic debugging of model transformations. In: Engels G, Opdyke B, Schmidt DC, Weil F (eds) Model Driven Engineering Languages and Systems. Springer, Berlin, Heidelberg, pp 589–604. ISBN: 978-3-540-75209-7

Hinkel G (2016) NMF: A Modeling Framework for the. NET Platform. KIT

Hinkel G, Burger E (2019) Change propagation and bidirectionality in internal transformation DSLs. In: Software & Systems Modeling. 18.1, pp. 249–278. https://doi.org/10.1007/s10270-017-0617-6

Hinkel G, Goldschmidt T (2019) Using internal domain-specific languages to inherit tool support and modularity for model transformations. In: Software & Systems Modeling, Reussner R. https://doi.org/10.1007/s10270-017-0578-9

Horn T (2013) Model querying with FunnyQT. In: International Conference on Theory and Practice of Model Transformations, Springer, pp 56–57. https://doi.org/10.1007/978-3-642-38883-5_7

Hove SE, Anda B (2005) Experiences from conducting semi-structured interviews in empirical software engineering research. In: 11th IEEE International Software Metrics Symposium (METRICS'05), 10 pp.–23. https://doi.org/10.1109/METRICS.2005.24

Höppner S, Kehrer T, Tichy M (2021) Contrasting dedicated model transformation languages vs. general purpose languages: A historical perspective on ATL vs. Java based on complexity and size. In: Software and Systems Modeling. https://doi.org/10.1007/s10270-021-00937-3

Hutchinson J, Rouncefield M, Whittle J (2011) Model-driven engineering practices in industry. In: Proceedings of the 33rd International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '11, p 633–642. https://doi.org/10.1145/1985793.1985882. https://doi.org/10.1145/1985793.1985882

Hutchinson J, Whittle J, Rouncefield M, Kristoffersen S (2011) Empirical assessment of MDE in industry. In: Proceedings of the 33rd International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '11, p 471–480. https://doi.org/10.1145/1985793.1985858. https://doi.org/10.1145/1985793.1985858

Hutchinson J, Whittle J, Rouncefield M (2014) Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. In: Science of Computer Programming 89. Special issue on Success Stories in Model Driven Engineering pp. 144–161. https://doi.org/10.1016/j.scico.2013.03.017. https://www.sciencedirect.com/science/article/pii/S0167642313000786

Johannes J, Zschaler S, Fernández MA, Castillo A, Kolovos DS, Paige RF (2009) Abstracting complex languages through transformation and composition. In: Model Driven Engineering Languages and Systems. MODELS 2009. https://doi.org/10.1007/978-3-642-04425-0_41

Jonkers H, Stroucken M, Vdovjak R, Campus HT (2006) Bootstrapping domain-specific model-driven software development within philips. In: 6th OOPSLA Workshop on Domain Specific Modeling (DSM 2006), Citeseer, p 10

Jouault F, Allilaire F, Bézivin J, Kurtev I, Valduriez P (2006) ATL: A QVT-like transformation language. In: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06. https://doi.org/10.1145/1176617.1176691

Juhnke K, Tichy M, Houdek F (2020) Challenges concerning test case specifications in automotive software testing: assessment of frequency and criticality. https://doi.org/10.1007/s11219-020-09523-0. https://doi.org/10.1007/s11219-020-09523-0

Kahani N, Bagherzadeh M, Cordy JR, Dingel J, Varró D (2019) Survey and classification of model transformation tools. https://doi.org/10.1007/s10270-018-0665-6

Kallio H, Pietilä A-M, Johnson M, Kangasniemi M (2016) Systematic methodological review: developing a framework for a qualitative semi-structured interview guide. In: Journal of Advanced Nursing 72.12, pp. 2954?2965. https://doi.org/10.1111/jan.13031

Kernighan BW, Pike R (1984) The unix programming environment. Prentice hall. Englewood Cliffs, NJ

Kolovos DS, Paige RF, Polack FAC (2008) The epsilon transformation language. In: Theory and Practice of Model Transformations. ICMT 2008. https://doi.org/10.1007/978-3-540-69927-9_4

Kramer ME, Hinkel G, Klare H, Langhammer M, Burger E (2016) A controlled experiment template for evaluating the understandability of model transformation languages. In: 2nd International Workshop on Human Factors in Modeling, HuFaMo 2016; Saint Malo; France; 4 October 2016 through. Ed. : M. Goulao. Vol. 1805. CEUR Workshop Proceedings. CEUR Workshop Proceedings, pp. 11?18

Krause C, Tichy M, Giese H (2014) Implementing graph transformations in the BulkăSynchronousăParallel model. In: Gnesi S, Rensink A (eds) Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 325–339

Kuckartz U (2014) Qualitative text analysis: A guide to methods, practice and using software. Sage, ISBN: 978-1-4462-6774-5

Lawley M, Raymond K (2007) Implementing a practical declarative logic-based model transformation engine. In: Proceedings of the 2007 ACM Symposium on Applied Computing, SAC '07. https://doi.org/10.1145/1244002.1244216

Liebel G, Tichy M, Knauss E, Ljungkrantz O, Stieglbauer G (2018) Organisation and communication problems in automotive requirements engineering. In: Requirements Engineering 23.1, pp. 145–167. https://doi.org/10.1007/s00766-016-0261-7. https://doi.org/10.1007/s00766-016-0261-7

Liepiņš R (2012) Library for model querying: Iquery. In: Proceedings of the 12th Workshop on OCL and Textual Modelling, OCL '12. https://doi.org/10.1145/2428516.2428522

Malavolta I, Muccini H, Pelliccione P, Tamburri D (2010) Providing architectural languages and tools interoperability through model transformation technologies. In: IEEE Transactions on Software Engineering 36.1, pp. 119?140. https://doi.org/10.1109/TSE.2009.51

Mayring P (1994) Qualitative Inhaltsanalyse. IVK Univ.-Verl. Konstanz, ISBN: 3-87940-503-4

Mens T, Gorp PV (2006) A taxonomy of model transformation. In: Electronic Notes in Theoretical Computer Science (GraMoT 2005). https://doi.org/10.1016/j.entcs.2005.10.021

Metzger A (2005) A systematic look at model transformations. In: Model-driven Software Development, Springer, pp 19–33. https://doi.org/10.1007/3-540-28554-7_2

Meyer MA, Booker JM (1990) Eliciting and analyzing expert judgment: A practical guide. https://doi.org/10.2172/5088782
Microsoft Language server protocol specification. https://microsoft.github.io/language-server-protocol/specifications/specification-current/
Mohagheghi P, Dehlen V (2008) Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In: Schieferdecker I (ed) Model Driven Architecture ? Foundations and Applications. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 432–443
Mohagheghi P, Gilani W, Stefanescu A, Fernandez MA (2013) An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. In: Empirical Software Engineering. https://doi.org/10.1007/s10664-012-9196-x
Mohagheghi P, Gilani W, Stefanescu A, Fernandez MA, Nordmoen B, Fritzsche M (2013) Where does model-driven engineering help? experiences from three industrial cases. In: Software & Systems Modeling 12.3, pp. 619–639. https://doi.org/10.1007/s10270-011-0219-7. https://doi.org/10.1007/s10270-011-0219-7
Newcomer KE, Hatry HP, Wholey JS (eds) (2015) Handbook of practical program evaluation (p. 492). USA: John Wiley & Sons, https://doi.org/10.1002/9781119171386
OMG (2001) Model driven architecture (MDA), ormsc/2001-07-01
OMG (2014) Object contraint language. https://www.omg.org/spec/OCL/2.4/About-OCL/
OMG (2016) Meta Object Facility (MOF). https://www.omg.org/spec/MOF
OMG (2016) Meta object facility (MOF) 2.0 query/view/transformation specification. https://www.omg.org/spec/QVT/About-QVT/
Pietron J, Raschke A, Stegmaier M, Tichy M, Rukzio E (2018) A study design template for identifying usability issues in graphical modeling tools. In: MODELS Workshops, pp 336–345
Raggett D, Le Hors A, Jacobs I et al (1999) HTML 4.01 Specification. In: W3C recommendation 24
Rainer A, Wohlin C (2021) Recruiting credible participants for field studies in software engineering research. https://doi.org/10.48550/ARXIV.2112.14186
SAEMobilus (2004) Architecture analysis and design language (AADL)
Samiee A, Tiefnig N, Sahu JP, Wagner M, Baumgartner A, Juhász L (2018) Model-driven-engineering in education. In: 2018 18th International Conference on Mechatronics - Mechatronika (ME), pp 1–6
Schmidt D (2006) Guest editor's introduction: Model-driven engineering. In: Computer-IEEE Computer Society. https://doi.org/10.1109/MC.2006.58
Selic B (2003) The pragmatics of model-driven development. In: IEEE Software 20.5, pp. 19?25. https://doi.org/10.1109/MS.2003.1231146
Sendall S, Kozaczynski W (2003) Model transformation: the heart and soul of model-driven software development. In: IEEE Software. https://doi.org/10.1109/MS.2003.1231150
Sprinkle J, Mernik M, Tolvanen J, Spinellis D (2009) Guest editors' introduction: What kinds of nails need a domain-specific hammer? In: IEEE Software 26.4, pp. 15?18. https://doi.org/10.1109/MS.2009.92
Staron M (2006) Adopting model driven software development in industry – a case study at two companies. In: Model Driven Engineering Languages and Systems, MODELS 2006. https://doi.org/10.1007/11880240_5
Stegmaier M, Raschke A, Tichy M, Meßner EM, Hajian S, Feldengut A (2019). In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). https://doi.org/10.1109/MODELS-C.2019.00063
Steinberg D, Budinsky F, Merks E, Paternostro M (2008) EMF: eclipse modeling framework. Pearson Education
Stol KJ, Ralph P, Fitzgerald B (2016) Grounded theory in software engineering research: A critical review and guidelines. In: Proceedings of the 38th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '16, p 120–131. https://doi.org/10.1145/2884781.2884833, https://doi.org/10.1145/2884781.2884833
Strüber D, Anjorin A (2016) Comparing reuse mechanisms for model transformation languages: Design for an empirical study. In: HuFaMo@ MoDELS, Citeseer, pp 27–32
Tehrani SY, Zschaler S, Lano K (2016) Requirements engineering in model-transformation development: An interview-based study. In: International Conference on Theory and Practice of Model Transformations, Springer, pp 123–137. https://doi.org/10.1007/978-3-319-42064-6_9
Troya J, Segura S, Burgueño L, Wimmer M (2022) Model transformation testing and debugging: A survey. In: ACM Computing Surveys (*CSUR*). https://doi.org/10.1145/3523056
Van Deursen A, Klint P (2002) Domain-specific language design requires feature descriptions. In: Journal of Computing and Information Technology. https://doi.org/10.2498/cit.2002.01.01
Vollstedt M, Rezat S (2019) An introduction to grounded theory with a special focus on axial coding and the coding paradigm. In: Kaiser G, Presmeg N (eds) Compendium for Early Career

Researchers in Mathematics Education. Springer International Publishing, Cham, pp 81–100. https://doi.org/10.1007/978-3-030-15636-7_4

Weiber R, Mühlhaus D (2021). In: Strukturgleichungsmodellierung: Eine anwendungsorientierte Einführung in die Kausalanalyse mit Hilfe von AMOS, SmartPLS und SPSS, 3rd edn. Springer-Verlag. https://doi.org/10.1007/978-3-658-32660-9

Whittle J, Hutchinson J, Rouncefield M, Burden H, Heldal R (2013) Industrial adoption of model-driven engineering: Are the tools really the problem? In: Model-Driven Engineering Languages and Systems, MODELS 2013. https://doi.org/10.1007/978-3-642-41533-3_1

Wimmer M, Kusel A, Schoenboeck J, Kappel G, Retschitzegger W, Schwinger W (2009) Reviving qvt relations: Model-based debugging using colored petri nets. In: Schürr A, Selic B (eds) Model Driven Engineering Languages and Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 727–732

Affiliations

Stefan Höppner[1] · Yves Haas[1] · Matthias Tichy[1] · Katharina Juhnke[1]

Yves Haas
yves.haas@uni-ulm.de

Matthias Tichy
matthias.tichy@uni-ulm.de

Katharina Juhnke
katharina.juhnke@uni-ulm.de

[1] Ulm University, James-Franck-Ring 1, Ulm 89081, Germany

F.3 Paper D

Investigating the Origins of Complexity and Expressiveness in ATL Transformations

S. Götz, M. Tichy

Journal of Object Technology (JoT), volume 19, article number 2, July 2020
AITO — Association Internationale pour les Technologies Objets

DOI: doi:10.5381/jot.2020.19.2.a12

Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Investigating the origins of complexity and expressiveness in ATL transformations

Stefan Götz[a] Matthias Tichy[a]

a. Institute of Software Engineering and Programming Languages, Ulm University, Germany

Abstract Model transformations provide an essential element to the model driven engineering approach. Over the years, many languages tailored to this special task, so-called model transformation languages, have been developed. A multitude of advantages have been proclaimed as reasons to why these dedicated languages are better suited to the task of transforming models than general purpose programming languages. However, little work has been done to confirm many of these claims. In this paper, we analyse ATL transformation scripts from various sources to investigate three common claims about the expressiveness of model transformation languages. The claims we are interested in assert that automatic trace handling and implicit rule ordering are huge advantages for model transformation languages and that model transformation languages are able to hide complex semantics behind simple syntax. We use complexity measures to analyse the distribution of complexity over transformation modules and to gain insights about what this means for the abstractions used by ATL. We found that a large portion of the complexity of transformations stem from simple attribute assignments. We also found indications for the usefulness of conditioning on types, implicit rule ordering and automatic trace resolution.

Keywords ATL; Complexity; Expressiveness; Model Transformation Languages; Analysis.

1 Introduction

Model transformations are a pivotal part of model-driven engineering (MDE) [SK03, Sch06]. This is also evident from the amount of transformation languages that have been proposed, i.e. ATL [JAB+06], Henshin [ABJ+10], ETL [KPP08], Viatra [BV06] and QVT [Kur07] just to name a few.

While the number of transformation languages and their features is ever increasing, little time is spent on empirical studies on the use of said languages [SCD17]. A fact

Stefan Götz, Matthias Tichy. *Investigating the origins of complexity and expressiveness in ATL transformations*. In Journal of Object Technology, vol. 19, no. 2, 2020, pages 12:1–21. doi:10.5381/jot.2020.19.2.a12

that is true not only for model transformation languages but any kind of DSL as evident from the results of [KBM16].

The authors of [SCD17], have shown that studying the use of transformation languages on code repositories such as the ATL Zoo [1] can provide insights into how a transformation language is used which can help developers with language evolution.

Such studies are also necessary because there is continual debate about whether dedicated model transformation languages are necessary at all [HSB+18, BCG19] since GPLs like Java can also be used for writing transformations and have been discussed as an alternative since the introduction of model transformations [SK03].

In the study described here, we apply these goals to the Atlas model Transformation Language (ATL) [JAB+06]. We are particularly interested in investigating transformation scripts to gather data concerning the following claims which have been made multiple times in literature:

***H1*:** *Model transformation languages hide complex semantics behind simple syntax [JABK08, KCF14, SK03, GK03].*

***H2*:** *Automatic handling and resolution of trace information by the transformation engines is a huge advantage of model transformation languages [JABK08, LR07, HGBR19].*

***H3*:** *Model transformation languages allow for implicit rule ordering which can lessen the load on developers [JABK08, LR07].*

One thing that immediately stands out from the three claims is that they are intertwined. Automatic handling of traces and implicit rule ordering are both concepts that can hide certain semantics within the transformation engine. So to investigate their impact and provide insights into the complexity within model transformations as a whole we devised 5 research questions to focus our research on:

***RQ1*:** *How is the complexity of ATL transformations distributed over multiple transformations and transformation components?* This question forms a basis data set for the following investigations. Its results can provide useful insights into where the complexity in ATL transformations originates from to provide starting points for more focused investigations. It can also help to uncover potential strengths and weaknesses of the abstractions used by ATL (*H1*).

***RQ2*:** *When looking at the complexity distributions of individual transformation components, are there any salient characteristics?* ATL components such as the out-pattern consist of a set of bindings that assign values to the attributes of the output model. The question that arises from such structures is, whether the complexity of out-patterns stems largely from single complex bindings or a number of simpler bindings. With this research question we aim to investigate such effects which can indicate points where ATL does a good job of hiding complexity (*H1*)

***RQ3*:** *How does the usage of refining mode impact the complexities of ATL modules?* ATLs refining mode was introduced to ease refinement transformations by allowing developers to only focus on the code generating modified elements while leaving all other elements unchanged. Accordingly, the complexity of

[1] `https://www.eclipse.org/atl/atlTransformations/`

refining mode transformations should originate to large parts in refining activities. Otherwise it would indicate that the refining mode fails in supporting developers with model refinements. This in turn would be a counterexample to the claims made in *H1*.

RQ4: *How large is the percentage of bindings that require trace-based binding resolution?* Before being able to argue about the usefulness of trace information (*H2*) for model transformations it should be investigated to what extent their existence influences a model transformation script. If only a small proportion of transformations utilize traces then maybe the development effort for implicit trace handling is not worth it.

RQ5: *What portion of ATL transformations use implicit rule ordering?* The amount of implicitly ordered rules compared to manual rule ordering can be a good indication into whether the feature is well liked by developers hinting at an advantage over manual ordering.

To answer the research questions we selected a total of 33 ATL transformations from various sources to analyse. We use two sets of complexity measures based on [LKRSA18] to measure the complexity of ATL transformations. A meta-model representing the basic components of ATL modules is used to compile the complexity values together. Information about trace usage and rule ordering is taken directly from the models representing the ATL transformations.

The remainder of this paper is structured as follows: First in Section 2 an introduction into relevant aspects of ATL is given. Section 3 defines the used complexity measures. Afterwards in Section 4 we present our extraction and analysis procedures. The results of our analysis are then presented in Section 5. Section 7 discusses potential threats to the validity of the described proceedings while Section 6 places the approach in the context of existing work. Lastly Section 8 concludes and proposes potential future work.

2 The Atlas Transformation Language (ATL)

Specifications in ATL are organized in one of three kinds of so called *Units*. A unit is either a *module*, a *library* or a *query*. Depending on their type, units can consist of *rules*, *helpers* and *attributes*, which are a special kind of helper.

ATL uses the Object Constraint language (OCL) [OMG06] for both data types and expressions.

2.1 Modules

Modules are used to define transformations. ATL modules are made up of three segments (see Listing 1): the *module header* which defines the modules name as well as the types of the input and output meta-models, a number of optional *imports* and a set of *helper* and *rule* definitions.

```
1 module NAME
2   create OUT1:OUTTYPE1, ...
3   [from|refining] IN1:INTYPE1, ...
4
5   [uses LIBRARY]*
```

```
6   [RULEDEF|HELPERDEF]*
```

Listing 1 – Structure of an ATL module

Libraries consist of a set of helper definitions. Libraries can be imported into modules.

Lastly, **Queries** are comprised of an import section, a *query* element and a set of *helper* definitions. Queries are used to define transformations from models to simple OCL types rather than output models.

2.2 Helpers and Attributes

Helpers allow developers to define outsourced expressions that can be called from within rules. Helper definitions can define a data type for which the helper is specified, called *context*. ATL also allows developers to define so called *Attribute* helpers. The main difference between *attributes* and *helpers* is that *attributes* do not accept parameters. *Attributes* serve as constants that are defined for a specific context.

The definition of both traditional helpers and attribute helpers follow the same syntax patterns (see Listing 2). The only difference lies in whether input parameters are defined.

```
1 helper [context CONTEXTTYPE]? def : NAME[(PARAMETERS)]? : TYPE =
    EXPR;
```

Listing 2 – Syntax to define Helpers

2.3 Rules

In ATL, *rules* are used to specify the transformation of input models into output models. There exist two main types of rules: *called rules* and *matched rules*. Matched rules enable a declarative way to define how a model element of a specific type is transformed into output model elements, while called Rules enable generation of target model elements from imperative code. Matched rules are executed automatically on all matching input model elements by the ATL engine.

Matched rules are comprised of four main sections (see Listing 3):

An *In-Pattern* which defines source model elements that are being transformed. In-Patterns can contain a filter expression which defines a condition that must be met for the rule to be applied.

An optional *Using-Block* that allows to define local variables.

The *Out-Pattern* which defines a number of output model elements that are created for the model element defined in the in-pattern when the rule is applied. Each output model element is defined by an *Out-Pattern element* which contains so called *bindings* that assign values to attributes of the model element.

And lastly an optional *Action-Block* which allows the specification of imperative code that is executed once the target elements have been created.

```
1 [lazy| unique lazy]? rule NAME {
2   from
3     INVAR : INTYPE [(CONDITION)]*
4   [using {
5     [VAR : VARTYPE = EXPR;]+
```

```
  }]?
  to
    [OUTVAR : OUTTYPE {
      [ATR <- EXPR,]+
    },]+
  [do {
    [STATEMENT;]*
  }]?
}
```

Listing 3 – Syntax to define matched rules

Apart from regular **matched rules** there are also **lazy** rules. They are defined by adding the key word *lazy* in front of a matched rule definition. Lazy rules are executed only when explicitly called for a specific model element that matches the rules type and filter expression. Lazy rules can be called multiple times on the same model element to produce multiple distinct output elements.

Unique lazy rules, defined through the *unique lazy* key words, change this behaviour. Instead of producing a new model element for each call, unique lazy rules always return the same output element when called on the same input model element.

Lastly, **called rules** are defined in a similar fashion to **matched rules** (see Listing 4). The main difference between the two being that **called rules** do not contain an *In-Pattern* and allow the definition of required parameters.

```
rule NAME([PARAMETER,]*) {
  [using {
    [VAR : VARTYPE = EXPR;]+
  }]?
  to
    [OUTVAR : OUTTYPE {
      [ATR <- EXPR,]+
    },]+
  [do {
    [STATEMENT;]*
  }]?
}
```

Listing 4 – Syntax to define called rules

2.4 Refining mode

The refining mode is a special execution mode for ATL rules which is intended to assist developers with refactoring models, i.e., endogenous transformations.

Normally, the ATL engine only produces output model elements for input elements on which rules are executed on. When using the refining mode however, the ATL engine executes all rules on matching input elements and produces a copy of all unmatched elements. This way developers are able to focus solely on the refining part of their refactoring efforts according to the language developers.

3 Complexity Measures

There exist several approaches for measuring complexity of model transformation languages and ATL in particular ([DRDRIP15, Vig09, TSMGD+11, KGBH10, LKRSA18]). Most of these approaches use a simple metric that relates the number of transformation components such as rules or helpers to the complexity of a transformation module. In our opinion, however, the number of rules or helpers alone does not capture the complexity of model transformations well enough. For that reason, we opted to adopt the complexity measure proposed by [LKRSA18] which includes not only the number of transformation components but also the complexity of expressions used within the transformation.

In the following, the complexity measures will be explained.

3.1 Syntactic complexity

The syntactic complexity $c(\tau)$ of a transformation specification τ is defined based on the complexity of expressions and activities within the defined transformation [LKRSA18]. The general idea behind it being that the complexity of each construct is comprised of a static value for the construct itself plus the sum of the complexities of its contained elements.

The complexity of a module as defined in Listing 1 would be comprised of the sum of the complexities for its contained helper definitions and rule definitions. The complexity of rules, defined as shown in Listings 3 and 4, is then comprised of the complexity of their contained `from`-block (In-Pattern), the `to`-block (Out-Pattern), the `using`-block and `do`-block plus a static value of 1 for the rule itself.

The complexity of In-Patterns is defined by their contained filter expression and a static value for the construct itself, while the complexity of Out-Patterns is defined by a static value for the construct as well as the sum of the complexities of all contained Out-Pattern elements and their contained bindings. An overview over the most important complexity measure definitions can be found in Table 1 for expressions and Table 2 for activities/structural elements[2].

We adopted the complexity measure with slight modifications since we disagreed with certain defined values. The following adjustments were made to the definition from [LKRSA18]:

First, the complexity of helpers was adapted to also include the complexity of their context. The reason for this change being the fact that the context of a helper has to be considered when trying to understand its function. Furthermore, in our opinion there is no difference between attribute and operation helpers, the additional, static complexity attributed to both types of helper definitions was aligned at 1. For this the static complexity of attribute helpers was reduced from 3 to 1 and that of operation helpers was increased from 0 to 1.

Action blocks were given an additional static complexity value of 1 which was missing from the definitions of [LKRSA18]. This aligns it with the static complexity that is attributed to all elements contained within rules, i.e. In-Patterns, Out-Patterns and Using-blocks.

The complexity attributed to operation calls was increased to 1 to align it with that of attribute and navigation calls. In our opinion calling an operation on an object is

[2]Full definitions can be found in `https://spgit.informatik.uni-ulm.de/stefan.goetz/mtl-complexities/blob/master/ATL/transformations/qvt/transforms/complexity.qvto`

just as complex as accessing one of its attributes. For the same reason the complexity for collection operation calls was also increased to 1 as well.

Table 1 – Definitions of expression complexity measure based on [LKRSA18].

Expression e	Complexity $c(e)$
Numeric, boolean or String value	0
Identifier *iden*	1
Attribute call $source.attr$	$c(source) + 2$
Operation call $source.op(p1, ..)$	$2 + c(source) + \sum_i c(p_i)$
Operator call $e1\ op\ e2$	$c(e1) + c(e2)$
CollectionOperation call $source- > op(p1, ..)$	$2 + c(source) + \sum_i c(p_i)$
$if\ e1\ then\ e2\ else\ e3\ endif$	$c(e1) + c(e2) + c(e3) + 1$
$let\ v\ :\ t\ =\ e1\ in\ e2$	$c(t) + c(e1) + c(e2) + 4$
CollectionExpression $Col\{e1, ..\}$	$1 + \sum_i c(e_i)$
Primitive Type (Integer,String,...)	1
Collection Type $Col(t)$	$1 + c(t)$

3.2 Computational complexity

The computational complexity is an extension of the syntactic complexity. Its goal is to more closely capture the underlying complexity of transformation definition with respect to outsourced expressions and called transformation rules. To achieve this, the complexity of `Operation Call`s is calculated by taking the complexity of the called operation into account instead of adding a static value regardless of the called operation. For example given a helper `sample` of syntactic complexity 12, the call `sample()` has a syntactic complexity of 2 whereas its computational complexity amounts to 12.

Moreover the complexity of used variables is also resolved by taking the definition expression of the variable into account instead of using a static value of 1.

4 Methodology

Apart from the selection of the ATL transformation modules to analyse, we strongly oriented our proceedings along the research questions from section 1.

4.1 Module Selection

The selection of ATL modules was aimed to achieve a wide spread of transformations based on their source, purpose and size in terms of lines of code. We also aimed to achieve an even distribution of modules that use the refining mode and modules that do not.

For this purpose, we searched GitHub for ATL projects by using the search string ‘ATL transformation‘ and included all novel (meaning not present in the ATL zoo) transformations for which we were also able to find the input and output meta-models

Table 2 – Definitions of complexity measure for ATL elements/activities based on [LKRSA18]. ATL elements are capitalized while expression elements are written in lower case.

ATL element A	Complexity $c(A)$
Module $H_1, .., R_1, ..$	$\sum_i c(H_i) + \sum_i c(R_i)$
Helper *helper context c def* : n : t = e	$c(c) + c(t) + c(e)$
MatchedRule *rule* N $\{From\ Using\ To\ Do\}$	$c(From) + c(To) + c(Do) + c(Using)$
CalledRule *rule* $N(p)$ $\{Using\ To\ Do\}$	$c(To) + c(Do) + c(Using)$
VariableDefinition n : t = e	$c(t) + c(e) + 3$
InPattern *from* s : t (f)	$c(f) + c(t) + 3$
OutPattern $o : t$ $\{B_1, ..\}$	$c(t) + \sum_i c(B_i) + 2$
Binding n <- e	$c(e) + 2$
ActionBlock *do* $\{S\}$	$c(S)$
$S1; S2$	c(S1) + c(S2)
if e then $S1$ else $S2$	$c(e) + c(S1) + c(S2) + 1$
for v : e do S	c(e) + c(S) + 1
Binding Statement v <- e	c(v) + c(e) + 1

Table 3 – Meta-data about the analysed transformation modules.

Data	minimum	average	maximum	total
LOC	39	408	1364	13455
Rules	1	14	55	460
Helpers	0	11	74	376
Bindings	2	112	487	3695

since those were required for parts of our analysis(see Section 4.4). This resulted in a total of 16 transformation modules. Additionally we included the R2ML2XML transformation from [vAvdB11] and the families2persons transformation from the ATL zoo because it is a widely used example for model transformations. We then supplemented the set of transformations with transformations from the ATL zoo to try and achieve an even distribution between modules that use the refining mode and modules that do not.

The result was a set of 33 ATL transformations (some meta-data about the transformations can be found in Table 3). Of those 33 transformations, 15 use the refining mode of ATL while 18 are exogenous transformations. A complete overview over the selected transformations, including names and sources can be found under `https://spgit.informatik.uni-ulm.de/stefan.goetz/mtl-complexities/blob/master/ATL/resources/input/cases/justifications`.

4.2 RQ1,2: How is the complexity of ATL transformations distributed over multiple transformations and transformation components and are there any salient characteristics?

To be able to collect and analyse complexity data of ATL transformations and relevant elements thereof a meta-model was constructed [3]. Its structure was designed to be able to break down the full representation of an ATL transformation into the basic components that make up ATL transformations as described in Section 2. With this structure it is also possible to investigate where the complexity of entire ATL modules and rules originate from, e.g. whether a rule is complex because of its size or due to a few complex contents like filter expressions. The design of the meta-model followed the principles of abstraction and pragmatics. Compared to the ATL meta-model our developed meta-model focuses solely on those parts of the ATL transformations we are interested in and provides an easy way to track their complexity and the origin thereof.

To transform transformation modules into a model of the presented meta-model and to calculate the complexities of its components along the way, a QVT-o transformation [4] was defined. Its correctness was evaluated using unit tests: A test module containing at least one of each activities/expressions for which a complexity value can be calculated was defined. The complexity values for each element was calculated manually based on the previously introduced complexity definition. Afterwards the results of the transformation were manually compared with the manually calculated complexity values. Discrepancies between the complexity values were investigated and corrected.

In order to collect data for analysis, the tested transformation was applied to the 33 ATL transformations.

Apart from the raw complexity data, we resorted to using several diagrams such as histograms, violin and alluvial plots as well as code snippets to investigate the complexity distribution, both syntactical and computational, of ATL transformations.

In order to better understand the meaning behind the complexity values example code snippets for each component were extracted from the 33 selected transformation modules. The code snippets were selected so that their complexity values correspond to the components median complexity within the data set. One such code snippet can be seen in Listing 5. All used snippets can be found under `https://spgit.informatik.uni-ulm.de/stefan.goetz/mtl-complexities/tree/master/ATL/resources/input/medians`.

```
1 helper context SimpleClass!Class def: associations: Sequence(
      SimpleClass!Association)=
2   SimpleClass!Association.allInstances() -> select(asso | asso.
      value = 1);
```

Listing 5 – Helper with a syntactic complexity corresponding to the median of all helper complexities.

[3] the meta-model can be found under `https://spgit.informatik.uni-ulm.de/stefan.goetz/mtl-complexities/-/tree/master/ATL/metamodels/ATLComplexity/model`

[4] `https://spgit.informatik.uni-ulm.de/stefan.goetz/mtl-complexities/blob/master/ATL/transformations/qvt/transforms/complexity.qvto`

4.3 RQ3: How does the usage of refining mode impact the complexities of ATL modules?

As explained in Section 1, we also intended to analysed ATL modules using the refining mode as a example of how transformation languages hide semantics.

To do so, we used the 15 selected transformation modules that use refining mode and analysed their complexities separately and in comparison to those modules not using the refining mode.

4.4 RQ4: How large is the percentage of bindings that require trace-based binding resolution?

To investigate the usefulness of trace-based binding resolution (and thus to an extent that of implicit trace management) we resorted to analysing how often it is used in transformation modules. A high proportion of trace-based resolutions used would then indicate their usefulness. Since trace-based binding resolution only happens along reference types of the input and output elements we extracted all reference types per module element for all output meta-models. For this we used a simple Java-program that given an Ecore-file would produce a list of reference types for each contained `EClass`.

Afterwards the bindings within all selected transformation modules were analysed for usage of the extracted reference types. The amount of bindings that use traces compared to simple assignments was then analysed on the basis of these results.

4.5 RQ5: What portion of ATL transformations use implicit rule ordering?

Similar to the trace usage, the usefulness of implicit rule ordering can be indicated by the distribution of implicitly ordered transformation elements compared to explicitly ordered ones.

Called and lazy matched rules all get explicitly ordered by developers when calling them while matched rules enable the ATL transformation engine to traverse the source model and implicitly order their execution. Thus the ratio of matched rules to called and lazy rules gives an indication into how relevant implicit rule ordering is for model transformations.

Data for this analysis can be gathered from both the complexity distributions from *RQ 1* as well as directly from the number of definitions.

5 Result Summary and Analysis

We present the results of our analysis in this section in accordance with the research questions posed in Section 1.

5.1 RQ1: How is the complexity of ATL transformations distributed over multiple transformations and transformation components?

Figures 1 and 2 show alluvial plots over the distribution of syntactic complexity and computational complexity respectively of module elements within ATL transformation modules. They display how much of the complexity of all investigated transformations originate in which components beginning with the modules themselves following the

definitions down to the contained expressions and the static value associated with each component.

Interestingly, while making up nearly 45% of all top level definitions, **Helpers** only contribute to roughly 18% of the total complexity of a transformation module [5]. The largest portion of complexity is attributed to **matched rules** which contribute to over 3/4 of the total complexity of transformation rules while accounting for 53% of all top level definitions. And lastly **called Rules**, which are not widely represented in our data sets, while making up about 1% contribute to 5% of the overall complexity of modules.

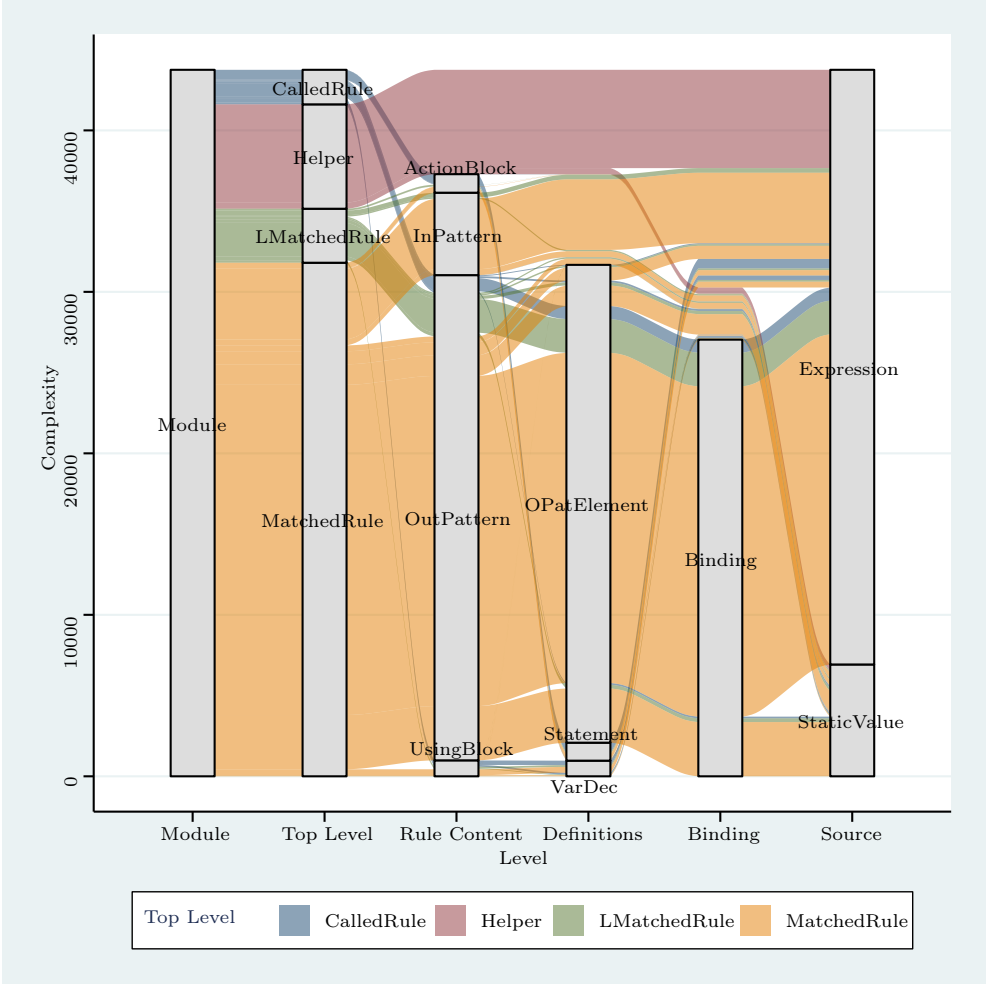


Figure 1 – Distribution of syntactic complexity over all ATL modules.

Another observation that can be made from Figure 1, is that about 80% of the syntactic complexity of (lazy) **Matched-rules** stems from their **Out-Patterns** while only 15% come from **In-Patterns** and a nearly negligible 5% originate in *action-* and **using blocks**. Following this trend downwards 73% of the complexity of these rules stems from their contained bindings, i.e assigning a value to

[5] the raw data can be found under `https://spgit.informatik.uni-ulm.de/stefan.goetz/mtl-complexities/tree/master/ATL/data`

attributes of the output model element. Meaning most effort in transformations is spent not in selecting the correct model elements to transform but simply assigning the output values (see Section 5.2 for a more detailed discussion). This effect is still present when looking at the computational complexity distribution (as shown in Figure 2) which rules out the possibility that the effect is created by outsourcing of filter conditions in In-Patterns through helpers. This leads us to:

Observation 1: Over half of the effort spent in writing ATL transformations is spent assigning values to the output model.

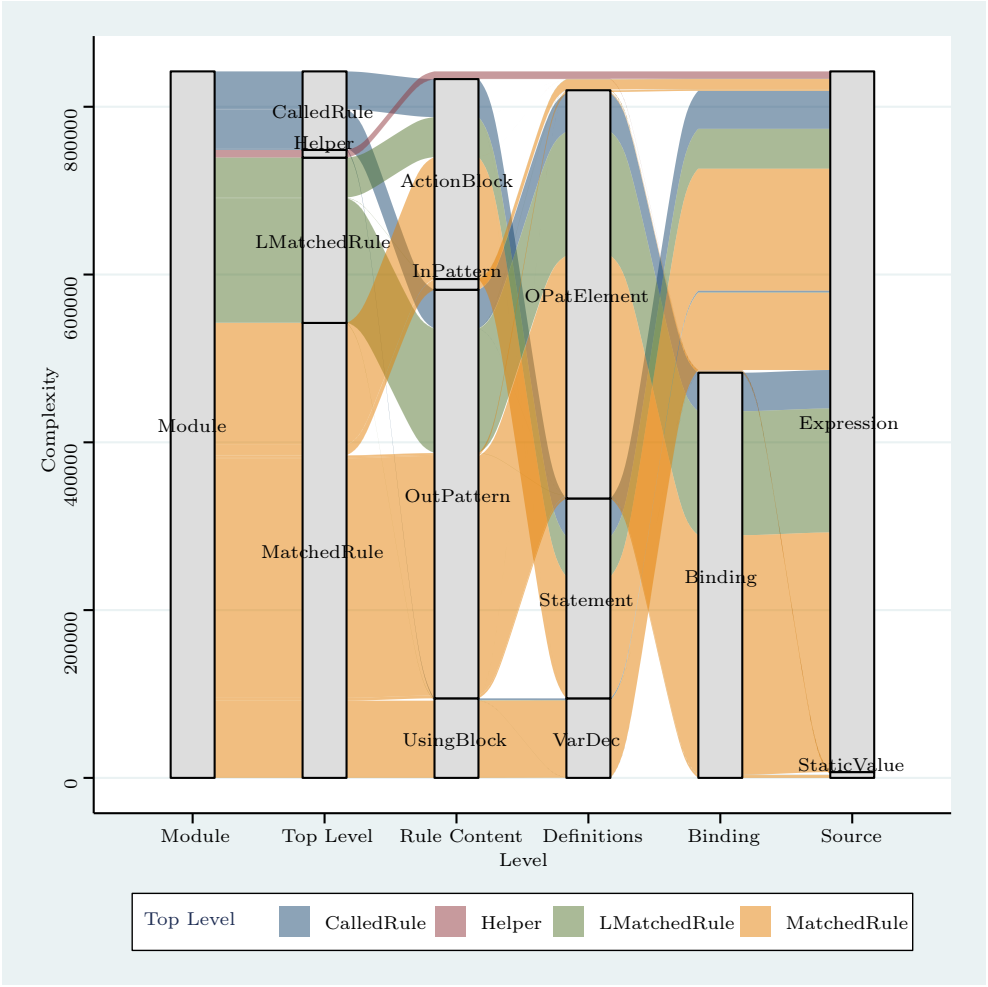


Figure 2 – Distribution of computational complexity over all ATL modules.

Furthermore the 15% of matched rules syntactic complexity that comes from In-Patterns shows that conditioning the application of transformation is a relevant task for model transformations. The low proportion especially when considering the computational complexity rather suggests that the conditioning on types (as opposed to filter conditions without pre conditioning on types) which ATL does for all matched rules by default alone already provides a useful abstraction for model transformations. This assumption is supported by the fact that about 25% of all matched rules get by with only using the standard type conditioning without any

additional filter expression in the **In-Pattern**. Of those 25% only 12% (which therefore constitute only 3% of all matched rules) are trivial transformation rules. Simple transformations in the context of this paper mean transformations that contain no filter condition and only assign attributes from the input model element to the output model element without doing any additional operations. The fact that a large portion of transformations get by with only the default conditioning on types in ATL leads us to:

Observation 2: Conditioning on types provides an abstraction well suited for model transformations.

5.2 RQ2: When looking at the complexity distributions of individual transformation components, are there any salient characteristics?

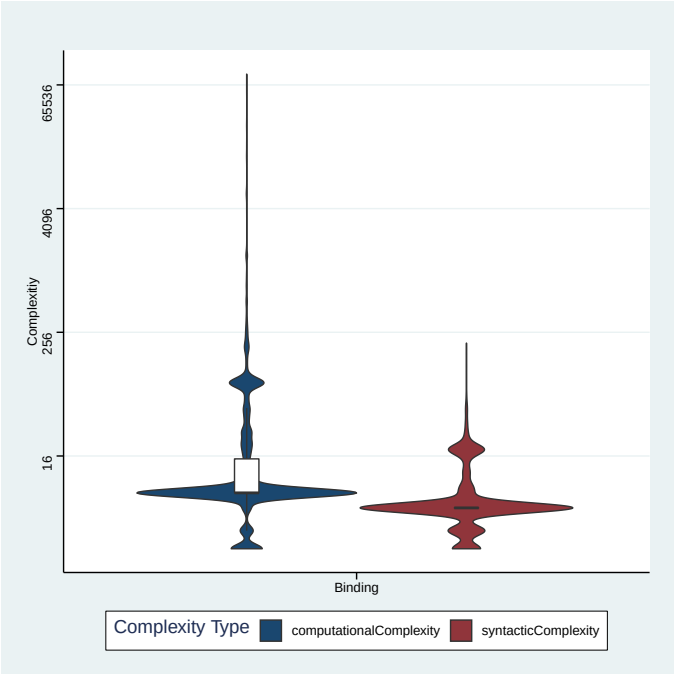


Figure 3 – Syntactic complexity distribution of Bindings.

As previously mentioned, the proportion of the complexity of bindings within transformations also stands out in Figure 1. Bindings alone make up over half of the complexity of transformations, a trend, that persists even when looking at the computational complexity of transformation modules. Interestingly, the complexity within bindings is very unevenly distributed. Figure 3, which shows a violin and box plot for the syntactic complexity distribution of bindings (note the logarithmic scale of the y axis), illustrates this. The majority of all bindings have a syntactic complexity 5 ($b_5 = 60\%$). This corresponds to directly accessing an attribute of an object as shown in Listing 6, calling a helper on said object or accessing a global attribute (`thisModule.attribute`).

Further analysis shows that a total of 93% of all bindings with a syntactic complexity of 5 do indeed stem from direct accesses of attributes of the input model element ($b_{5a} = 93\%$). Only 2% are global attribute accesses and the last 5% originate from helper calls on the source model element. This is also indicated by the

fact that a majority of bindings have a computational complexity of 7 which can only correspond to accessing attributes on input elements. In summary, this leads us to:

Observation 3: Over half ($b_5 * b_{5a} = 56\%$) of all bindings are used to map one attribute of an input model element to one attribute of an output model element.

Adding to this point, only 5% of bindings with a syntactic complexity of 5 stem from trivial transformations, i.e., transformations that simply map attributes from an input element to an output element without doing any meaningful filtering or modification of the content. This reinforces *Observation 3* since we can rule out that the majority of bindings with complexity of 5 stem from trivial transformations which do by definition only contain bindings of this or lower complexity .

Looking at ATL modules as a whole *Observation 3* means that 33% of their total syntactic complexity comes from the activity of copying input model attributes to output model attributes.

```
1 rule MedianBinding {
2   from s : Families!Member
3   to t : Persons!Female (
4     fullName <- s.firstName
5   )
6 }
```

Listing 6 – Rule containing a binding with a syntactic (computational) complexity of 5 (7)

That much of the complexity of transformation modules comes from bindings means that the main effort when writing model transformations in ATL consists of defining how the output should look which is actually one of the main goals of model transformation languages. This in turn suggests that ATL does a good job in abstracting away other tasks in model transformation such as model traversal, conditioning on types as shown in Section 5.1, tracing and rule ordering to which we will come in Sections 5.4 and 5.5.

5.3 RQ3: How does the usage of refining mode impact the complexities of ATL modules?

Given the observations from the previous sections we would expect that the syntactic complexity distribution of bindings to deviate away from 5 (and the computational complexity form 7) since the refining mode is designed to enable developers in focusing only on the refining part of the transformation.

In the transformations investigated for this paper this is however not the case as can be seen in Figure 4, the median syntactic complexity of bindings remains 5 and that of computational complexity remains 7.

This indicates the usefulness of the changes made to the refining mode with the introduction of the 2010 ATL compiler. Since 2010 refining mode allows real in-place transformations which means that rules only need to specify changes to elements while all the other elements remain untouched. And because the main effort in the investigated transformations, which were all defined for ATL compilers prior to 2010, is spent on copying attributes from the input model element (98% of all bindings) to its output counterpart, newer versions of the ATL compiler would heavily reduce this necessary overhead, allowing developers to focus solely on actually refining models. To us this suggests that the current versions of ATLs refining mode can significantly reduce unnecessary overhead for refining transformations. There is also an observation

to be made from this:

Observation 4: GitHub and especially the ATL Zoo lack samples of ATL transformations using the refining mode with compiler versions at least as current as 2010.

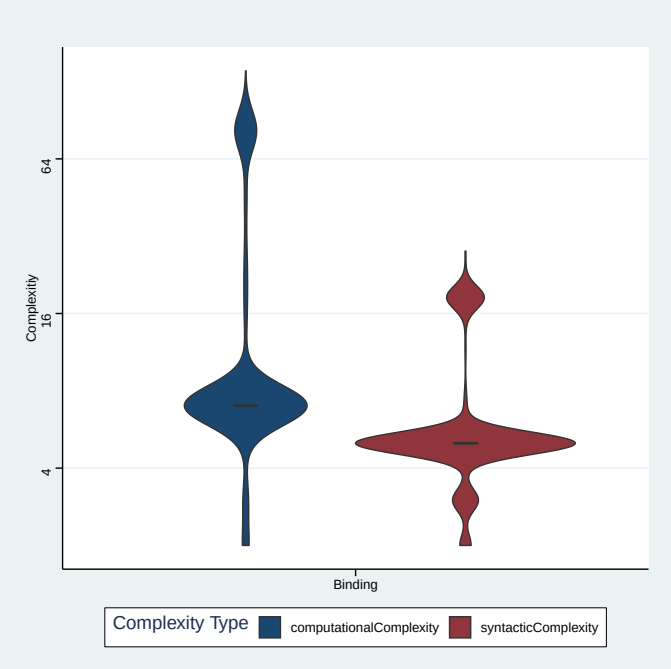


Figure 4 – Complexity distribution of Bindings in refining mode.

Furthermore, Injjj Patterns in refining mode are, on average about twice as complex as in non refining modules. Moreover only a small portion ($\sim 7\%$) of In-Patterns do not contain a filter expression at all compared to $1/3$ of In-Patterns in non refining mode.

This leads us to two additional observations:

Observation 5: When refining models, filters are more heavily used than when transforming between different meta-models.

Observation 6: Filter expressions are more complex in refining mode due to having to select elements with more specific properties.

5.4 RQ4: How large is the percentage of bindings that require trace-based binding resolution?

About 15% of all bindings in the analysed transformation modules require traces. While this makes it apparent that traces are less frequently required than one would expect, it still demonstrates their necessity since 15% is not a negligible proportion. This leads us to:

Observation 7: Bindings that require traces constitute a significant part of the model transformations considered.

It is also worth mentioning that while such trace resolution can save developers substantial amounts of time they can also be a source of errors.

Considering the complexity of the bindings that require traces also reveals something interesting. About half of all bindings that require traces have a syntactic complexity

of 5 and computational complexity of 7. This shows how well automatic trace handling can hide complexity. The developer can simply access the input model element that is supposed to be transformed into the correct output model element and the transformation engine handles resolving and referencing. Would the developer have to take care of this process manually both syntactic and computational complexity would be significantly higher since this would require identifying and accessing the corresponding output model element through additional code.

5.5 RQ5: What portion of ATL transformations use implicit rule ordering?

In the ATL modules analysed for this study, a total of 460 rules are defined. Of those 364 or 79% are matched rules, 84 or 18% are lazy matched rules which need to be invoked to transform model elements and only 12 or 2% are called rules.

Our results, deviate slightly from the results found by [SCD17] but still reveal the same preference trend of ATL developers:

Observation 8: Developers strongly prefer matched rules over lazy matched rules and called rules.

Since matched rules allow for implicit model traversal and rule ordering this can indicate that these concepts provide good support for transformation developers. This is also evident from the fact that the proportion of Out Pattern complexity (both syntactic and computational) to Action Block complexity is far more balanced in called rules than in (lazy) matched rules (see Figures 1 and 2) again indicating that called rules require more structural code such as calling other rules and conditioning.

6 Related Work

In [DRDRIP15], the authors analyse the impact of input and output meta-models on, amongst other things, the complexity of ATL transformations. For this purpose they use a number of meta-model metrics and correlate these with metrics for ATL transformations using Spearman's rank correlation coefficient. Their findings include a high correlation between the number of structural features of the output meta-models and the number of used bindings in an ATL transformation module. An insight which can be reflected upon in our results. In contrast to the complexity measures applied in this work however, the measures proposed for complexity in their work is confined to the number of structural features such as bindings or helpers of ATL transformations rather than the complexity of their structure and contained expressions. Which is not to say that the applied measure is not indicative of the complexity of transformations rather that it is only part of what makes a transformation complex in our opinion.

The authors of [vAvdB11] propose the usage of *cyclomatic complexity* to measure the complexity of helpers. They also envision incorporating the complexity of the contained OCL expression into its complexity measure. Similarly to [DRDRIP15], they also use the number of different transformation components as metrics for measuring ATL transformations. The described metrics are applied to seven transformations. And the resulting values are then related to quality attributes, based on the assessment of nineteen experts, such as *understandability*, *maintainability* and *conciseness* using Kendall's τ_b correlations. Notable results include a significant correlation between the number of transformation rules and *conciseness* and the number of out-patterns and *understandability*. In comparison to this, we try to draw direct conclusions about the

structure and structure of transformations from our gathered data instead of about quality features.

In [Vig09] the complexity of ATL transformations is related to a variety of introduced metrics. Most of the related metrics are once again quantifications of different components within ATL modules such as the *number of matched rules* or *average number of filters* used in rules. They also relate the *cyclomatic complexity* to complexity much like [vAvdB11]. As previously mentioned we believe that the number of components are only part of what makes transformations complex which is why the used complexity measure in this paper also incorporates the complexity of expressions.

The numbers of ATL transformation components have also been used in [TSMGD+11] to make comparisons between several transformation modules to investigate the feasibility of applying transformations to transformation modules. The authors concede that the applied metrics need further research and development and predict that such measures could assist with identifying aspects of ATL transformations to optimize.

Similarly [KSW+13] analysed the ATL Zoo with the goal to gain insights about the frequency of use of reuse mechanisms. For this the authors devised a semi-automated process to extract and analyse projects from the ATL zoo. They found that reuse mechanisms are exclusively used within a transformation and that helpers are the most frequently used reuse mechanism while only little rule inheritance is used. In contrast to their work, our focus does not directly relate to reuse mechanisms although the computational complexity was introduced in part to account for the outsourcing of complexity due to reuse mechanisms.

7 Threats to validity

This section addresses the potential threats to validity identified for the performed study.

The transformations evaluated for the purpose of this study were chosen from various sources to reduce the influence of programming habits of individual transformation engineers. Consequently the purposes and characteristics of the transformations vary immensely. To be able to compare transformation modules using refining mode with modules that do not use refining mode we also aimed to use a similar amount of respective transformation modules. While this strengthens the external validity of our comparison it can potentially lead to a reduction in the external validity of our other findings since an even distribution of refining and non refining modules is potentially less representative of the overall ATL ecosystem. Given the selection of transformation modules it is also not possible to draw representative conclusions about model transformation languages in general but rather for ATL specifically.

There is of course a discussion to be held about the complexity measure used. As discussed in Section 6 most research uses the number of elements as basis for complexity measures. We and [LKRSA18] argue that this alone does not fully cover the complexity of transformations. The syntactic complexity measure used in this study uses the complexity of expressions and activities as defined in Tables 2 and 1. The number of elements are also taken into account in these definitions but do not constitute the majority of the complexity value of an ATL transformation this is reserved to the complexity of expressions used within the transformation modules as evident from Figure 1. While we are missing a formal validation of the measures used we believe that this indicates their overall usefulness. The computational complexity is then a natural extension of the syntactic complexity to more closely resemble the

actual complexity that is hidden in operation calls in expressions.

8 Conclusion and Future Work

In this work we presented our results of analysing ATL modules to provide insights into three common claims about the advantages of model transformation languages, namely that transformation languages hide complex semantics behind simple syntax, that automatic trace handling in transformation languages is advantageous and that implicit rule ordering supports developers in defining transformations.

For this purpose we used two complexity measures to investigate how complexity is distributed over ATL transformation modules which we applied to a total of 33 modules. We also analysed the proportions of matched rules compared to other types of rules and the proportion of bindings that require trace information to be resolved.

We found, that while transformations can get complex, the complexity originates mainly in definitions of how the output models should be populated rather than how the transformation should be executed. To us this provides an indication for how well ATL abstracts away from certain tasks necessary for model transformation such as model traversal, rule ordering and trace handling.

We have also shown that conditioning on types is well suited for model transformations since a total of 22% of all non-trivial matched rules get by with only filtering on types. This also provides a clear example why implicit rule ordering can be beneficial for model transformation definitions since developers can simply define to which kind of input model element a transformation should apply and the transformation engine handles execution.

This is further supported by the fact that we found that nearly 80% of all defined rules are matched rules which make use of exactly this mechanism.

Next we analysed required trace information in bindings. We came to the conclusion that while bindings that do require trace information are outweighed by those that do not, they still constitute a significant portion of model transformations. And while this suggests that automatic trace handling is advantageous further research is necessary to more precisely capture its impact.

Lastly we compared the complexities of transformation modules using the refining mode with those that do not. We found that while the complexity of matched rules defined in a refining module is much higher, the increase in complexity can be attributed to an increase in simple bindings. A fact we were able to attribute to the use of older ATL compilers which did not allow in-place refinements.

For future work, we are interested in repeating the described proceedings on transformations written in general purpose programming languages. While the resulting values can not be compared directly, the complexity distributions can be used to gain insights into where the complexity in these transformation definitions lie. Which we believe can produce further contributions to the discussion of GPLs vs MTLs for defining model transformations.

References

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place emf model transformations. In *Model Driven Engineering*

Languages and Systems, Berlin, Heidelberg, 2010. doi:https://doi.org/10.1007/978-3-642-16145-2_9.

[BCG19] Loli Burgueño, Jordi Cabot, and Sébastien Gérard. The future of model transformation languages: An open community discussion. *Journal of Object Technology*, July 2019. doi:10.5381/jot.2019.18.3.a7.

[BV06] András Balogh and Dániel Varró. Advanced model transformation language constructs in the viatra2 framework. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, New York, NY, USA, 2006. doi:10.1145/1141277.1141575.

[DRDRIP15] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Mining correlations of atl model transformation and metamodel metrics. In *2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering*, 2015. doi:10.1109/MiSE.2015.17.

[GK03] J. Gray and G. Karsai. An examination of dsls for concisely representing model traversals and transformations. In *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, Jan 2003. doi:10.1109/HICSS.2003.1174892.

[HGBR19] Georg Hinkel, Thomas Goldschmidt, Erik Burger, and Ralf Reussner. Using internal domain-specific languages to inherit tool support and modularity for model transformations. *Software & Systems Modeling*, Feb 2019. doi:https://doi.org/10.1007/s10270-017-0578-9.

[HSB+18] Regina Hebig, Christoph Seidl, Thorsten Berger, John Kook Pedersen, and Andrzej Wąsowski. Model transformation languages under a magnifying glass: A controlled experiment with xtend, atl, and qvt. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2018. doi:10.1145/3236024.3236046.

[JAB+06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: A qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, New York, NY, USA, 2006. doi:10.1145/1176617.1176691.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 2008. doi:https://doi.org/10.1016/j.scico.2007.08.002.

[KBM16] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 2016. doi:https://doi.org/10.1016/j.infsof.2015.11.001.

[KCF14] Filip Krikava, Philippe Collet, and Robert France. Manipulating Models Using Internal Domain-Specific Languages. In *Symposium On Applied Computing*, Gyeongju, South Korea, March 2014. doi:10.1145/2554850.2555127.

[KGBH10] Lucia Kapová, Thomas Goldschmidt, Steffen Becker, and Jörg Henss. Evaluating maintainability with code metrics for model-to-model transformations. In *International Conference on the Quality of Software Architectures*, 2010. doi:https://doi.org/10.1007/978-3-642-13821-8_12.

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon transformation language. In *Theory and Practice of Model Transformations*, Berlin, Heidelberg, 2008. doi:https://doi.org/10.1007/978-3-540-69927-9_4.

[KSW+13] Angelika Kusel, Johannes Schönböck, Manuel Wimmer, Werner Retschitzegger, Wieland Schwinger, and Gerti Kappel. Reality check for model transformation reuse: The atl transformation zoo case study. In *Amt@ models*, pages 1–11, 2013.

[Kur07] Ivan Kurtev. State of the art of qvt: A model transformation language standard. In *International Symposium on Applications of Graph Transformations with Industrial Relevance*, 2007. doi:https://doi.org/10.1007/978-3-540-89020-1_26.

[LKRSA18] Kevin Lano, Shekoufeh Kolahdouz-Rahimi, Mohammadreza Sharbaf, and Hessa Alfraihi. Technical debt in model transformation specifications. In *Theory and Practice of Model Transformation*, Cham, 2018. Publishing. doi:10.1007/978-3-319-93317-7_6.

[LR07] Michael Lawley and Kerry Raymond. Implementing a practical declarative logic-based model transformation engine. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, New York, NY, USA, 2007. doi:10.1145/1244002.1244216.

[OMG06] OMG. Object constraint language (ocl), April 2006.

[SCD17] Gehan MK Selim, James R Cordy, and Juergen Dingel. How is atl really used? language feature use in the atl zoo. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2017. doi:10.1109/MODELS.2017.20.

[Sch06] Douglas Schmidt. Guest editor's introduction: Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY*, 03 2006. doi:10.1109/MC.2006.58.

[SK03] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *IEEE Software*, Sep. 2003. doi:10.1109/MS.2003.1231150.

[TSMGD+11] José Barranquero Tolosa, Oscar Sanjuán-Martínez, Vicente García-Díaz, B Cristina Pelayo G-Bustelo, and Juan Manuel Cueva Lovelle. Towards the systematic measurement of atl transformation models. *Software: Practice and Experience*, 2011. doi:https://doi.org/10.1002/spe.1033.

[vAvdB11] Marcel F van Amstel and MGJ van den Brand. Using metrics for assessing the quality of atl model transformations. In *MtATL@ TOOLS*, 2011.

[Vig09] Andrés Vignaga. Metrics for measuring atl model transformations. *MaTE, Department of Computer Science, Universidad de Chile, Tech. Rep*, 2009.

About the authors

Stefan Götz is a PhD student at Ulm University. His research is focused on topics surrounding the development and evaluation of model transformation languages. Prior to his work as a PhD student he was a student of Software Engineering at Ulm University. Contact him at `stefan.goetz@uni-ulm.de`.

Matthias Tichy is full professor for software engineering at the University of Ulm and director of the institute of software engineering and programming languages. His main research focus is on model-driven software engineering, particularly for cyber-physical systems. He works on requirements engineering, dependability, and validation and verification complemented by empirical research techniques. He is a regular member of programme committees for conferences and workshops in the area of software engineering and model driven development. He is co-author of over 110 peer-reviewed publications. Contact him at `matthias.tichy@uni-ulm.de`.

Acknowledgments This work was partially funded through the MICE project of the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - Ti 803/4-1.

F.4 Paper E

Contrasting Dedicated Model Transformation Languages Versus General Purpose Languages: A Historical Perspective on ATL Versus Java Based on Complexity and Size

S. Höppner, M. Tichy, T. Kehrer

International Journal on Software and Systems Modeling (SoSyM), volume 21, pages 805–837, 2022
Springer Nature

DOI: 10.1007/s10270-021-00937-3

Software and Systems Modeling (2022) 21:805–837
https://doi.org/10.1007/s10270-021-00937-3

REGULAR PAPER

Contrasting dedicated model transformation languages versus general purpose languages: a historical perspective on ATL versus Java based on complexity and size

Stefan Höppner[1] · Timo Kehrer[2] · Matthias Tichy[1]

Received: 17 June 2021 / Revised: 19 August 2021 / Accepted: 15 September 2021 / Published online: 17 November 2021

Abstract
Model transformations are among the key concepts of model-driven engineering (MDE), and dedicated model transformation languages (MTLs) emerged with the popularity of the MDE pssaradigm about 15 to 20 years ago. MTLs claim to increase the ease of development of model transformations by abstracting from recurring transformation aspects and hiding complex semantics behind a simple and intuitive syntax. Nonetheless, MTLs are rarely adopted in practice, there is still no empirical evidence for the claim of easier development, and the argument of abstraction deserves a fresh look in the light of modern general purpose languages (GPLs) which have undergone a significant evolution in the last two decades. In this paper, we report about a study in which we compare the complexity and size of model transformations written in three different languages, namely (i) the Atlas Transformation Language (ATL), (ii) Java SE5 (2004–2009), and (iii) Java SE14 (2020); the Java transformations are derived from an ATL specification using a translation schema we developed for our study. In a nutshell, we found that some of the new features in Java SE14 compared to Java SE5 help to significantly reduce the complexity of transformations written in Java by as much as 45%. At the same time, however, the relative amount of complexity that stems from aspects that ATL can hide from the developer, which is about 40% of the total complexity, stays about the same. Furthermore we discovered that while transformation code in Java SE14 requires up to 25% less lines of code, the number of words written in both versions stays about the same. And while the written number of words stays about the same their distribution throughout the code changes significantly. Based on these results, we discuss the concrete advancements in newer Java versions. We also discuss to which extent new language advancements justify writing transformations in a general purpose language rather than a dedicated transformation language. We further indicate potential avenues for future research on the comparison of MTLs and GPLs in a model transformation context.

Keywords ATL · Java · Model transformations · Model transformation language · General purpose language · Comparison · MTL versus GPL · Historical perspective · Complexity measure · Size measure

1 Introduction

Model transformations are among the key concepts of the model-driven engineering (MDE) paradigm [1]. They are a particular kind of software which needs to be developed along with an MDE tool chain or development environment. With the aim of supporting the development of model transformations, dedicated model transformation languages (MTLs) have been proposed and implemented shortly after the MDE paradigm gained a foothold in software engineering.

Communicated by Esther Guerra.

Stefan Höppner
stefan.hoeppner@uni-ulm.de

Timo Kehrer
timo.kehrer@informatik.hu-berlin.de

Matthias Tichy
matthias.tichy@uni-ulm.de

[1] Ulm University, 89081 Ulm, Germany

[2] Humboldt University Berlin, 10099 Berlin, Germany

1.1 Context and motivation

In the literature, many advantages are ascribed to model transformation languages, such as better analysability, comprehensibility or expressiveness [2]. Moreover, model transformation languages aim at abstracting from certain recurring aspects of a model transformation such as traversing the input model or creating and managing trace information, claiming to hide complex semantics behind a simple and intuitive syntax [1,3–5].

Nowadays, however, such claims have two main flaws. First, as discussed by Götz et al., there is a lack of actual evidence to have confidence in their genuineness [2]. Second, we argue that most of these claims emerged together with the first MTLs around 15 years ago. The Atlas Transformation Language (ATL) [6], for example, was first introduced in 2006, at a time when third-generation general purpose languages (GPLs) were still in their infancy. Arguably, these flaws are underpinned by the observation that MTLs have been rarely adopted in practical MDE [7].

Within our research group as well as in conversations with other researchers, the presumption that transformations can just as well be written in a GPL such as Java has been discussed frequently. In fact, in our own research, we have implemented various model transformations using a GPL; examples of this include the meta-tooling facilities of established research tools like SiLift [8] and SERGe [9,10], or the implementation of model refactorings and model mutations in experimental setups of more recent empirical evaluations [11,12]. The presumption that model transformations can just as well be written in a GPL has been confirmed by a community discussion on the future of model transformation languages [7], and, at least partially, by an empirical study conducted by Hebig et al. [13]. Our argumentation for specifying model transformations using a modern GPL is mainly rooted in the idea that new language features allow developers to heavily reduce the boilerplate code that MTLs claim to abstract away from. There are also other features that certain model transformation languages can provide such as graph pattern matching, incrementality, bidirectionality or advanced analysis, but for now our study focuses solely on the abstraction and ease of writing argument.

1.2 Research goals and questions

To validate and better understand this argumentation, we elected to compare ATL, one of the most widely known MTLs, with Java, a widespread GPL. More specifically, we compare ATL with Java in one of its recent iterations (Java SE14) as well as at the level of 2006 (Java SE5) when ATL was introduced.[1] The goal of this approach is twofold. First, we intend to investigate how transformation code written in Java SE14 can be improved compared to the Java code using the Java version SE5 that was timely when ATL was released. Second, we want to contextualize these improvements by relating them to transformation code written in ATL. We opted to use both size and complexity measures for this purpose because both can provide useful insights for this discussion.

In order to achieve these goals, we developed four research questions to guide our research efforts:

RQ1 How much can the complexity and size of transformations written in Java SE14 be improved compared to Java SE5?
RQ2 How is the complexity of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to each other and ATL?
RQ3 How is the size of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to each other and ATL?
RQ4 How does the size of query aspects of transformations written in Java SE5 & SE14 compare to each other and ATL?

RQ1 aims to provide a general overview of how both size and complexity of transformations in Java might be improved using language features provided in newer Java versions. For a more detailed discussion and comparison it is then necessary to inspect and compare how the transformation code based is associated to the different aspects of a transformation, e.g. *model traversal*, *tracing* or the actual *transformation* of elements. This is the goal of **RQ2** and **RQ3** for complexity and size, respectively. With these two research questions we aim to investigate for which aspects new language features of Java help to reduce size and complexity of the associated code segments and what this means compared to ATL. Lastly, it is often assumed that querying aided by language constructs in MTLs is one key factor for their suitability over GPLs [14]. With **RQ4** we aim to investigate this assumptions via an explicit comparison between queries written in Java and ATL.

1.3 Research methodology

The process to answer the discussed research questions was structured around four consecutive steps. First, we selected a total of 12 existing ATL transformations taken from the

[1] Interestingly, there was no significant evolution of the ATL language since its initial introduction in 2006 [7].

Table 1 Meta-data about the selected transformation modules

Transformation name	LOC	# rules	# helpers
ATL2BindingDebugger	41	2	0
ATL2Tracer	96	2	0
DDSM2TOSCA	582	19	2
ExtendedPN2ClassicalPN	86	7	0
Families2Persons	49	2	2
istar2archi	99	6	1
Modelodatos2FormHTML	127	9	3
Palladio2UML	189	19	0
R2ML2XML	1125	60	1
ResourcePN2ResourceM	44	3	1
SimpleClass2RDBMS	63	4	3
UML22Measure	371	27	11
Average	236.25	13.3	2

ATL Zoo[2] and several projects from GitHub[3] to the basis for our study. References to all included transformations can be found in our supplementary material in Höppner, Tichy, and Kehrer [15]. The selection of ATL modules was done with several goals in mind. First, we wanted to include transformations of different size and purpose. We also aimed to include both transformations using ATLs' refining mode and normal transformations. Lastly, due to the fact that our translations would be done manually, we decided to limit the total number of transformations to 12 and the maximum size of a single transformation to around 1000 LOC. Since our work is, in part, based on the work presented in Götz and Tichy [16] and their selection criteria align with ours, we opted to make the selection of modules from the set of transformations analysed by them. The module selection process resulted in a total of 12 ATL transformations, from a variety of sources including the ATL Zoo. Basic meta-data about the transformations can be found in Table 1, while further details can be found in the supplementary materials.

Next, we devised, and tested, a schema to translate the selected ATL transformations to Java. To develop the translation schema, we followed the design science research methodology [17] using an iterative pattern for designing and enhancing the schema until it fit our purpose. To validate the correctness of the translated transformations, we used the input and output models that were provided within the ATL transformation projects. The input models were used as input for the Java transformations, and the output models were then compared with the output of the ATL transformations.

Afterwards, we developed a classification schema to divide Java code into its components and relate each component to the different aspects of the transformation process, i.e. transforming, tracing, and traversing. All Java code was then labelled based on the classification schema. For ATL, a similar schema from Götz et al. [16] already exists which we adopted and applied to the selected ATL transformations.

Lastly, we decided on and applied several code measures to allow us to compare the transformations. For comparing transformations specified in Java SE5 and SE 14, we use a combination of four metrics for measuring size and complexity, namely lines of code (LOC), word count (# words) [18], McCabe's cyclomatic complexity [19], and weighted method count (WMC). We use WMC based on McCabe complexity, i.e. the sum of the McCabe complexities of all elements, as the complexity measure in cases where the complexities of several elements need to be grouped together. Word count is used to supplement the standard code size measure LOC as a measure that is less influenced by code style and independent from keyword and method name size [18]. Furthermore, word count allows a direct size comparison between ATL and Java, which is hardly possible with LOC due to the languages' significantly different structure.

Our comparison of complexity and size distributions is thus based on LOC, word count, McCabe's cyclomatic complexity, and WMC, and we incorporate the findings of Götz et al. on how code is distributed within ATL transformations [16].

1.4 Results

Our analysis for **RQ1** shows that newer Java versions allow for a significant reduction in code complexity and lines of code, while the number of required words stays about the same. We attribute this to a more information dense style of writing single statements in the more functional programming style enabled by Java SE8 (2014) and newer.

The results for **RQ2** reflect the reduction in complexity overhead mainly in the methods involving model traversal. We also conclude that in newer language versions the most prominent remaining complexity overhead stems from manual trace management in Java compared to ATL.

The more detailed investigation done for **RQ3** supports these observations. We show that tracing is not only a prominent part in the methods dedicated to trace management but also in the methods that are dedicated to actually transforming input into output elements.

Overall the results for **RQ2** and **RQ3** suggest that still, a lot of complexity and size overhead for traversal, tracing, and supplementary code is required in Java even though newer Java versions improve the overall process of writing transformations. Of these, tracing is the biggest obstacle for efficiently developing transformations in a general purpose language. The overhead associated with this transformation aspect is the most significant and, arguably, most error-prone

[2] https://www.eclipse.org/atl/atlTransformations.

[3] https://www.github.com.

one. A large portion of the advancements of Java SE14 over Java SE5 stem from the inclusion of more recent language aspects such as streams and functional interfaces. This fact is highlighted in our results from **RQ4** where those two aspects are the main factors for improvements in the size of OCL expressions written in Java.

1.5 Contributions and paper structure

This paper extends prior work on comparing Java and ATL transformations [20]. The extension consists of (i) a more detailed description of the applied translation schema from ATL to Java, (ii) the inclusion of an additional measure, namely number of words, for comparison, and (iii) the consideration of a larger set of transformations. Furthermore, we (iv) greatly expanded our discussion of overhead introduced by using Java for transformations based on the results from the newly included measure. This includes a more detailed inspection of Java code as well as a direct comparison between Java and ATL. Additionally, based on all the results and our own experiences, we (v) are now able to discuss more explicitly what newer Java versions improve over older ones and where the language is still lacking compared to ATL. Finally, we (vi) present a description of scenarios where these advancements are enough to justify Java over ATL and (vii) consider other features of model transformation languages not present in ATL and their impact on the suitability of general purpose languages.

The remainder of this paper is structured as follows: First, Sect. 2 introduces the relevant aspects of ATL as well as the relevant differences between Java 5 and Java 14. Afterwards, in Sect. 3, we give an overview of how we translate ATL transformations to Java. Because the discussions for **RQ2&3** require a precise classification of how code segments in Java are associated to the different transformation aspects, we provide an explanation for this in Sect. 4. In Sect. 5, we present our detailed method for analysing the size and complexity of the translated transformations. The results of our analysis and extensive comparison between the different transformation approaches are then presented in Sect. 6. Based on these results, Sect. 7 discusses our take-aways for what newer Java versions improve over older ones, where the language did not advance, and when these advancements are enough to justify Java over ATL. Section 8 then discusses potential threats to the validity of our work, while related work is considered in Sect. 9. Lastly, Sect. 10 concludes the paper and presents potential avenues for future research.

2 Background

In this section, we briefly introduce the relevant background knowledge required for this paper. First, since model transformations can only be specified precisely based on some concrete model representation, we introduce the structural representation of models in MDE which is typically assumed by all mainstream model transformation languages, including ATL. Afterwards, since our work builds on ATL as well as the technological advancement of Java, it is necessary to introduce the relevant background knowledge on ATL and to present the important differences between Java SE5 and Java SE14, respectively.

2.1 Models in MDE

In MDE, the conceptual model elements of a modelling language are typically defined by a meta-model. The Eclipse Modeling Framework (EMF) [21], a Java-based reference implementation of OMG's Essential Meta Object Facility (EMOF) [22], has evolved into a de-facto standard technology to define meta-models that prescribe the valid structures that instance models of the defined modelling language may exhibit. It follows an object-oriented approach in which model elements and their structural relationships are represented by objects (EObjects) and references whose types are defined by classes (EClasses) and associations (EReferences), respectively. Local properties of model elements are represented and defined by object attributes (EAttributes). A specific kind of references are containments. In a valid EMF model, each object must not have more than one container and cycles of containments must not occur. Typically, an EMF model has a dedicated root object that contains all other objects of the model directly or transitively.

2.2 ATL

ATL distinguishes among three kinds of so-called *Units*, being either a *module*, a *library* or a *query*. Depending on the type of unit, they consist of *rules*, *helpers* and *attributes*. For data types and expressions, ATL uses the Object Constraint Language (OCL) [23].

2.2.1 Units

As illustrated in Listing 1, transformations are defined in *Modules*, taking a set of input models (line 3) which are transformed to a set of output models (line 2) by *rule* and *helper* definitions which make up the transformation (line 6).

Libraries do not define transformations but only consist of a set of helper definitions. Libraries can be imported into modules to enhance their functionality (line 5).

Queries are special types of libraries that are used to define transformations from model elements to simple OCL types. They are comprised of a *query* element and a set of *helper* definitions.

```
1 module NAME
2 create OUT1:MetaModelB, ...
3 [from|refining] IN1:MetaModelA, ...
4
5 [uses LIBRARY]*
6 [RULEDEF|HELPERDEF]*
```

List. 1 Structure of an ATL module.

```
1 helper [context MODELTYPE]? def :
      NAME[(PARAMETERS)]? :TYPE  = EXPR;
```

List. 2 Syntax to define Helpers.

```
1  [lazy| unique lazy]? rule NAME {
2      from
3      INVAR : MODELATYPE [(CONDITION)]*
4      [using {
5          [VAR : VARTYPE = EXPR;]+
6      }]?
7      to
8      [OUTVAR : MODELBTYPE {
9          [ATR <- EXPR,]+
10     },]+
11     [do {
12         [STATEMENT;]*
13     }]?
14 }
```

List. 3 Syntax to define matched rules.

2.2.2 Helpers and attributes

Helpers allow outsourcing of expressions that can be called from within rules, similar to simple functions in general purpose languages. Helper definitions can specify a so-called *context* which defines the data type for which the helper is defined as well as parameters passed to the helper. ATL also allows the definition of *attribute* helpers. Attribute helpers differ from helpers in that they do not accept any parameter and always require a context data type. They serve as constants for the specified context. Listing 2 shows the syntax to define helpers and attribute helpers.

2.2.3 Rules

In ATL, transformations of input models into output models are defined using *rules*. There are two main types of rules: *matched rules* and *called rules*.

Matched rules The declarative part of an ATL transformation is comprised by matched rules which are automatically executed on all matching input model elements, thus allowing to define type-specific transformations into output model elements. For this, the ATL engine traverses the input model in an optimized order. Furthermore, matched rules generate *traceability* links (trace links for short) between the source and target elements. These links can be navigated throughout the transformation specification to access references to elements created from a source element. Matched rules are comprised of four sections (see Listing 3):

- The *In-Pattern* (lines 2 to 3) defines the type of source model elements that are to be matched and transformed. An optional filter expression allows the definition of a condition that must be met for the rule to be applied.
- An optional *Using-Block* (lines 4 to 6) allows to define local variables based on the input element.
- The *Out-Pattern* (lines 7 to 10) then defines a number of output model elements that are to be created from the input element when the rule is applied. Each output model element is defined using a set of so-called *bindings* for assigning values to attributes of the output model element.
- Lastly, an optional *Action-Block* (lines 11 to 13) can be defined which allows the specification of imperative code that is executed once the target elements have been created.

Matched rules can also be defined as *lazy* rules by adding the keyword *lazy* to the rule definition (line 1). In contrast to regular matched rules, lazy rules are only executed when explicitly called for a specific model element that matches both the rule's type and its filter expression. They can be called multiple times on the same model element to produce multiple distinct output elements. To change the behaviour of lazy rules to always produce one and the same output element for the same source model element, lazy rules can be declared as *unique* (line 1).

Called rules As opposed to matched rules, called rules enable an explicit generation of target model elements in an imperative way. Called rules can be called from within the imperative code defined in the *Action-Block* of rules. They are defined similarly to matched rules. The main difference is that they do not contain an *In-Pattern* but instead allow the definition of required parameters. These parameters can then be used in the *Out-Pattern* and *Action-Block* to produce output model elements.

2.2.4 Refining mode

The refining mode is a special execution mode for ATL modules which aims at supporting an easy definition of in-place transformations [24,25]. Normally, the ATL engine only creates new output model elements from input model elements matched by the rules defined in a module. However, in the refining mode, the ATL engine instead executes all rules on matching input elements and produces a copy

```
public interface Function<T,R> {
    public R apply(T par);
}
```

List. 4 Definition of the Function interface.

```
Function<Integer, Integer> doubleIt = (value)
    -> value * 2;
```

List. 5 Lambda expression definition based on Function.

```
List<String> myList =
    Arrays.asList(1,2,3,4,5,6);
myList.stream().filter(i -> i % 2 ==
    0).forEach( System.out::println);
```

List. 6 Finding and printing all even numbers in a list.

of all unmatched input elements automatically. This aims to allow developers to focus solely on local modifications such as model refactorings rather than also having to manually produce copies of all other model elements.

2.3 Technological advancements in Java SE14 compared to Java SE5

Since the release of J2SE 5 in September of 2004, there have been a lot of improvements made to the Java language. In this section, however, we will only cover the ones relevant in the context of this paper. All the relevant features relate to a more functional programming style as they allow developers to express some key aspects of a transformation specification more concisely.

2.3.1 Functional interfaces

With the introduction of the *functional interfaces* in Java SE8, Java made an important step towards embracing the functional programming paradigm, paving the way to define lambda expressions in arbitrary Java code. Lambda expressions, also called anonymous functions, are functions that are defined without being bound to an identifier. This allows developers to pass them as arguments.

In essence, a *functional interface* is an interface containing only a single abstract method. One example of this is the interface called `Function<T,R>` (see Listing 4). It represents a function which takes a single parameter and returns a value. This abstract method can then be implemented by means of a Java lambda expression (see Listing 5).

Lambdas defined with the interface `Function<T,R>` as their type are then nothing more than objects with their definition as the implementation of the apply method wrapped in a more functional syntax (see Listing 5).

Java provides a number of predefined functional interfaces, such as the aforementioned `Function<T,R>`, or `Consumer<T>` which takes one argument and has void as its return value.

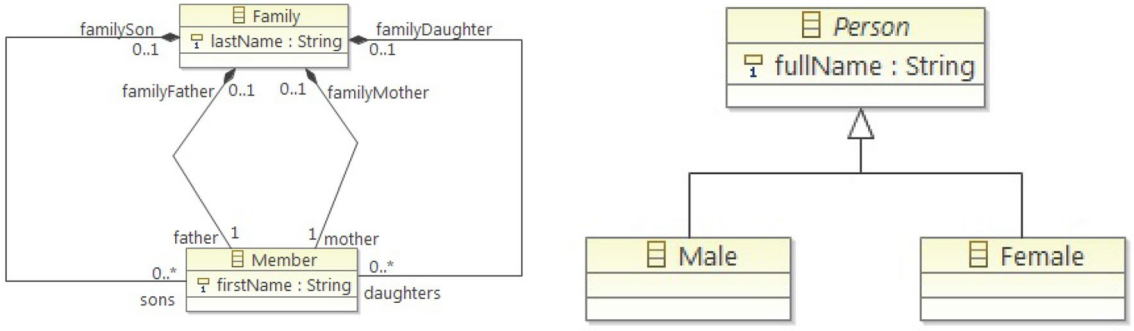
2.3.2 Streams

Streams represent a sequence of elements and allow a number of different operations to be performed on the elements within the sequence. Stream operations can either be intermediate or terminal. This means that the operations can either produce another stream as their result or a non-stream result which therefore terminates the computation on the stream. This also means that intermediate operations work with all elements within the stream without the developer having to define a loop over it.

The example in Listing 6 shows how one can find and print all even numbers in a list using streams.

3 Translation schema

In the following, we will present a detailed description of, first, how the translation schema was developed (see Sect. 3.1), before then describing the translation schema itself (Sects. 3.2 to 3.6).



(a) The Families meta-model. **(b)** The Persons meta-model.

Fig. 1 The Families and Persons meta-models from the families2persons case taken from the ATL wiki [27]

The description of the translation schema is split into five parts. In Sect. 3.2, we describe the general setup used to emulate ATL semantics in Java and the basic structure that all translated modules follow. Then, in Sect. 3.3, we introduce and describe three libraries to reduce repetitive code between translated modules, one for trace handling, one for model traversal, and one for model loading and persisting. Sections 3.4 and 3.5 describe how the essential building blocks, namely matched rules and called rules, of ATL transformations are translated into Java. And lastly, in Sect. 3.6 we explain how helpers and general OCL expressions are translated.

All descriptions are illustrated by the use of a running example. For this, we use an ATL solution found in the ATL Zoo for the families2persons case from the TTC'17 [26] the code of which can be found in Listing 7, while its Java SE14 counterpart can be found in Listing 8. The meta-models for the transformation case are shown in Fig. 1. The example illustrates how different ATL elements are translated into their corresponding Java code based on the described schemata. Our descriptions will focus on the Java SE 14 translation schemata. Notable differences between the Java SE 14 and Java SE5 translation schemata are highlighted as such.

3.1 Schema development

To develop the translation schema, we followed the design science research methodology [17]. We used the ATL solution found in the ATL Zoo for the families2persons case from the TTC'17 [26] as our initial test input for the translation scheme and focused on developing the schema for Java SE14.

The development process followed a simple, iterative pattern. A translation schema was developed by the main author and applied to the Families2Persons case. The resulting transformation was then reviewed by one co-author, focusing on completeness and meaningfulness. Afterwards, the results of the review were used as input for reiterating the process.

In a final evolution step, the preliminary transformation schema was applied to all 12 selected ATL transformations. Afterwards, both co-authors reviewed the resulting transformations separately based on a predefined code review protocol. In a joint meeting, the results of the reviews were discussed and final adjustments to the transformation schema were decided. These were then used to create a final translation of all 12 ATL transformations.

Lastly we ported the developed transformations to Java SE5 by forking the project, reducing the compiler compliance level, and re-implementing the parts that were not compatible with older compiler versions.

To validate the correctness of the translated transformations, we used the input and output models that were provided within the ATL transformation projects. The input models were used as input for the Java transformations and the output models were then compared with the output of the transformations. Since neither an input nor an output model was available for the R2ML2XML transformation, we had to rely solely on the results of our code reviews for its validation. This validation approach is similar to how Sanchez Cuadrado et al. [28] validate their generated code.

Our translation schema allows us to translate any ATL module into corresponding Java code. The only assumption we make is that all the meta-models of input and output models are explicitly available. The reason for this is that we work with EMF models in so-called static mode, which means that all model element types defined by a meta-model are translated into corresponding Java classes using the EMF built-in code generator.

3.2 General setup and module translation

In our translation scheme, we generally assume that each model contains a single root element. This is standard for EMF but could be easily extended by using lists as input and output.

An ATL module is represented by a Java class which contains a single point of entry method that takes the root element of the input model as its input and returns the root element of the output model. The `transform` method in line *13* of Listing 8 represents this entry point for the families2persons transformation. It takes the *root* model element of type `Family` from the input model and returns a List of type `Person` which serves as the root element for the `Persons` meta-model.[4]

```
1  module Families2Persons;
2  create OUT : Persons from IN : Families;
3
4  helper context Families!Member def: familyName :
       String =
5      if not self.familyFather.oclIsUndefined() then
6          self.familyFather.lastName
7      else
8          if not self.familyMother.oclIsUndefined()
       then
9              self.familyMother.lastName
10         else
11             if not self.familySon.oclIsUndefined()
12         then
13                 self.familySon.lastName
14             else
15                 self.familyDaughter.lastName
16             endif
17         endif
18     endif;
19
20 helper context Families!Member def: isFemale()
21         : Boolean =
22     if not self.familyMother.oclIsUndefined() then
23         true
24     else
25         if not
       self.familyDaughter.oclIsUndefined()
26         then
27             true
28         else
29             false
30         endif
31     endif;
32
33 rule Member2Male {
34     from
35         s : Families!Member (not s.isFemale())
36     to
37         t : Persons!Male (
38     fullName <- s.firstName + ' ' + s.familyName
39     )
40 }
41
42 rule Member2Female {
43     from
44         s : Families!Member (s.isFemale())
45     to
46         t : Persons!Female (
47     fullName <- s.firstName + ' ' + s.familyName
48     )
49 }
```

List. 7 Families2Persons ATL solution.

[4] In reality the `Persons` meta-model does not have a root element and the list is used as a substitute for the transformation to conform with the translation schema as well as general EMF standards. To produce this list from the transformed elements the `Family2List` method in lines *36-42* is introduced which does not have a counterpart in ATL.

Additionally, some setup code is needed for extracting a model and its root element from a given source file, calling the entry point of the actual transformation class, and serializing the resulting output model. The code required for our running example is shown in Listing 9. We utilize one of our developed libraries, namely `IO`, for reading an xmi-file containing a `Families` model, extracting the root object of type `Family` and passing it to the `transform` method of the `Families2Persons` class to initiate the actual transformation. The resulting output of type `List<Person>` is then written to an xmi-file, again, utilizing our `IO` library.

Because traceability links need to be created before they can be used, we split the transformation process into two separate runs. The first run creates all target elements as well as all traceability links between them and their source elements, while the second run can safely traverse over model references and populate the created elements by utilizing the traceability links when needed. Consequently, the corresponding Java transformation class comprises two separate methods, dedicated to each run and being called by the entry point method. In our example in Listing 8, the methods `preTransform(Family root)` and `actualTransform(Family root)` in lines *18* and *25* represent these two runs. Their implementation will be explained later throughout Sect. 3.4.

3.3 Libraries

For both model traversal as well as trace generation and resolving, we developed generic libraries which can be reused across all transformation classes. Additionally, we also required a library to outsource the reading and writing of models from and into files. The remainder of this section will describe these libraries in more detail.

3.3.1 IO library

The IO library contains methods used for reading and writing models from and to files. The library exposes two methods, namely `readModel(String uri)` and `persistModel(EObject root, String uri)` which both bundle together several EMF and file-IO methods to achieve the desired effects. To do so the library utilizes the `Resource`[5] type which represents a "persisted document" in EMF and allows to read and write `EObjects` from and to it. To be able to read and write different file types such as *xmi* or *ecore*, a corresponding `ResourceFactory` needs to be registered in the *ExtensionToFactoryMap* of the ResourceFactory registry. For this reason, we opted to only support *xmi*, *ecore* and *uml* files since EMF provides default ResourceFactory implementations for all three.

The `persistModel` method takes a root element of a model as well as a desired output path, and creates a resource containing the root element (and all its children) which is then saved to the specified path. The `readModels` method reverses this approach by extracting the resource pointed to by the passed path and returning all contents of the referenced resource to the caller. Due to the makeup of EMF compliant files such as *xmi*, *ecore* or *uml* the first element within the contents will then always contain the root element of the model within the file which can then be used as seen in Listing 9.

3.3.2 Traversal library

The traversal library allows us to outsource the traversal of the source model and thus reduce the amount of boilerplate code written for each translated transformation. It builds upon a `HashMap` that maps a `Class<?>` to a `Consumer<EObject>`. The `Consumer<EObject>` interface represents a function that takes an input object of type `EObject` and has a return type of `void`. During traversal, which is encapsulated within the library, the Consumer function that corresponds to an `EObject` can be retrieved from the `HashMap` by using the class of the EObject as key. To achieve this, the library exposes the methods `addFunction` and `traverseAndAccept`.

The `addFunction` method allows us to add a key-value-pair to the encapsulated hashmap. The `traverseAndAccept` method then takes an `Iterable` collection containing `EObjects`, iterates over all contained objects, fetches the function that corresponds to the concrete class of the `EObject`, and executes it. This way, we only have to write code that adds the required key-value-pairs to the traverser, while the code for traversing the input model as well as resolving the correct function which is to be called is completely outsourced. Note that adding such function calls is only necessary for *matched rules* since *lazy* and *called rules* are called within the transformation code and not automatically executed based on element- type matching. An example of how the traversal library is used can be found in lines *19-22* and *28-31* of Listing 8 and will be explained in more detail in Sect. 3.4.

For the Java SE5 solution we decided on an alternative solution using the conditional dispatcher pattern instead of outsourcing the traversal. The reason for this was a weighing of alternatives. Outsourcing the traversal in Java SE5 would require the utilisation of anonymous classes. This in turn would offer a similar workflow and an equal McCabe complexity for defining model traversal as with the functional interface solution in Java SE14. It would however significantly increase the required number of words and lines of code compared to the conditional dispatcher solution. Only

[5] https://download.eclipse.org/modeling/emf/emf/javadoc/2.4.3/org/eclipse/emf/ecore/resource/Resource.html.

```
public class Families2Persons {
  private static final PersonsFactory PERSONSFACTORY = PersonsFactory.eINSTANCE;
  private static final Tracer TRACER = new Tracer();

  private static boolean isFemale(Member member) {
    return member.getFamilyDaughter() != null || member.getFamilyMother() != null;
  }

  private static String familyName(Member member) {
    return ((Family) member.eContainer()).getLastName();
  }

  public static List<Person> transform(Family family) {
    preTransform(family);
    return actualTransform(family);
  }

  private static void preTransform(Family root) {
    var iterator = root.eAllContents();
    var traverser = new Traverser(TRACER);
    traverser.addFunction(Member.class, x -> {Member2MalePre((Member) x);Member2FemalePre((Member) x);});
    traverser.traverseAndAcceptPre(iterator);
  }

  private static List<Person> actualTransform(Family root) {
    var newRoot = Family2List(root);

    var iterator = root.eAllContents();
    var traverser = new Traverser(TRACER);
    traverser.addFunction(Member.class, x -> {Member2Male((Member) x);Member2Female((Member) x);});
    traverser.traverseAndAccept(iterator);

    return newRoot;
  }

  private static List<Person> Family2List(Family root) {
    var persons = new LinkedList<Person>();
    persons.add(TRACER.resolve(root.getFather(), Male.class));
    persons.add(TRACER.resolve(root.getMother(), Female.class));
    persons.addAll(root.getDaughters().stream().map($ -> TRACER.resolve($, Female.class)).collect(Collectors.toList()));
    persons.addAll(root.getSons().stream().map($ -> TRACER.resolve($, Male.class)).collect(Collectors.toList()));
    return persons;
  }

  private static void Member2MalePre(Member m) {
    if (!isFemale(m)) {
      TRACER.addTrace(m, PERSONSFACTORY.createMale());
    }
  }
  private static void Member2Male(Member m) {
    var t = TRACER.resolve(m, PERSONSFACTORY.createMale());
    t.setFullName(m.getFirstName() + " " + familyName(m));
  }

  private static void Member2FemalePre(Member m) {
    if (isFemale(m)) {
      TRACER.addTrace(m, PERSONSFACTORY.createFemale());
    }
  }
  private static void Member2Female(Member m) {
    var t = TRACER.resolve(m, PERSONSFACTORY.createFemale());
    t.setFullName(m.getFirstName() + " " + familyName(m));
  }
}
```

List. 8 The Families2Persons solution translated in Java SE14.

with the improved syntax provided through the functional interfaces in Java SE8 could a decrease of the McCabe complexity be accompanied with an uniform word count and lines of code. Overall, the decision leads to an increase in the McCabe complexity of the traversal code in Java SE5 but allows for word count and LOC to remain stagnant. We will come back and discuss the impact of this decision (in the relevant parts of our results discussion| in Sect. 7.1) later on.

This design decision affects the methods `preTransform` and `actualTransform`. Their implementation in Java SE5 is shown in Listing 10. Instead of populating the traverser objects we instead manually iterate over the whole model and decide which methods to call based on the type of the currently visited object.

```
1 List<EObject> ins =
      IO.readModel(''Family.xmi'');
2 Family family = (Family) ins.get(0);
3 List<Person> persons =
      Families2Persons.transform(family);
4 IO.persistModel(persons, ''persons.xmi'');
```

List. 9 Setup code for the Families2Persons transformation.

```
1  //...
2  private static void preTransform(Family root) {
3      TreeIterator<EObject> iterator =
       root.eAllContents();
4      while (iterator.hasNext()) {
5          EObject next = iterator.next();
6          if (next instanceof Member) {
7              Member m = (Member) next;
8              Member2MalePre(m);
9              Member2FemalePre(m);
10         }
11     }
12 }
13
14 private static List<Person>
       actualTransform(Family root) {
15     List<Person> newRoot = Family2List(root);
16
17     TreeIterator<EObject> iterator =
       root.eAllContents();
18     while (iterator.hasNext()) {
19         EObject next = iterator.next();
20         if (next instanceof Member) {
21             Member m = (Member) next;
22             Member2Male(m);
23             Member2Female(m);
24         }
25     }
26     return newRoot;
27 }
28 //...
```

List. 10 Translated model traversal in Java SE5.

3.3.3 Trace library

The trace library emulates the management of traceability links of ATL. Similar to the traversal library, the trace library is built based on a `HashMap`. In this case, however, the `HashMap` maps source `EObjects` to target `EObjects` and thus can be used both in Java SE5 and Java SE14.

In essence, the trace library exposes two methods. First, for adding a trace (`addTrace`), thus requiring the source and target objects to be passed as parameters. Second, for resolving a trace based on a source object named `resolve`. To achieve type consistency `resolve` also requires the class of the intended target object to be passed as parameter. An example of how the trace library is used can be found in line *51* of Listing 8 and will be explained in more detail in Sect. 3.4.

For more advanced trace management, additional methods exist that take an additional String parameter to be able to add and distinguish multiple target objects for a single source object. This functionality is sometimes required to access not the direct target object but another object that was created during the translation of a source object.

3.4 Matched rule translation

Matched rules are translated into two methods within the transformation class. One method is responsible for creating a target object and its corresponding trace link, and one method is responsible for populating the created target object in accordance with the bindings in its corresponding ATL rule. The second method will also incorporate all code corresponding to the imperative code written in the *Action-Block* of the translated rule. As already indicated in Sect. 3.2 when introducing our two-step transformation process, the main idea behind this separation is that all traces and referenced objects can be safely resolved by the second method (called during the second traversal) because they are created by the first method (called during the first traversal). That is, calls for the object and trace creation are put by the `preTransform` method, while calls for the second method are put into the body of the `actualTransform` method.

For the rules `Member2Male` and `Member2Female`, this is illustrated in lines *45* and *50* of Listing 8. The rule `Member2Male` from Listing 7 is translated into the methods `Member2MalePre` (in line *45* of Listing 8) and `Member2Male` (in line *50* of Listing 8). `Member2MalePre` creates an empty `Male` object as well as a trace from the input `Member`, and method `Member2Male` fills the corresponding `Male` object with data as defined through the bindings from the `ATL` rule. To actually perform the transformation on all `Member` objects, the methods `preTransform` and `actualTransform` define for which type of object which method should be executed. This is done using methods from

```
1  private static Female lazyMember2Female(Member
       m) {
2      if (isFemale(m)) {
3          Female t = TRACER.add(m,
       PERSONSFACTORY.createFemale());
4          t.setFullName(m.getFirstName() + " " +
       familyName(m));
5          return t;
6      }
7      return null;
8  }
```

List. 11 Example translated lazy rule.

```
1  private static B uniqueLazyMember2Female(A a) {
2      Female t = TRACER.resolve(m,
       PERSONSFACTORY.createFemale());
3      if (t == null) {
4          if (isFemale(m)) {
5              t.setFullName(m.getFirstName() + "
       " + familyName(m));
6              return t;
7          }
8          return null;
9      }
10     return t;
11 }
```

List. 12 Example translated unique lazy rule.

```
1  rule calledMember2Female(Member m, String name)
       {
2      to
3          t : Female (
4              fullName <- name
5          )
6  }
```

List. 13 Example ATL called rule.

```
1  private static Female
       calledMember2Female(Member m, String name) {
2      Female t = PERSONSFACTORY.createFemale();
3      t.setFullName(name);
4      return t;
5  }
```

List. 14 Example translated called rule.

the traversal library to add the corresponding function calls for the `Member` class as shown in lines *21* and *30* of Listing 8.

A special feature that comes from using our traversal library is that we only need to translate the condition whether a rule should be applied in the pre method that is translated from it. This is because the `traverseAndAccept` method only executes the corresponding function for an object after it verified that an associated target object can be found via a trace. If no target object can be found, the function is not executed. An example of this can be found in the translation of the `Member2Male` rule. Line *32* of Listing 7 states that `Member2Male` is only executed under the condition that `not s.isFemale()`. In the Java code in Listing 8, this is only translated into the `Member2MalePre` method in line *46*, whereas `Member2Male` in line *50* does not contain this condition.

Lazy rules and unique lazy rules do not require as much overhead as matched rules since they are called directly from within other rules/methods and thus do not need to be integrated into the traversal order. However, they do require traces to be created and added to the global tracer. Additionally, methods translated from these types of rules have the target object as their return value rather than the return type being `void`. Suppose `Member2Female` was a lazy matched rule. In that case, instead of the code in lines *21*, *30*, and *59-63* for `Member2Female`, only the code shown in Listing 11 would be added to the `Families2Persons` class. The method `lazyMember2Female` returns an object of type `Female` while also creating a trace from the passed `Member` to the returned `Female`. In case `Member2Female` was a unique lazy matched rule, a precondition using trace links is added to the translated Java code that ensures that the method always returns the same object when called for the same input object. This is illustrated in Listing 12.

3.5 Called rule translation

Called rules, much like lazy rules, can be translated into a single method that creates the output object, populates it in accordance with the bindings of the ATL rule, and then returns it. Other than the methods created for matched rules, the methods for called rules can take more than one parameter as input since called rules in ATL can define an arbitrary amount of parameters of varying types. Moreover, called rules do not create or use trace links. A sample called rule translated into Java can be found in Listings 13 and 14.

3.6 Helper and OCL expression translation

Helpers can be translated into methods much like called rules. The contained OCL expressions can easily be translated into semantically equivalent Java code. Examples of such semantically equivalent translations can be found in lines *9-11* of Listing 8 which correspond to the OCL code in lines *4-17* of Listing 7. One distinction that can be made here is again between the different Java versions used in terms of our study. Streams can be used to simulate the syntax of OCL, in particular the arrow symbol for implicitly navigating over collections, while older Java versions need to use loops instead. Table 2 shows a number of OCL expressions and their Java SE14 counterpart using streams. Note that in contrast to OCL, Java requires all collections to be converted to streams and back to be able to manage them in a func-

Table 2 A selection of OCL expressions translated to Java SE14

OCL	Java SE14
collection->select(e)	collection.stream().filter(e).collect(Collectors.toCollection())
collection->collect(e)	collection.stream().map(x -> e.apply(x)).collect(Collectors.toCollection())
collection->includes(x)	collection.stream().anyMatch(a -> x == a).collect(Collectors.toCollection())
element.attribute	element.getAttribute()
collection.attribute	collection.stream().map(x -> x.getAttribute()).collect(Collectors.toCollection())
i | i > 5	i -> i > 5

tional programming style. The same expressions written in Java SE5 without streams can be found in Listings 24 to 29 in Appendix A.

4 Code classification schema

In this section we introduce the classifications of Java and ATL code used throughout **RQ2** and **RQ3**. The ATL classification described in Sect. 4.1 is taken from [16] and is based on the hierarchical structure of ATL. The classification of Java code described in Sect. 4.2 was developed specifically for the analysis of this research. It is based in the structure of Java code and its components as well as the relation thereof to general transformation aspects and ATL. We will again use the families2persons example to illustrate how the classification schemas are applied.

4.1 ATL

The hierarchy for the ATL classification was already established by Götz and Tichy [16] and consists of the following levels and their corresponding categories:

1. Module Level
2. Rule Type & Helper Level
3. Rule Blocks Level
4. Content Level
5. Binding Level

The aim of this classification system is to differentiate the different components and their contained subcomponents within an ATL module. As such, this classification represents a way to indicate how a syntax element is contained within the complete structure of the ATL code. This allows us to make precise observations on the structure of ATL modules based on their components and, for example, the distribution of number of words required to write each component. An overview of the classification hierarchy can be found in Fig. 2. And the complete labelling for the ATL solution of Families2Persons can be found in Fig. 3.

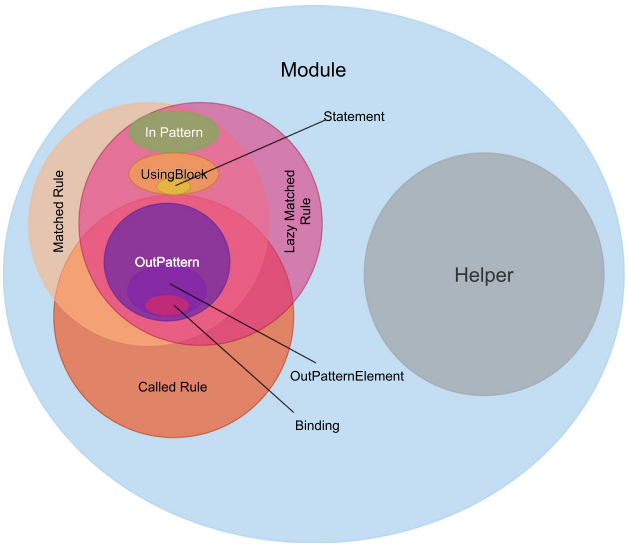


Fig. 2 Overview of the ATL classification from Götz and Tichy ([16]

The **Module Level** defines the belonging of all elements within a module to said module. Below it on the **Rule Type & Helper Level** a distinction between helpers and the different types of rules is made. In the Families2Persons example from Listing 7 and Fig. 3 the helpers in lines *4* & *19* are labelled as *Helper*, while both rules `Member2Male` and `Member2Female` in lines *30* & *39* are labelled as *Matched Rule*. All elements within the rules and helpers again inherit the respective classification for this level from their parent elements.

The **Rule Blocks Level** distinguishes between the different types of blocks that make up rules, i.e. *Using Block*, *OutPattern*, *InPattern*, and *Action Block*. A more specific distinction of helper contents is not done due to them only containing OCL expressions. The rules in the Families2Persons example only contain InPatterns (lines *31-32, 40-41*) and OutPatterns (lines *33-36, 42-45*).

Below the Rule Blocks Level the **Content Level** then allows a more precise description of the elements contained within the rule blocks. The potential classifications on this level are: *OutPatternElement*, *Statement*, and *Vari-*

able Declaration. Lines *43-45* for example are labelled as an *OutPatternElement*.

Lastly, the **Binding Level** again only contains one characteristic and allows to label bindings as exactly that. Lines *35* and *44* are bindings and thus labelled as such as seen in Fig. 3.

4.2 Java

In order to draw parallels between transformation code written in Java and ATL, it is necessary to relate all code components in the Java code to the transformation aspects they implement. For this purpose, we developed a hierarchical classification for Java code. The hierarchy follows the natural structure of Java code much like the classification for ATL. However, contrary to ATL, the code structure of Java does not allow us to directly break it down into transformation-related components. This is due to the fact that Java is focused around object-oriented and imperative components rather than transformation-specific ones. As a result, the classification schema breaks Java code down into its OO and imperative components and then relates those components to transformation aspects. The hierarchy levels of the classification are as follows:

1. Class Level
2. Attribute & Method Level
3. Statement-Type Level
4. ATL Counterpart Level

An overview of the classification levels and the characteristics attributed to each level can be found in Fig. 4. A sample labelling for the Java solution of Families2Persons can be found in Fig. 5.

The **Class Level** stands on top of the hierarchy. The class level itself is made up of only one type of characteristic, the *Class* itself. In the Families2Persons example from Listing 8 the class definition and all elements contained within the class body is thus labelled as belonging to the class characteristic of the Class Level (as seen in Fig. 5). This also indirectly represents a relation between the class and the transformation module from which it was translated from, indicating that the class and all its components relate to the transformation module and its components. More specific relation between the contained components is then described through the lower levels within the classification system.

Below the Class Level lies the **Attribute & Method Level** in which we classify to which transformation aspect an attribute or method is related. The characteristics that can be attributed on this level are: *Traversal* when a method is used for the traversal of the input model. *Transformation* when a method contains code for the actual transformation of one model element to another. *Tracing* for all

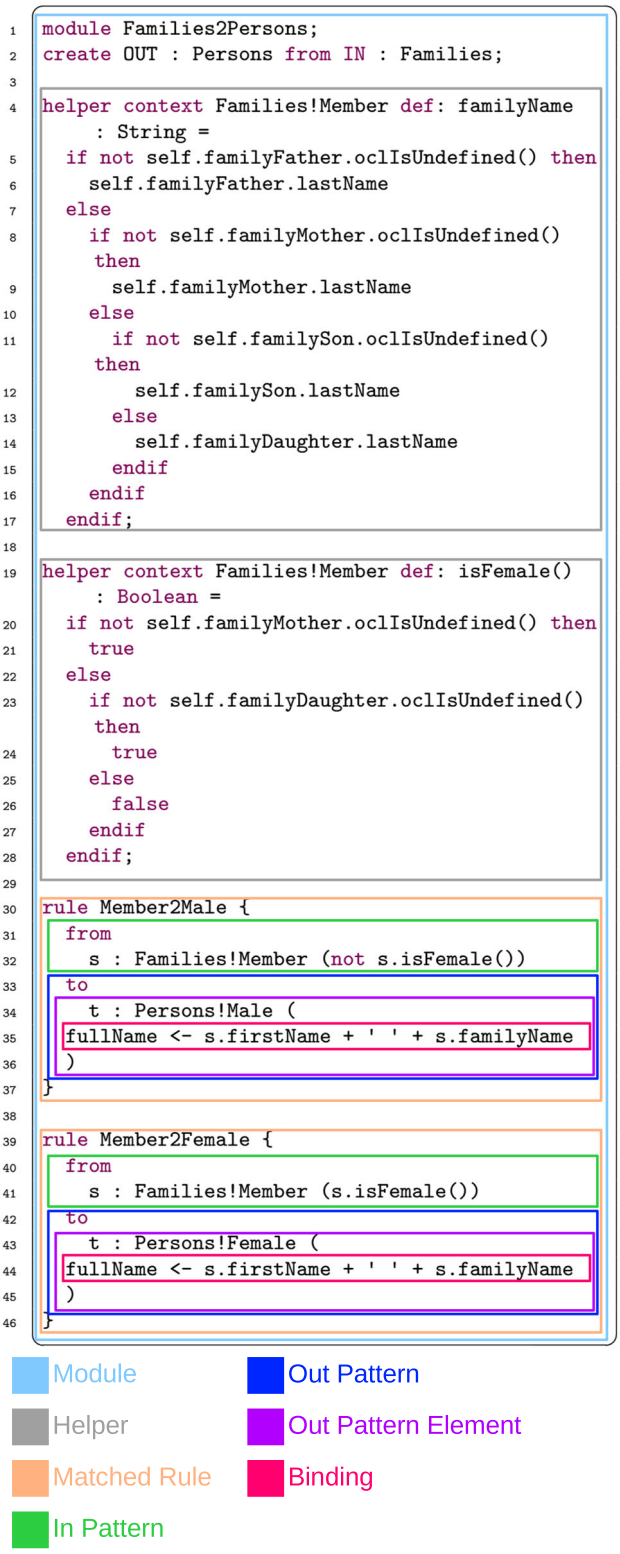


Fig. 3 Labelled ATL solution for the Families2Persons case

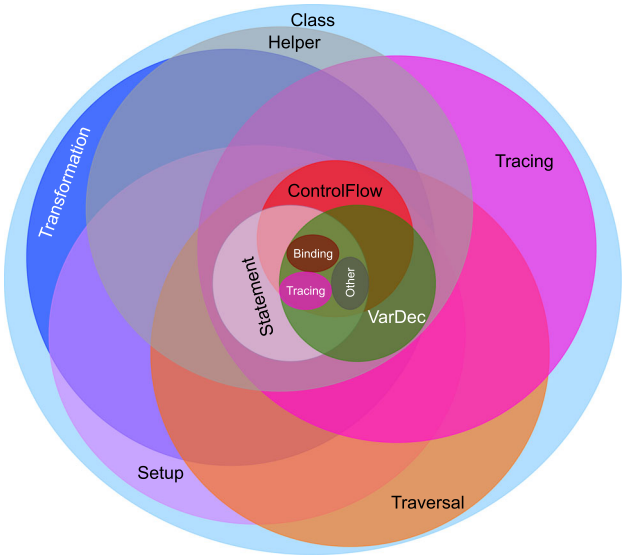


Fig. 4 Overview of the makeup of our Java classification

methods that are related to the creation or resolution of traces. *Helper* when a method corresponds to a helper and lastly *Setup* for all attributes that are required to exist for access throughout the transformation. The `isFemale` method in lines *5-7* from Fig. 5 is thus assigned the label *Helper* for the **Attribute & Method Level** in addition to its *Class* label on the **Class Level**. The `transform`, `preTransform` and `actualTransform` methods all get assigned the *Traversal* label, while `Family2List`, `Member2Male` and `Member2Female` are all labelled as *Transformation* related on the Attribute & Method Level. Lastly, `Member2MalePre` and `Member2FemalePre` both relate to *Tracing* and are thus characterized as such. All statements within the methods again inherit the classification of the Class Level and the Attribute & Method level from their respective parents in which they are contained in and get more specialized again through the lower levels within the system.

Below the Attribute & Method Level then lies the **Statement-Type Level** in which all statements within methods are characterized based on whether they are *Control Flow* statements (i.e. conditions or loops), *Variable Declaration*s or any other type of *Statement*. The categorization on this level does not directly relate to any transformation aspect but rather allows us to differentiate between different types of statements in Java that are relevant for highlighting differences between the structure of Java and ATL code. The condition defined in line *46* of Fig. 5 is labelled as belonging to *Control Flow* on this level while again inheriting its Class Level and Attribute & Method Level from its container Method `Member2MalePre`.

The next lower level is the **ATL counterpart Level**. On this level, we categorize whether a statement fulfils the role of a *Binding* in ATL or if it contains code to create or resolve *Traces* or if it is any *Other* type of Java code that does not directly relate to transformation aspects. At this level, one would expect that the categorization of statements is dependent on the categorization of the **Attribute & Method Level** of the methods they are contained in, i.e. a statement within a *Transformation* method should either be categorized as *Binding* or *Other*. However, in Java transformations these boundaries become somewhat blurred due to the fact that traces need to be explicitly resolved to access the corresponding output model elements when assigning them to output attributes. This can for example be seen for line *51* of Fig. 5. The classification comes from it being a variable declaration that assigns the result of a trace resolution call within a method that performs the transformation of a `Member` into a `Male`.

Lastly, we can also label different parts of a single line with different labels based on their functionality. Line *38* of Fig. 5, for example, has elements that perform assignments, i.e. bindings translated to Java, and elements that perform additional tracing operations. The labelling of this line reflects these different functionalities within the line by labelling sub-statements within the line instead of the whole line.

5 Size and complexity analysis methodology

Our analysis of the transformation specifications is guided by the research questions introduced in Sect. 1.2.

5.1 RQ1: How much can the complexity and size of transformations written in Java SE14 be improved compared to Java SE5?

To compare the transformations written in Java SE14 and Java SE5, we decided to use code measures focused on code complexity and size. For this reason, we chose McCabe's cyclomatic complexity and LOC which are shown to correlate with the complexity and size of software [29]. To keep the LOC count as fair as possible, all Java code was developed by the same researcher and we used the same standard code formatter for all Java code. Furthermore, we supplement LOC with an additional measure for code size based on word count, the combination of these two measures also allowed additional insights. Word count means the number of words that are separated either by whitespaces or other delimiters used in the languages, such as a dot (`.`) and different kinds of parentheses (`()` `[]` `{}`). This measure supplements LOC because it is less influenced by code style and independent from keyword and method name size [18]. This method for calculating transformation code size has already been successfully used by Anjorin, Buchmann, Westfechtel, et al. [18] to compare several (bidirectional) transformation

```java
public class Families2Persons {
  private static final PersonsFactory PERSONSFACTORY = PersonsFactory.eINSTANCE;
  private static final Tracer TRACER = new Tracer();

  private static boolean isFemale(Member member) {
    return member.getFamilyDaughter() != null || member.getFamilyMother() != null;
  }

  private static String familyName(Member member) {
    return ((Family) member.eContainer()).getLastName();
  }

  public static List<Person> transform(Family family) {
    preTransform(family);
    return actualTransform(family);
  }

  private static void preTransform(Family root) {
    var iterator = root.eAllContents();
    var traverser = new Traverser(TRACER);
    traverser.addFunction(Member.class, x -> {Member2MalePre((Member) x);Member2FemalePre((Member) x);});
    traverser.traverseAndAcceptPre(iterator);
  }

  private static List<Person> actualTransform(Family root) {
    var newRoot = Family2List(root);

    var iterator = root.eAllContents();
    var traverser = new Traverser(TRACER);
    traverser.addFunction(Member.class, x -> {Member2Male((Member) x);Member2Female((Member) x);});
    traverser.traverseAndAccept(iterator);

    return newRoot;
  }

  private static List<Person> Family2List(Family root) {
    var persons = new LinkedList<Person>();
    persons.add(TRACER.resolve(root.getFather(), Male.class));
    persons.add(TRACER.resolve(root.getMother(), Female.class));
    persons.addAll(root.getDaughters().stream().map($ -> TRACER.resolve($,
     Female.class)).collect(Collectors.toList()));
    persons.addAll(root.getSons().stream().map($ -> TRACER.resolve($,
     Male.class)).collect(Collectors.toList()));
    return persons;
  }

  private static void Member2MalePre(Member m) {
    if (!isFemale(m)) {
      TRACER.addTrace(m, PERSONSFACTORY.createMale());
    }
  }
  private static void Member2Male(Member m) {
    var t = TRACER.resolve(m, PERSONSFACTORY.createMale());
    t.setFullName(m.getFirstName() + " " + familyName(m));
  }

  private static void Member2FemalePre(Member m) {
    if (isFemale(m)) {
      TRACER.addTrace(m, PERSONSFACTORY.createFemale());
    }
  }
  private static void Member2Female(Member m) {
    var t = TRACER.resolve(m, PERSONSFACTORY.createFemale());
    t.setFullName(m.getFirstName() + " " + familyName(m));
  }
}
```

Class Transformation Helper Variable Declaration
Setup Tracing Traversal Binding Control Flow

Fig. 5 Partially labelled Java solution for the Families2Persons case

languages including eMoflon [30], JTL [31], NMF Synchronizations [32] and their own language BXtend [33]. Their argument for using word count is that because it approximates the number of lexical units it more accurately measures the size of a solution than lines of code.

We applied the Java code metrics calculator (CK) [34] on all 24 transformations (12 Java SE5 + 12 Java SE14) to calculate both metrics and used a program developed by us to calculate the word count measure. For a basic overview we then compare the total size between Java SE5 and Java SE14 based on both LOC and word count and discuss observations as well as possible discrepancies between the two measures. The same is done for McCabe complexity as well. Because CK calculates metrics on the level of *classes*, *methods*, *fields* and *variables* we opted to additionally use the values calculated on the level of *methods*, i.e. the LOC, word count and McCabe complexity of the method bodies, to gain a more detailed understanding of where differences in size and complexity arise from. Since neither the *fields* level nor the *variables* level contained values for McCabe complexity and no interesting values for LOC and word count we decided to omit data from those in our analysis. The metric values calculated by CK were then analysed and compared based on maximum, minimum, median, and average values.

RQ1 serves the purpose of providing a general overview of the differences between the code size and complexity between Java SE5 and Java SE14. The results from this research question are analysed and discussed in more detail in **RQ2&3**.

5.2 RQ2: How is the complexity of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to each other and ATL?

To answer **RQ2**, we compare the distribution of complexity within the Java code with regard to the different steps within the transformation process. In particular, we want to see how much effort needs to be put into writing those aspects that ATL can abstract away from. To be able to analyse the complexity distribution in Java transformations, it is necessary to differentiate the different steps within the Java code, i.e. *model traversal*, *transformation*, *tracing*, *setup* and *helper*. Since cyclomatic complexity can not be calculated for each line but only for set of instructions we decided to fall back on the granularity of methods and use the classification and labelling given to each method in Sect. 4.

Based on the classification introduced in Sect. 4.2, all Java transformations were labelled by one author. The labelling was verified by the other two authors with one of them cross-checking 2 transformations and the other one checking 4. The checked transformations were *istar2archi*, *Palladio2UML*, and *R2ML2XML* all in both Java SE5 and Java SE14 which in total meant that about 51% of the total Java code lines were reviewed.

We then used the measures calculated for **RQ1** to create plots of the complexity distribution. The distribution shown in the resulting plots was then analysed taking into account the results of Götz and Tichy [16] regarding the distribution of different transformation aspects in ATL. The goal in this step was to see how the complexity in Java transformations is distributed onto transformation aspects, such as tracing and input model traversal, that are abstracted or hidden away in ATL as well as to see the evolution of this distribution between the two different Java versions.

5.3 RQ3: How is the size of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to each other and ATL?

The approach for this research question is twofold and follows a top down methodology. First, we compare the distribution of code size within the Java code over the different transformation aspects using the classification from Sect. 4. Afterwards, we focus on the actual code. Here, we compare how code written in ATL compares to the Java code that represents the same aspect within a transformation.

We opted to use word count as a measure for the detailed discussion of code size. The reason why we use word count and not lines of code lies in their granularity. For some parts of our analysis, it is necessary to split the value of single statements up into that of their components. This is much easier to do when using word count as a measure and does not require code to be rewritten in an unintuitive way. Moreover, the finer granularity also allows a more detailed look into the structure of methods that was not possible in **RQ2** due to the limitation of cyclomatic complexity.

The idea behind our approach is to calculate the word count for all transformations written in Java and ATL and then compare both the total count of words as well as the number of words required for specific aspects within the transformation process. While the word count for Java transformations is calculated specifically for this study, the data for the ATL transformations are taken from the results of Götz and Tichy [16].

```
1 rule SimpleBinding {
2     from s : Member
3     to t : Female (
4         name <- s.firstName
5     )
6 }
```

List. 15 A rule with a simple binding.

Based on the introduced categorizations, we then create Sankey diagrams for the distribution of word count in both

```
rule Trace {
    from s : Member
    to t : Male (
        father <- s.familyFather
    )
}
```

List. 16 A rule with a binding using traces.

```
helper context Class def: associations:
    Sequence(Association) =
    Association.allInstances() ->
    select(asso | asso.value = 1);
```

List. 17 A typical helper in ATL.

Java and ATL. These graphs then form the basis for our comparison. Here, we compare both the distributions of the individual transformation aspects in Java with ATL as well as the concrete sizes on the basis of the numbers. When comparing the size distribution, we analyse how the distribution of the transformation aspects in Java differs from ATL, i.e. which aspects are disproportionally large or small compared to ATL. We also explicitly look at how much code is required for tracing in Java. For this, we look at the proportion of the transformations that require traces and how that compares to the total size of Java code related to traces. Lastly, the total number of words between Java and ATL are also directly compared to see which language allows for shorter transformation code based on this measure.

To illustrate where the observed effects originate from, we use a selection of three ATL fragments representing code which is often written in ATL transformations. The first fragment (see Listing 15) represents code that copies the value of an input attribute to an attribute of the resulting output model element, an action which constitutes 56% of all bindings in the set analysed by [16]. The second fragment (see Listing 16) represents code that requires ATL to use traceability links, which [16] found to constitute 15% of all bindings. Because the attribute `s.familyFather` does not contain a primitive data type, but a reference to another element within the source model, the contained value cannot simply be copied to the output element. Instead, ATL needs to follow the traceability link created for the referenced input element to find its corresponding output element which can then be referenced in the model element created from `s`. The last code fragment (see Listing 17) is a helper definition of average size and complexity.

We use those code fragments and compare them with the Java code that they are translated to in order to highlight differences between the languages.

5.4 RQ4: How does the size of query aspects of transformations written in Java SE5 & SE14 compare to each other and ATL?

As previously discussed, the goal of this research question is to investigate the claim that writing queries for models was improved with the introduction of model transformation languages such as ATL and to check if this is still the case when utilizing new languages features in general purpose languages today. This discussion of Java vs OCL has already been raised approaches to replace OCL with Java [35]. The data basis for this analysis is formed by all helpers and their corresponding Java translations in form of methods within the 12 transformation modules subject in this study. Because this set only contains a total of 15 helpers, we complement it with a large collection of helpers and their translations from a set of supplemental libraries used in the *UML2Measure* transformation.

In our analysis, we compare Java and ATL helpers first based on their total word count and then by contrasting each ATL helper with its Java counterpart using regression analysis. All observations in this analysis are supplemented with code segments that highlight them. The regression analysis uses a linear regression model to predict the word count of Java methods ($J5WC$, $J14WC$) based on the word count of ATL Helpers ($HelperWC$). This was chosen based on an hypothesis that Java code entails an additional fix cost compared to OCL expressions as well as an increase by some factor due to the more verbose syntax of Java. This approach allows us to both verify the hypothesis and identify an approximation of the interrelationship between the code sizes.

6 Results

In this section, we present the results of our analysis in accordance with the research questions from Sect. 1.

6.1 RQ1: How much can the complexity and size of transformations written in Java SE14 be improved compared to Java SE5?

Table 3 presents an overview of lines of code (LoC), word count (# words) and the sum of McCabe complexities of all methods contained in the transformation classes (WMC). Looking at the total lines of code and WMC, the numbers display an expected decrease in both size and complexity. Our transformations written in Java SE5 total 3252 lines of code and have a WMC of 792. The same transformations written in Java SE14 require only 2425 lines of code and have a WMC of 411. Based on these measures, the size reduces by about 25%, while the cyclomatic complexity is cut in half to

about 52% of its Java SE5 counterpart. This decreased WMC can be attributed to the improvements made through utilizing streams for handling collections. The traversal library also contributes to this by removing all control flow branching for the `transform` methods and thus reducing the McCabe complexity of these methods.

The word count measure, however, shows a different picture. While the Java SE5 implementation uses 13007 words, the Java SE14 implementations use nearly the same amount of words, 13118 to be exact. When combining this with the reduced number of code lines provides and interesting observation. Transformation code written in Java SE14 for our transformation set is more dense, i.e. a single line of code contains a lot more words and thus more information about the transformation.

Overall, both the total number of lines of code as well as the WMC of transformations in the newer Java version are greatly reduced. However, there is no notable change in the number of required words, which hints at a more information-dense code rather than simply less code.

Table 4 summarises the calculated size (LOC and word count) and complexity (McCabe) measurements on the method level for both the Java SE5 and Java SE14 transformation code.

As expected from the total numbers, the average and median length, measured in LoC, of methods in Java SE14 is reduced by about 30%. The already low minimum of 3 lines has not been further reduced in the newer version, but the longest method is now 51 lines shorter.

Contrasting the numbers for lines of code with word count, we see a small increase in both the average and median method sizes in Java SE14 compared to Java SE5. However, the maximum number of words for a method is about 43% shorter in Java SE14 than in Java SE5. This means that while on average (or median) the number of words required to implement transformation-related methods in Java SE14 increased compared to Java SE5, newer Java versions help to reduce the size of methods that required large number of words in older Java versions.

The reduction in cyclomatic complexity seen in the total numbers is also reflected for the more detailed consideration on method level. The average transformations written in Java SE14 are 45% less complex than in Java SE5. A result also reflected in the median. Furthermore, the maximum McCabe complexity is reduced from 44 to 11, which is a significant decrease as this suggests that even highly complex methods within the transformations can be expressed a lot less complex in newer Java versions. This, again, can be attributed to the utilization of streams and functional interfaces which help to remove the requirement to manually implement large amounts of loops and nested conditions.

The more detailed results reflect what was already shown on a coarse-grained level. Compared to Java SE5, new language features in Java SE14 help to reduce the required number of code lines, while the number of words stays about the same. The cyclomatic complexity is significantly reduced, most prominently seen in the fact that the most complex method in Java SE14 is only 1/4th of the complexity of the most complex method in Java SE5.

6.2 RQ2: How is the complexity of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to ATL?

The results for this research question are split up into two parts. We first report on our findings for Java SE5 and its comparison to ATL in Sect. 6.2.1, before reporting the findings for Java SE14 and its comparison to ATL and Java SE5 in Sect. 6.2.2.

6.2.1 Java SE5

Figure 6 shows a plot over the distribution of WMC split up into the different transformation aspects involved in a transformation written in Java SE5 and Java SE14. It shows that about 60% of the complexity involved in writing a transformation in Java SE5 stems from the actual code representing the transformations and helpers. The other 40% are distributed among the model traversal, tracing, and setup code. In ATL, these three aspects are completely hidden behind ATL's syntax. In other words, this means that 40% of the complexity within the transformations written in Java SE5 stems from overhead code.

Overall, the results support the consensus from back when ATL was introduced that a significant portion of complexity can be avoided when using a dedicated MTL for writing model transformations.

6.2.2 Java SE14

Given the observations from **RQ1** combined with the general improvements that Java SE14 brings to the translation scheme, one would expect better results for the complexity distribution of transformations written in that Java version. However, when looking in Fig. 6, which again shows a plot over the distribution of McCabe complexity split up into the different transformation aspects involved in a transformation written in Java, there is still a significant portion of complexity associated with the model traversal, tracing, and setup code in Java SE14.

While the complexity associated with model traversal is greatly reduced by the use of the traversal library, the overall

Table 3 Measurement data on the translated transformation modules

Transformation Name	LOC		# words		WMC	
	Java SE5	Java SE14	Java SE5	Java SE14	Java SE5	Java SE14
ATL2BindingDebugger	22	19	93	88	4	2
ATL2Tracer	74	17	285	283	7	5
DDSM2TOSCA	509	339	2137	2036	103	44
ExtendedPN2ClassicalPN	147	107	569	553	37	19
Families2Persons	72	62	273	297	22	14
istart2archi	184	115	689	714	57	24
Modelodatos2FormHTML	215	178	761	750	58	40
Palladio2UML	303	253	1066	1100	70	47
R2ML2XML	1181	855	4720	4966	303	139
ResourcePN2ResourceM	99	67	380	389	29	13
SimpleClass2RDBMS	163	111	629	581	50	26
UML22Measure	283	249	1405	1356	52	38
Total	3252	2425	13007	13118	792	411
Median	173.5	113	599	647	51	25
Average	271	202.1	1088.9	1092.75	66	34.25

Table 4 Measurement data on the methods in the translated transformation modules

Measure	Minimum		Median		Average		Maximum	
	Java SE5	Java SE14	Java SE5	Java SE14	Java SE5	Java SE14	Java SE5	Java SE14
LoC	3	3	7	6	12.5	9.4	135	105
# words	1	2	5	6	5.2	6.4	64	37
McCabe complexity	1	1	2	1	3	1.6	44	11

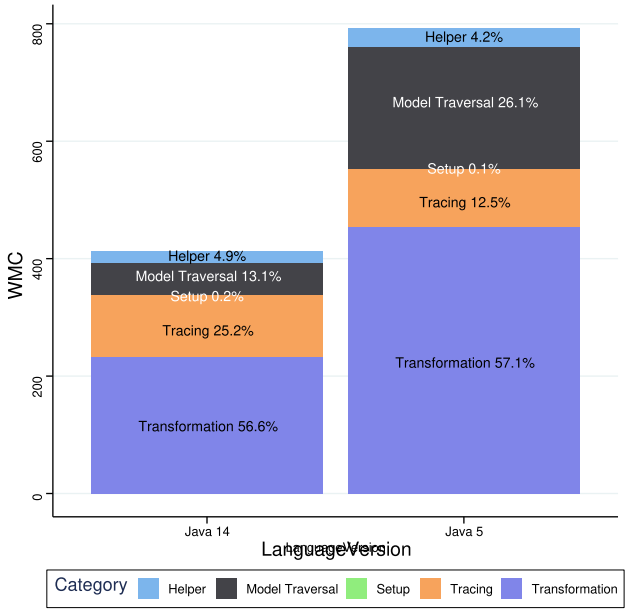
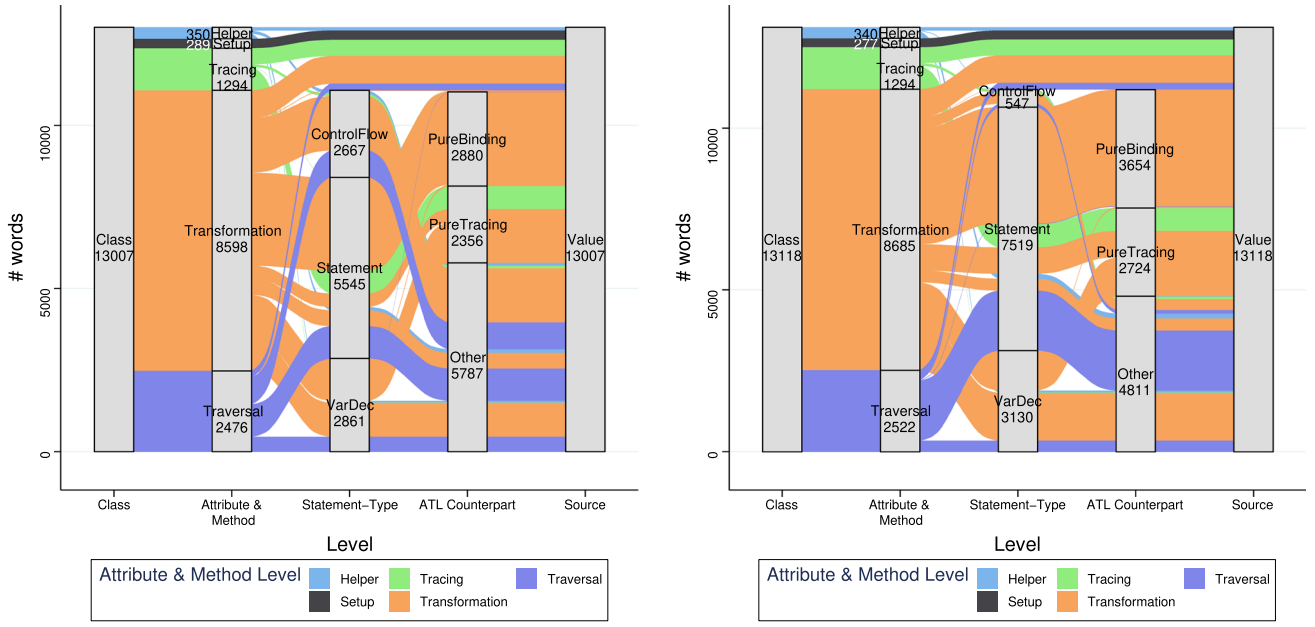


Fig. 6 Distribution of WMC over transformation aspects in Java SE5 and SE14

distribution between the actual code representing the transformations and helpers and the model traversal, tracing, and setup code does not change much. About 40% of the overall transformation specification complexity still stems from overhead code. Moreover, not only did this ratio stay similar compared to Java SE5, also the ratio between helper code complexity and transformation code complexity stayed about the same. One potential reason for this is that while newer Java features help to reduce complexity, they do so for all aspects of the transformation, thus the distribution stays about the same.

The reason that the code related to trace management experiences an increase in its complexity ratio compared to other parts of the transformation can be explained by the fact that this code stayed the same between the different Java versions. Thus, while the complexity of all other components shrank, the complexity of trace management methods stayed the same, leading to higher relative complexity.

Overall, the results point towards even newer versions of Java still having to deal with the complexity overhead that ATL is able to hide. Specifically, handling traces still entails a large overhead.



(a) Distribution in Java SE5.

(b) Distribution in Java SE14.

Fig. 7 Distribution of word count over transformation aspects in Java SE5 and SE14

6.3 RQ3: How is the size of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to ATL?

The reporting of results for this research question follows the same structure as Sect. 6.2. First in Sect. 6.3.1 the results of our analysis of Java SE5 and its comparison with ATL are reported. Afterwards in Sect. 6.3.2 the results for Java SE14 and its comparison with ATL are discussed. This section also contains a comparison to the results of Java SE5.

6.3.1 Java SE5

The total size of Java SE5 transformations compared to ATL transformations is much larger when using word count as a measure. All ATL transformations in our set together amount to 7890 words, while the Java SE5 code needs 13007. This is an increase of 64.8%. Figure 7a allows us to look at the distribution of written words over the transformation aspects introduced in Sect. 4. The x-axis of the graph describes the hierarchy levels from Sect. 4. The word count is depicted on the y-axis, and on each hierarchy level on the x-axis the word count distribution of its different aspects is shown. How each level is made up of its sub-levels is then shown by means of the alluvial lines flowing from left to right. The flow lines are coloured according to the Attribute & Method level as it represents the top level of separation and eases readability.

Looking at the graph we see a large portion of the number of words is actually associated with the transformation code itself. Overhead from tracing, traversal, and setup exists, but it is not as prevalent as expected from the results presented in Sect. 6.3. However looking more closely into each of the aspects and their makeup reveals that there is more overhead still hidden in the transformation-related code. In the following, we will look at the individual aspects and their more precise breakdown and what this means for transformations written in Java SE5, also in comparison with ATL.

The number of words required to express Helper code for our transformation set is low. It constitutes 2.9% of all words within the transformation class which is in line with the size of helpers in ATL as seen in Fig. 8.

Similarly, the number of words required for setup code is also of little consequence as it constitutes only about 2.2% of the total word count in the transformations considered in this work. However, even though the amount is small, the code still has to be written and maintained when evolving the transformation.

Another part of the code within the transformation classes that represents overhead in Java SE5 compared to ATL is the code related to tracing. While ATL abstracts away tracing and does target element creation implicitly, in Java this behaviour has to be recreated by hand. The library for tracing introduced in Sect. 3.3 helps reduce the implied overhead, but the creation of target objects as well as traces for them still has to be initiated manually. The methods involved in this constitute for 9.9% of words used in our translated transformations and

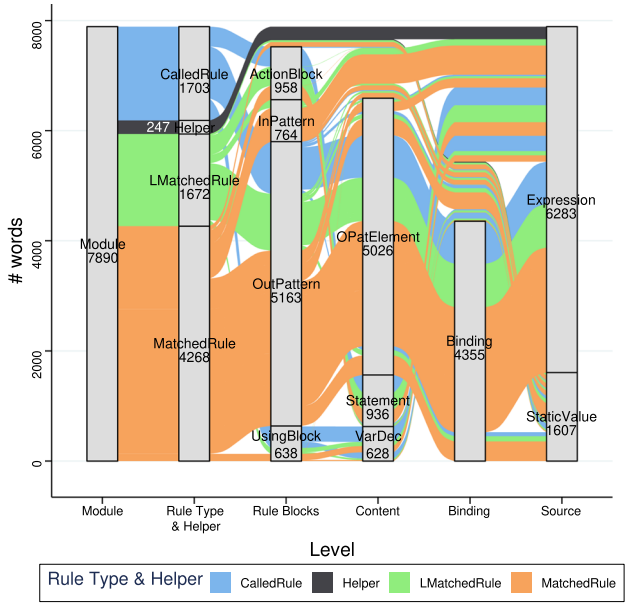


Fig. 8 Distribution of word count "complexity" measure over transformation aspects in ATL calculated based on Götz and Tichy[16]

are made up of methods in style of what is described in Sect. 3.4.

As previously stated, a large portion (65.8%) of the word count comes from methods and attributes related to the actual transformation. This however changes when looking at the lower levels of classification within those methods. In ATL 60% of the total number of words and 61% of the words within rules stem from bindings, i.e. the core part responsible for transforming input into output. In our Java SE5 translation this differs greatly. The translated binding code only makes up 22% of the total word count or 33.5% within the transformation methods. This points to the fact that much less of what is written in Java SE5 actually relates to actual transformation activities. In Java many more words are spent on code not directly transformation-related but rather on tasks necessary for the transformation to work. Three such types of code stand out.

One is statements that resolve traces built up in the tracing methods discussed in the last section (as seen by the flow from Transformation over Statement and Variable Declarations towards Tracing in Fig. 7a). Examples of such code in the Families2Persons example from Fig. 5 and Listing 8 can be found in lines *38,39* and *51*.

The second one is code to initialise temporary variables used for processing steps within the transformation (as seen by the flow from Transformation over Variable Declarations into Other in Figure 7a).

And lastly there are a large number of words associated with control flow via loops and conditions to process collections in order to bind their transformed contents onto attributes of the current output object (as seen by the flow

```
1  private void simpleBinding(Member s) {
2      ...
3      t.setName(s.getFirstName());
4  }
```

List. 18 A rule with a simple binding in Java SE5.

```
1  private void simpleBinding(Member s) {
2      ...
3      t.setName(TRACER.resolve(
   s.familyFather, Male.class));
4  }
```

List. 19 A rule with a binding using traces in Java SE5.

from Transformation over Control Flow towards Other in Fig. 7a).

Code relating to traversal is again overhead introduced due to the usage of Java over ATL. The number of words required for writing traversal-related code for our set of transformation constitutes 18.9% of the total word count of transformation classes.

Overall, the overhead produced by Tracing, Traversal and Setup code amounts to 31% of the total number of words for our Java SE5 transformations. Furthermore, while 65.8% of words within the transformation classes are related to the process of transformation, many of them are again overhead from manual trace resolving, model traversal and supplemental code.

When comparing a simple binding (see Listing 15) written in ATL with its translation in Java SE5 (see Listing 18), there is not much difference. Both require nothing more than their language constructs for accessing attribute values and assigning them to a different attribute.

This is not the case when traces are involved. While ATL allows developers to treat source elements as if they were their translated target element (see Listing 16), some explicit code needs to be written in Java (see Listing 19). As a result, the transformation specification gets larger since it is not only required to call the trace resolution functionality, but it is also necessary to put some additional type information in so the Java compiler can handle the resulting object correctly. The type information is necessary since, as described in Sect. 3.3.3, the trace library holds `EObjects` which have to be converted to the correct type after they have been retrieved based on the source object.

The increase in size is even more prevalent when looking at the translation of a typical helper. The helper in Listing 17 requires OCL code that works with collections which, thanks to OCL's "→ syntax", can be expressed in a concise manner. In Java SE5, however, as seen in Listing 20, the code gets a

```
1 private List<Association>
      associations(Class self) {
2     List<Association> list = new
      LinkedList<Association>();
3     for (Association asso :
      ALLASSOCIATIONS) {
4         if (asso.getValue() == 1) {
5             list.add(asso);
6         }
7     }
8     return list;
9 }
```

List. 20 A typical helper in Java SE5.

lot more complex and bloated. This is due to, as previously stated in Sect. 6.3, the fact that the only way to implement the selection is to iterate over the collection through an explicit loop (lines 3 to 9) and to use an if-condition within the loop (lines 4 to 6). We investigate and discuss this in more detail later in Sect. 6.4.

Overall, the examples show that simple bindings can be expressed easily in both ATL and Java SE5. Bindings involving trace resolution require some additional effort in Java SE5 while ATL can handle those like any other binding. The most significant difference, however, comes from expressions involving collections. Due to the required usage of explicit loops, the Java SE5 code blows up in size and complexity compared to the more compact ATL notation.

6.3.2 Java SE14

Comparing the total number of words in Java SE14 transformations with ATL, a similar picture as for Java SE5 arises. The translated transformations require 13118 words, while ATL only requires 7890. Surprisingly, as also discussed in Sect. 6.1, the number of words in Java SE14 is higher than that of Java SE5, although only by around 100 words, despite requiring less lines of code and cyclotomic complexity. We believe this to be the result of two effects. One, using streams for processing collections reduces the lines of code and cyclomatic complexity because they are single statements and are thus not split over as many lines as when using loops. But, setting up streams and transforming them back into the original collection requires several additional method calls which offset the overall reduction of number of words.

The distribution of the number of words between Java SE5 and Java SE14 also differs immensely, especially around the makeup of transformation methods, as evident from Fig. 7b. It also again highlights key differences between the ATL transformations and their Java counterparts.

```
1 for (InElement i : input.getInElements()) {
2     output.getOutElements()
3       .add(TRACER.resolve(i, OutElement.class));
4 }
```

List. 21 Trace resolution example of a collection in Java SE5.

The portion of words required for writing Setup and Helper code has slightly reduced compared to Java SE5, while the proportion of words for Transformation and Traversal methods increased. The Methods & Attributes for setting up helpers does not change which is due to the fact that the underlying code does not change between Java SE5 and Java SE14.

Thus, more can be concluded from how the number of words are distributed within the Transformation and Traversal methods in Java SE14.

For Traversal, it is noticeable that almost no control flow statements are used any more. Instead, most words now come from simple statements. This is because in Java SE14 we make use of the Traversal library, which allows us to pass only the classes to be matched and the methods to be called to the traverser instead of having to write loops and conditions manually. This evidently does not reduce the number of words, but it creates a different way of defining traversal.

Similarly, the transformation-related methods in Java SE14 also contain much less words that define control flow. The number of words for other statements not directly performing transformation tasks is also reduced. Instead, the translated bindings now make up a larger proportion of the word count. In our Java SE14 transformations, the code for translated bindings now makes up 27.8% of all words compared to the 22% in Java SE5 and 41.9% of words within the transformation methods. This stems from the usage of streams for processing collections of input elements rather than explicit loops and conditions. As a result the Java SE14 implementation is less control flow driven and focuses more on the data involved. However, while this allows for less lines of code and a reduction in cyclomatic complexity as shown in Sect. 6.1, it does not improve the required number of words. This is because in some cases, the setup overhead for streams counteracts their conciseness gain when using number of words as a measure. An example of this can be seen when comparing Listings 21 and 22. Both code segments resolve all `InElements` from the input into their corresponding `OutElements` and add them to the `OutElements` list of the output. The number of words required in Java SE5 for this totals 14, whereas the number of words in Java SE14 amounts to 17.

```
output.getOutElements()
 .addAll(input.getInElements().stream()
   .map(i -> TRACER.resolve(i, OutElement.class))
   .collect(Collectors.toList()));
```

List. 22 Trace resolution example of a collection in Java SE14.

Overall, our translated transformations in Java SE14 do not reduce the number of words compared to their Java SE5 counterpart. Newer language features do however help in reducing the amount of explicit control flow statements and supplemental code required. Most of this is now done directly in translated bindings which more closely follows the ATL-style. In this sense, Java SE14 helps to take a more data-oriented approach to transformation development compared to Java SE5. However, there is still much overhead from manual traversal, tracing and supplemental code compared to ATL.

When comparing the code segments for writing simple bindings and bindings involving traces in Java SE14 with ATL, there is no difference to the findings from comparing Java SE5 to ATL. This is due to the fact that no Java features introduced since SE5 help in reducing the complexity of code that needs to be written here.

```
private List<Association>
    associations(Class self) {
    return ALLASSOCIATIONS.stream()
    .filter(asso -> asso.getValue()==1)
    .collect(Collectors.toList());
}
```

List. 23 A typical helper in Java SE14.

Comparing translated helper code, however, does show some improvements of Java SE14 over Java SE5. Because of the introduction of the streams API, Java SE14 (see Listing 23) can now handle expressions involving collections nearly as seamless as ATL (see Listing 17). Only the overhead of calling `stream()` and `.collect(Collectors.toList())` remains. This and other observations regarding OCL expressions translated to Java are discussed in more detail later in Sect. 6.4.

Overall, the examples show that code for both simple bindings and bindings involving traces in Java SE14 stays just as complex in comparison to ATL as in Java SE5. Code involving collections, however, can now be expressed nearly as seamless as in ATL due to the introduction of the streams API in Java which offers a notation that is close to OCL notation.

6.4 RQ4: How does the size of query aspects of transformations written in Java SE5 & SE14 compare to each other and ATL?

Comparing the word count numbers of helpers from the transformation modules and libraries with their translated counterparts we can once again observe an increase in Java. While all helpers in ATL combined total 2299 words the Java SE5 code totals 3801 words which is an increase of about 65.3%. This was to be expected since Java SE5 is more verbose, especially when handling collections which are required for all helpers within the libraries. This becomes clear when looking at the Java SE5 translation of Listing 17 in Listing 20. Not only does Java require a loop and if-condition to filter out the desired association subset, a new results list also has to be created and filled with values. Compared to OCLs "→ syntax" this increases the number of required words to produce the same result drastically.

Next, as described in Sect. 5.4 a linear regression was calculated to predict the word count of Java SE5 code for Helpers based on their word count. We were able to find a significant regression model ($p < 2.2e-16$) with an adjusted R^2 of 0.649. The predicted word count of Java SE5 expressions for OCL expressions is estimated as 4.85364 + 1.31554 * $HelperWC$. The hypothesis of a linear relationship is also supported by a Pearson coefficient of 0.81 indicating this linear relationship.

Overall, we see a linear relationship between OCL expression code and the translated Java SE5 code. The factor with which the Java code increases in size more quickly is 1.53. This combined with the subjectively less clear way of handling collections through loops leads to the observation that Java5 was not well suited for defining expressions on models.

Looking at the number of words of Java SE14 Helpers compared with their ATL counterparts we see a similar but slightly smaller size than with Java SE5. As stated earlier all ATL library helpers total 2299 words and with 3350 words their Java SE14 counterpart is only about 45.8% larger compared to the 65.3% of Java SE5. This fits well into our observation that the verbose handling of collections is responsible for large portions of the size increase. The streams API, introduced in Java SE8, allows developers a less verbose way of handling collections as can be seen when comparing Listings 20 and 23. While there is still some overhead compared to the OCL counterpart, namely the necessary calls to `stream()` and `.collect(Collectors.toList())`, the total overhead is greatly reduced. Moreover, this difference could in principal be eliminated by using an alternative GPL. The Scala programming language, for example, does not require a conversion between streams and collections.

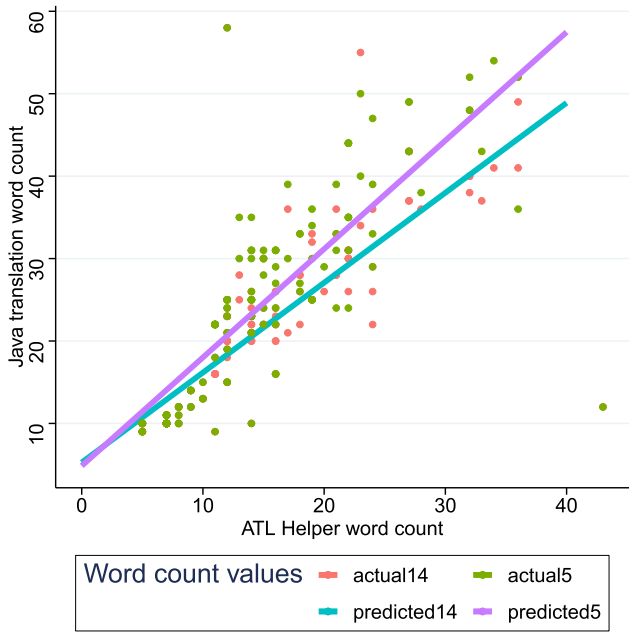


Fig. 9 Comparison of actual Java SE5 and SE14 helper size with predicted size based on linear the regression models

The decrease in size can also be observed in our linear regression model that predicts the word count of Java SE14 code for OCL expressions based on the word count of those expressions. The model we were able to find is significant ($p < 2.2e{-}16$) and has an adjusted R^2 of 0.64. The predicted word count of Java SE14 expressions for OCL expressions is estimated as 5.26631 + 1.09064 * $HelperWC$. And the hypothesis of a linear relationship is again supported by a Pearson coefficient of 0.8. Figure 9 shows how well both the regression models fit the data. It also highlights the decrease of words required for translated helpers in Java SE14 compared to Java SE5.

The x-axis depicts the word count value of OCL expressions, while the y-axis depicts the word count of Java SE5 codes. The dots within the graph then show the corresponding Java SE5 code word count for each translated OCL expression. Lastly, the red line shows the predicted correspondence based on our regression model.

Overall, we still see a linear relationship between OCL expression code and the translated Java SE14 code. However, the factor with which the Java code increases in size more quickly is only approximately 1.1. This leads us to believe that a well trained Java developer should be able to express OCL queries in Java without much difficulties.

7 Discussion

In this section we discuss our findings from Sect. 6 as well as our experiences from the process of translating and using transformations in Java. Our discussion revolves around two main topics. First, we want to discuss the impact that the design decision to not use anonymous classes to outsource traversal in our Java SE5 solution, explained in Sect. 3.3.2, has on the presented data. Then we discuss how the advancements that have been achieved in newer Java versions influence the ability for developers to efficiently develop transformations in Java. This also includes a conversation about what shortcomings still exist. And second we present a guide that suggests in what cases general purpose languages such as Java can be used in place of ATL. We also show cases where we would advise against writing transformations in Java because of its disadvantages. The argumentation of this part is based on the results presented in this publication as well as our experiences, both from this study as well as previous works [8–12]. Finally, we want to have a short discussion beyond the results of our study. Here we want to talk about other features that MTLs can provide and what those could mean for the comparison of MTLs vs. GPLs.

7.1 The impact of not outsourcing model traversal in Java SE 5

As explained in Sect. 3.3.2, we decided on using the conditional dispatcher pattern to implement traversal in our Java SE5 solution as opposed to implementing a traversal library, similar to the one used in Java SE14, using anonymous classes. This design decision has implications for the data presented throughout Sect. 6 which we discuss here.

As mentioned, using the presented approach leads to an increased McCabe complexity for the traversal implementation in Java SE5, while it reduces the LOC and number of words. This has concrete implications for the numbers discussed in Sections 6.1 to 6.3.

For one, this means that when comparing the concrete numbers as done in Sect. 6.1, the stagnation of number of words observed between the Java SE 5 and Java SE 14 variants, would not be present with the alternative Java SE 5 implementation. This is because it would be 812 words longer (making the total number of words 13819) than the presented implementation and thus one would instead observe the expected decrease in number of words in the Java SE 14 implementation. It would still not be as significant, because only the traversal part of all transformations are affected, but it would be more in line with the reduction in code size observed with the LOC measure in the presented implementations. Moreover, the LOC reduction itself would also be more pronounced because the alternative traversal implementation does require more lines of code per rule.

Specifically the total of the Java SE 5 implementation would be increased by 1020 LOC to a total of 4272 as opposed to 3252.

The difference in WMC between the Java SE 5 and Java SE 14 implementation on the other hand would be less clear-cut. As shown in Fig. 6 a significant portion of the WMC in the presented Java SE5 implementation stems from model traversal. In the alternative implementation this complexity would be significantly reduced by 152 to a total of 640 as opposed to 792. The overall WMC of the Java SE 5 transformations would still be higher, because the utilisation of streams in Java SE 14 reduces the McCabe complexity of other parts of the transformation as well, but it would no longer be nearly halved.

Our observations regarding the differences between implementations in the two different Java versions would, however, not change significantly with the alternative Java SE 5 implementation. Thanks to the functional interfaces and streams, in newer Java versions, a more declarative style for defining transformations can still utilised. The WMC of the code is also still reduced, and the general focus can be directed a more towards the actual transformation aspects. In addition, the observations regarding the comparison of Java and ATL do not change.

7.2 Language advancements and their influence on the ability to write transformations: a historical perspective

The overall number of words required to write transformations in Java SE14 compared to Java SE5 has not reduced, as shown in Sections 6.1 and 6.3. However, we have also seen that less explicit control flow needs to be written and the focus shifts more to the binding expressions. This shows in the results discussed in Sections 6.1 and 6.2 as the cyclomatic complexity of transformations written in Java SE14 is greatly reduced. In principle, a shift towards more data-driven development of transformations is therefore possible. Whether this brings an overall advantage or not is still a debated topic [2] and in our eyes depends on the experience and preference of the developers. However, there are many studies in the field of object-oriented programming that establish a connection between cyclomatic complexity and reliability [36–40], i.e. fault-proneness and error rate, as well as some that establish a connection between cyclomatic complexity and maintainability [41,42], i.e. change frequency and change size.

It has been our experience that newer Java features such as streams and the functional interfaces make the development process easier because less work has to be put into building the traversal, and the assignments within the transformation methods are now a more prominent part of them, i.e. they are less hidden in loops and conditions. Whether these advancements justify writing transformations in Java compared to ATL is discussed in the next section.

7.3 A guideline for when and when not to use Java or similar GPLs

As shown in Sections 6.2 and 6.3, while newer Java features shift the focus more towards a transformation-centric development, there is still significant overhead from setup, manual traversal, and especially tracing. Of those three, we believe the setup overhead to be of least relevance. That is because the total overhead for setup is small and it is only an initial overhead that, for the most part, does not need to be maintained throughout the lifecycle of a transformation. The situation is similar for traversal overhead. The code required to be added for all rules or transformation methods, while more significant in its size still only needs to be written once and can be ignored for most of the remaining development. There is little to no room for errors to be introduced , in any Java implementation that follows a style similar to our implementations, as each new rule requires nearly identical code to be added.

Tracing is where, in our opinion, most of the difficult overhead arises from. It is thus the main argument for writing transformations in ATL or similar MTLs compared to general purpose languages. Managing traces and implementing their complete semantics cannot be outsourced into a library, but we can only use a library to reduce the required effort. For many of the advanced use cases, the mapping semantic relies on String constants that are passed to both the creation and resolution methods, which is error-prone. Such cases arise when traces to objects are needed that were only a side effect of a transformation rule and not its primary output.

There is also little support through type-checking since the only way to store traces for all elements is to use the most generic type possible (i.e. `EObject`). This results in the burden of creating and fetching objects of the correct type to be shifted to the developer , which constitutes a clear disadvantage compared to ATL, where trace resolution is type-safe. In simple cases, this problem is less conspicuous, but in cases where advanced tracing is required, much of the described difficulties arise and can lead to errors that are hard to track to its origin. It also forces developers to be more aware of all parts of the transformation at all times, to make sure not to miss any possible object types that could be returned from resolving a trace. There are approaches, such as Goldschmidt, et al. [43], that bring type safety to GPL transformations, but they also come with their own set of limitations when considering advanced features such as incrementality and reusability of the introduced templates, that developers need to be aware of, as well as other boilerplate code that is required to set it up.

Based on the presented reflection, we believe that general purpose languages largely excel in transformations where little to no tracing and especially no advanced tracing is required. The overhead for setup and traversal is manageable in these cases. Moreover, when no traces are required for the transformation, we can scrap the two-phase mechanism completely and thus half the total overhead of traversal is required.

There is also an argument to be made about the expressiveness of Java for complex algorithms compared to the limited capabilities of OCL. We were faced with such a concrete case during the development of a model differencing tool called SiLift [8]. SiLift takes a so-called difference model as input and aims at lifting the given input to a higher level of abstraction by applying in-place transformations to group together interrelated changes. To achieve this low-level changes comprised by the given difference model are first grouped to so-called semantic change sets in a greedy fashion. This greedy strategy, however, can lead to too many change sets. Specifically, we need to get rid of overlapping change sets in a second phase of the transformation, referred to as post-processing in Kelter, and Taentzer [44]. The post-processing poses a set partitioning problem which may be framed as an optimization problem: We want to cover all low-level changes by a minimum amount of semantic change sets which are mutually disjoint. We implemented the heuristics presented in Kehrer, Kelter, and Taentzer [44] in Java. This can be hardly expressed in OCL, which was developed as a language for querying object structures but not for implementing complex algorithms like the post-processing step of the in-place model transformation scenario described above.

Lastly, related to the previously discussed point of expressiveness, the heterogeneity of Java code compared to ATL code also sticks out. The structure of ATL rules, enforced by ATL's strict syntax, allows for writing consistent code across different transformations. This means that developers can quickly see the basic intent of a rule. The same cannot be said for Java methods. While our translation scheme, combined with the developed libraries, produces an internal DSL for transformations, Java code is far less homogeneous due to the absence of any dedicated structure within methods that perform transformations. This can also be seen in our classification from Sect. 4.2. Each Java statement can either have transformation-specific semantics (i.e. *Binding* or *Tracing*) or perform any *other* transformation-unrelated task. This problem of intermixed transformation and non-transformation code within GPLs also persists throughout other internal transformation DSLs such as the NMF transformation languages [45], YAMTL [46], RubyTL [47], or SiTra [48]. But this does not only bring disadvantages. The strict structure of ATL allows to easily design mappings from one input type to one output type. This can suffice in many cases as highlighted by Götz and Tichy [16]. However, in cases where several different input types need to be matched to the same output type (n-to-1), one input type needs to be matched to several output types (1-to-n), or a combination of the two cases (n-to-m), code duplicates are often unavoidable. In heterogeneous Java code, such situations can be handled more easily. All in all, the relationship between the input and output meta-models should also be considered when deciding between using an MTL or a GPL.

7.4 Limits of our results in the context of the research field

Up till now our discussion of MTL vs. GPL largely boiled down to the abstraction of model traversal and tracing provided by ATL. This is of course by design as our study focused on the comparison of Java and ATL. ATL being the most used model transformation language and Java being one of the most dominant programming languages of the last decade. Nonetheless, there are more model transformation-specific features that other model transformation languages provide. Depending on the situation these features could also influence the decision of using a specific model transformation language over general purpose languages.

An extension of the model traversal and matching features of ATL comes in the form of graph pattern matching in graph-based model transformation languages such as Henshin [25]. This allows transformation developers to define complex model element relationships that are automatically searched and matched by advanced matching engines. There exist some advances of trying to replicate this behaviour in general purpose languages for example FunnyQT [49] or SDMLib/Fujaba [50], but even in those cases DSLs are used for defining the graph patterns.

Some model transformation languages allow to run analysis on the written transformations such as critical pair analysis [51] or even verify property preservation by a transformation [52], both of which are not easily accessible for transformations written in general purpose languages. The better analysability of MTLs stems from their syntax being transformation-specific, as also seen in the structure of our classification schemata from Sect. 4.

Being able to design bidirectional transformations based on only one transformation script is also a unique property of model transformation languages. Examples of such languages are detailed and compared in Anjorin, Buchmann, Westfechtel, et al. [18] or [53]. Some languages like eMoflon Leblebici et al.[30], NMF Synchronizations [45], or Viatra [54] extend this further by providing the ability to perform incremental transformations both being features that are hard to reproduce in general purpose languages in our experience. Even ATL now has several extensions allowing it to run incremental transformations [55,56].

Currently, for general purpose languages to be considered for writing transformations, all the stated advanced features such as graph pattern matching, bidirectional and incremental transformations as well as transformation analysis and verification should not be an essential requirement of the development. This is because none of them can be implemented with justifiable effort in GPLs.

8 Threats to validity

This section addresses potential threats to the validity of the presented work.

8.1 Internal validity

The manual steps done throughout our study pose some threat to the internal validity of our study. Both the translation based on our translation schema and the labelling of the Java code were done manually and thus open the possibility of human error. Furthermore the program we developed to calculate the word count of the Java code could also contain errors. We counteracted these threats by testing the correctness of the resulting transformations to the extent that was possible based on available resources. This was done by testing the output of the translated transformations against the output of the ATL transformations from which they originated as well as through rigorous peer reviews. We further verified the correctness of our labels and the produced word counts through reviews as detailed in Sect. 5.

All assumptions we make about cause and effect of increase or decrease of size and complexity as well as of overhead is supported by more detailed investigations and analysis throughout our research.

8.2 External validity

To mitigate a potential threat to the external validity of our work due to a bias in the selected transformation modules we chose the analysed transformations from a variety of sources and different authors. Moreover, both the purpose and involved meta-models differ between each transformation module, thus providing a diverse sample set.

However, the transformations chosen for evaluation in our work were subject to a number of constraints which poses a threat to the generalizability of our results. While we aimed to select a variety of transformation modules w.r.t. scope and size, the limitation of LOC may introduce a threat to the external validity of our work.

Due to the study setup of selecting ATL transformations and translating those into Java, there is the possibility of a bias in favour of ATL. It is potentially more likely for an ATL solution to exist, if the problem it solves is well suited for being developed in ATL. As a result the results of our study might not be applicable to all model transformations. However, our study does not try to confirm that ATL is the superior language for developing transformations, but discusses based on the presented observations, which advantages a dedicated language like ATL can offer. In order to be able to recognise why ATL is a good solution for certain cases, it is necessary to look at precisely such cases. In order to validate our results, a further study should be carried out. There, the study design should be reversed so that ATL solutions are derived from existing Java solutions.

Lastly, all our observations are limited to the comparison between ATL and Java which limits their generalizability. While the observations might also hold for comparing Java or similar languages with transformation languages similar to ATL, e.g. QvT-O, they cannot be transferred to graph-based transformation languages such as Henshin or even QvT-R.

8.3 Construct validity

The next threat concerns the appropriateness and correctness of our translation schema and the resulting transformations. We tried to mitigate this threat by following the design science research method and using two separate reviewers for the proposed transformation schema.

The used metrics for measuring complexity and size need also be discussed. We opted to use cyclomatic complexity for measuring the complexity of Java transformations because it is one of the most widely used measures for object-oriented languages and has been shown in numerous publications to relate both to the maintainability and reliability of code [29]. Because both quality attributes are of interest in the discussion of MTLs vs. GPLs, we believe the cyclomatic complexity to be a good measure to assess the impact that overhead Java code has on the quality of transformations. Likewise lines of code are a popular measure for size in all of programming but has also been criticized due to its disregard for the difference in programming styles and formatting. To counteract this problem, all Java code was developed by the same researcher using the same standard code formatter. To further counterbalance issues with lines of code as a solitary size measure, we supplemented it with the additional measure word count that has been argued to be more accurate in measuring the size of a programmed solution [18]. In cases where their ranking differs, we then investigated the cause of the discrepancy and discussed what this means for our observations and analysis.

8.4 Conclusion validity

To ensure reproducible results, we provide all the data and tools used for our study in the supplementary materials for this work. A repetition of our approach using the provided

materials will end with the same results as those presented here. However, more than one way of translating ATL constructs into Java constructs and thus multiple translation schemas are possible. This impacts the conclusion validity of our study because different design decisions for the translation schema may impact the reproducibility of our results.

9 Related work

To the best of our knowledge, there exists no research that relates the size and complexity of transformations written in a MTL with that of transformations written in a GPL. However, there do exist several publications that provide relevant context for our work.

Hebig et al. investigate the benefit of using specialized model transformation languages compared to general purpose languages by means of a controlled experiment where participants had to complete a comprehension task, a change task, and they had to write one transformation from scratch [13]. They compare ATL, QVT-O, and the GPL Xtend, and they found no clear evidence for an advantage when using MTLs. In comparison with their setup, we focus on a larger number of transformations. Furthermore, examples shown in the publication also suggest that they did not consider ATLs refining mode for their refactoring task nor did their examples focus on advanced transformation aspects such as tracing.

As previously described, parts of our research build upon the work presented in Götz and Tichy [16]. Here, the authors use a complexity measure for ATL proposed in the literature to investigate how the complexity of ATL transformations is distributed over different ATL constructs such as matched rules and helpers. Their results provide a relevant data set to compare our complexity distributions in Java transformations to.

In Amstel and Brand [57] the authors use McCabe complexity to measure the complexity of ATL helpers. Among others, this is also done in Vignaga [58]. Similar to this, we use McCabe complexity on transformations written in Java, which includes translated helpers, to measure the complexity of the code.

The Model Transformation Tool Contest (TTC)[6] aims to evaluate and compare various quality attributes of model transformation tools. While some of these quality attributes (e.g. readability of a transformation specification) are related to the MTL used by the tool, most of the attributes are related to tooling issues (such as usability or performance) which are out of the scope of our study. Moreover, the contest is about comparing different MTLs with each other rather than comparing them with a GPL. Nonetheless, some cases have been presented along with a reference implementation in Java [59,60], which could serve as another source for comparing MTLs and GPLs more widely, including tooling- and execution-related aspects.

Sanchez Cuadrado et al. [28] propose A2L, a compiler for parallel execution of ATL model transformations. A2L takes ATL transformations as input and generates Java code that can be run from within their self-developed engine. Their data-oriented ATL algorithm describes how ATL transformations are executed by their code and closely resembles the structure embodied in our translation schema.

Our approach to utilise libraries and define certain restrictions on the structure of code in Java defines an internal DSL for developing transformations. There exists a large body of research into the topic of the design of internal transformation languages for several general purpose languages. It would be impossible to list them all here. For this reason, we will limit our discussion to a small selection of internal DSLs which have points of contact with our Java DSL.

The Simple Transformation Library in Java (SiTra) introduced in Akehurst et al. [48] provides a simple set of interfaces for defining transformations in Java. Their interfaces abstract rules and traversal in which they follow an approach similar to ours. However, they do not provide ways for trace management.

Another JVM-based transformation DSL is presented by Boronat [46]. The language YAMTL is a declarative internal language for Xtend. In contrast to our approach, this language breaks with the imperative concepts of its host language and offers an ATL-like syntax for defining transformations.

Batory and Altoyan [61] describe Aocl, an implementation of OCLs underlying relational algebra for Java. Much like OCL, Aocl allows developers to define constraints and queries for a given model using a straightforward syntax. The authors further argue that, if expanded, Aocl could be used to write model-to-model transformations, but currently this feature does not exist. Using a MDE tool it is possible to generate a Java package that allows to use Aocl for a class diagram passed to the tool.

In Hinkel and Burger [45] the authors introduce NMF-Synchronisations, an internal DSL for C# for developing bidirectional transformations. The language is built with the intention to reuse as much of the tool support from its host language as possible. Much like our Java SE14 approach, they utilise functional language constructs added to C# to allow a more declarative way of defining transformations while still retaining the full potential of the host language.

10 Conclusion

In this work, we presented how we developed and applied a translation schema to translate ATL transformations to Java. We also described our results of analysing the complexity and size as well as their distribution over the different

[6] https://www.transformation-tool-contest.eu/.

transformation aspects. For this purpose, we used McCabe complexity, LOC, and word count to measure the size and complexity of 12 transformations translated to Java SE5 and Java SE14, respectively. Based on our findings, we then discussed improvements of Java over the years as well as how well suited these newer language iterations are for writing model transformations.

We found that new features introduced into Java since 2006 help to significantly reduce the complexity of transformations written in Java. Moreover, while they also help to reduce the size of transformations when measured in lines of code, we saw no decrease in the number of words required to write the transformations. This suggests an ability to express more information dense code in newer Java versions. We also showed that, while the overall complexity of transformations is reduced, the distribution of how much of that complexity stems from code that implements functionality that ATL and other model transformation languages can hide from the developer stays about the same. This observation is further supported by the analysis of code size distribution. Here, we found that while large parts of the transformation classes relate to the transformation process itself, within those parts there is still significant overhead from tracing as well as general supplemental code required for the transformations to work. We conclude that while the overall complexity is reduced with newer Java versions, the overhead entailed by using a general purpose language for writing model transformations is still present.

Our regression models for predicting Java code size based on OCL expressions suggest a linear relationship for both Java SE5 and Java SE14 with the newer Java version having a slightly lower growth factor.

Overall we find that the more recent Java version makes development of transformations easier because less work is required to set up a working transformation, and the creation of output elements and the assignment of their attributes are now a more prominent aspect within the code. From our results and experience with this and other projects, we also conclude that general purpose languages are most suitable for transformations where little to no tracing is required because the overhead associated with this transformation aspect is the most prominent one and holds the most potential for errors. However, while we do not see them as prominently used, we believe that advanced features such as property preservation verification or bidirectional and incremental transformation development cannot currently be implemented with justifiable effort in a general purpose language.

For future work, we propose to also look at the transformation development process as a whole, instead of only at the resulting transformations. In particular, we are interested in investigating how the maintenance effort differs between transformations written in a GPL and those written in a MTL. For this purpose, the presented artefacts can be reused. Simple modifications to the ATL transformations can be compared to what needs to be adjusted in the corresponding Java code. Furthermore, because developers are the first to be impacted by the languages, it is also important to include users into such studies. For this reason, we propose to focus on user-centric study setups to be able to better study the impact of the language choice on developers. Such studies could also investigate several other relevant aspects. For example, how well users are aided by *tool support* or the impact of *previous knowledge* of the languages or involved models on the resulting GPL or MTL code. Moreover, the impact of language choice on transformation performance, an aspect that gets more relevant with the ever increasing size of models [62], can also be investigated with our setup. Here, we envision the use of run-time measures like execution time and memory or CPU utilization to compare MTL solutions with their GPL counterparts, to investigate the scalability of the underlying technologies.

Another potential avenue to explore is the comparison with a general purpose language that has a more complete support for functional programming such as Scala. Additional features such as pattern matching and easier use of functional syntax for translating OCL expressions could potentially help to further reduce the complexity of the resulting transformation code.

Funding Open Access funding enabled and organized by Projekt DEAL.

A OCL expression translations in Java SE5

```
1 Collection<Type> newCollection = new
      Collection<>();
2 for (Type t: collection) {
3     if (e) {
4         newCollection.add(t);
5     }
6 }
```

List. 24 Translation of collection->select(e) in Java SE5.

```
Collection<ResultType> newCollection = new Collection<>();
for (Type t: collection) {
  ResultType r = ...; //manipulate t in accordance with e
  newCollection.add(r);
}
```

List. 25 Translation of collection->collect(e) in Java SE5.

```
boolean includes = false;
for (Type t: collection) {
    includes |= t == x;
}
```

List. 26 Translation of collection->includes(x) in Java SE5.

```
element.getAttribute();
```

List. 27 Translation of element.attribute in Java SE5.

```
Collection<AttributeType> newCollection =
    new Collection<>();
for (Type t: collection) {
    if (e) {
    newCollection.add(t.getAttribute());
    }
}
```

List. 28 Translation of collection.attribute in Java SE5.

```
if (i > 5) {}
```

List. 29 Translation of i | i > 5 in Java SE5.

References

1. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Softw. (2003). https://doi.org/10.1109/MS.2003.1231150
2. Götz, S., Tichy, Matthias, Groner, R.: Claimed advantages and disadvantages of (dedicated) model transformation languages: a systematic literature review. Softw. Syst. Model. **20**(2), 469–503 (2021). https://doi.org/10.1007/s10270-020-00815-4
3. Jouault, Frédéric., et al.: ATL: a model transformation tool. Sci. Comput. Program. (2008). https://doi.org/10.1016/j.scico.2007.08.002
4. Krikava, F., Collet, P., France, R.: Manipulating models using internal domain-specific languages. In: Symposium On Applied Computing. Gyeongju, South Korea (2014). https://doi.org/10.1145/2554850.2555127
5. Gray, J., Karsai, G.: An examination of DSLs for concisely representing model traversals and transformations'. In: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (2003). https://doi.org/10.1109/HICSS.2003.1174892
6. Jouault, F. et al.: ATL: a QVT-like transformation language. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (2006). https://doi.org/10.1145/1176617.1176691
7. Burgueño, L., Cabot, J., Gerard, S.: The future of model transformation languages: an open community discussion. In: Journal of Object Technology 18.3. Ed. by Anthony Anjorin and Regina Hebig. The 12th International Conference on Model Transformations, 7:1-11. ISSN: 1660-1769 (2019). https://doi.org/10.5381/jot.2019.18.3.a7
8. Kehrer, T., Kelter, U., Ohrndorf, M. et al.: Understanding model evolution through semantically lifting model differences with SiLift. In: 28th IEEE International Conference on Software Maintenance (ICSM), pp. 638–641. IEEE (2012)
9. Kehrer, T., Taentzer, G. et al.: Automatically deriving the specification of model editing operations from meta-models. In: International Conference on Theory and Practice of Model Transformations, pp. 173–188. Springer (2016)
10. Rindt, M., Kehrer, T., Kelter, U.: Automatic generation of consistency-preserving edit operations for MDE tools. In: Demos@ MODELS 14 (2014)
11. Schultheiß, A., Bittner, P.M. et al.: On the use of product-line variants as experimental subjects for clone-and-own research: a case study. In: SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19–23, 2020, Volume A. ACM, 27:1–27:6 (2020)
12. Schultheiß, A., Boll, A., Kehrer, T.: Comparison of graph-based model transformation rules. J. Object Technol. **19**(2), 1–21 (2020)
13. Hebig, R. et al.: Model transformation languages under a magnifying glass: a controlled experiment with Xtend, ATL, and QVT. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York, NY, USA (2018). https://doi.org/10.1145/3236024.3236046
14. Rentschler, A. et al.: Designing information hiding modularity for model transformation languages. In: Proceedings of the 13th International Conference on Modularity. MODULARITY '14 (2014). https://doi.org/10.1145/2577080.2577094
15. Höppner, S., Tichy, M., Kehrer, T.: Contrasting Dedicated Model Transformation Languages vs. General Purpose Languages: A Historical Perspective on ATL vs. Java based on Complexity and Size: Supplementary Materials (2021). https://doi.org/10.18725/OPARU-38923
16. Götz, S., Tichy, M.: Investigating the origins of complexity and expressiveness in ATL transformations. In: The 16th European Conference on Modelling Foundations and Applications (ECMFA 2020) Journal of Object Technology 19.2. Ed. by Richard Paige and Antonio Vallecillo, 12:1-21 (2020). https://doi.org/10.5381/jot.2020.19.2.a12
17. Wieringa, R.J.: Design science methodology for information systems and software engineering. Undefined (2014). https://doi.org/10.1007/978-3-662-43839-8
18. Anjorin, A., Buchmann, T., Westfechtel, B., et al.: Benchmarking bidirectional transformations: theory, implementation, application, and assessment. Softw. Syst. Model. (SoSyM). (2019). https://doi.org/10.1007/s10270-019-00752-x
19. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **SE–2**(4), 308–320 (1976). https://doi.org/10.1109/TSE.1976.233837
20. Götz, S., Tichy, M., Kehrer, T.: Dedicated model transformation languages vs. general-purpose languages: a historical perspective on ATL vs. java. In: Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development—

Volume 1: MODELSWARD, INSTICC. SciTePress, pp. 122–135 (2021). https://doi.org/10.5220/0010340801220135
21. Steinberg, D., et al.: EMF: Eclipse Modeling Framework. Pearson Education (2008)
22. OMG.: Meta Object Facility (MOF) (2016). https://www.omg.org/spec/MOF
23. OMG.: Object Constraint Language (OCL) (2014). https://www.omg.org/spec/OCL/2.4/PDF
24. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)
25. Strüber, D. et al.: Henshin: a usability-focused framework for emf model transformation development. In: International Conference on Graph Transformation, pp. 196–208. Springer (2017)
26. Anjorin, A., Buchmann, T., Westfechtel, B.: The families to persons case. In: TTC'17 (2017)
27. Jouault, F.: ATL/Tutorials—Create a simple ATL transformation (2013). https://wiki.eclipse.org/ATL/Tutorials_-_Create_a_simple_ATL_transformation. Accessed 12 June 2021
28. SanchezCuadrado, J., et al.: Efficient execution of ATL model transformations using static analysis and parallelism. IEEE Trans. Softw. Eng. (2020). https://doi.org/10.1109/TSE.2020.3011388
29. Jabangwe, R., et al.: Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review. Empir. Softw. Eng. **20**(3), 640–693 (2015). https://doi.org/10.1007/s10664-013-9291-7
30. Weidmann, N. et al.: Incremental (unidirectional) model transformation with eMoflon::IBeX. In: Transformation, Graph (ed.) Esther Guerra and Fernando Orejas, pp. 131–140. Springer, Cham (2019) 978-3-030-23611-3
31. Cicchetti, A., et al.: JTL: a bidirectional and change propagating transformation language. In: Malloy, B., Staab, S., van den Brand, M. (eds.) Software Language Engineering, pp. 183–202. Springer, Berlin (2011)
32. Hinkel, G.: NMF: A Modeling Framework for the. NET Platform, KIT (2016)
33. Buchmann, T.: BXtend-a framework for (bidirectional) incremental model transformations. In: MODELSWARD, pp. 336–345 (2018)
34. Aniche, M.: Java code metrics calculator (CK) (2015). https://github.com/mauricioaniche/ck
35. Batory, D.S., Altoyan, N.: Aocl: a pure-java constraint and transformation language for MDE. In: MODELSWARD, pp. 319–327 (2020)
36. Singh, Y., Kaur, A., Malhotra, R.: Application of logistic regression and artificial neural network for predicting software quality models. In: Software Engineering Research and Practice, pp. 664–670 (2007)
37. Aggarwal, K.K., et al.: Investigating effect of design metrics on fault proneness in object-oriented systems. J. Object Technol. **6**(10), 127–141 (2007)
38. Pai, J.G., BechtaDugan, J.: Empirical analysis of software fault content and fault proneness using bayesian methods. IEEE Trans. Softw. Eng. **33**(10), 675–686 (2007). https://doi.org/10.1109/TSE.2007.70722
39. Guo, Y. et al.: An empirical validation of the benefits of adhering to the law of demeter. In: 2011 18th Working Conference on Reverse Engineering, pp. 239–243 (2011). https://doi.org/10.1109/WCRE.2011.36
40. GopalakrishnanNair, T.R., Selvarani, R.: Defect proneness estimation and feedback approach for software design quality improvement. Inf. Softw. Technol. **54**(3), 274–285 (2012). https://doi.org/10.1016/j.infsof.2011.10.001
41. Olbrich, S. et al.: The evolution and impact of code smells: a case study of two open source systems. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 390–400 (2009). https://doi.org/10.1109/ESEM.2009.5314231
42. Alshayeb, M., Li, W.: An empirical validation of object-oriented metrics in two different iterative software processes. IEEE Trans. Softw. Eng. **29**(11), 1043–1049 (2003). https://doi.org/10.1109/TSE.2003.1245305
43. Hinkel, G., Goldschmidt, T., et al.: Using internal domain-specific languages to inherit tool support and modularity for model transformations. Softw. Syst. Model. **18**(1), 129–155 (2019). https://doi.org/10.1007/s10270-017-0578-9
44. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 163–172. IEEE (2011)
45. Hinkel, G., Burger, E.: Change propagation and bidirectionality in internal transformation DSLs. Softw. Syst. Model. **18**(1), 249–278 (2019). https://doi.org/10.1007/s10270-017-0617-6
46. Boronat, A.: Expressive and efficient model transformation with an internal DSL of Xtend. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. MODELS '18. Copenhagen, Denmark: Association for Computing Machinery, pp. 78–88. ISBN: 9781450349499 (2018). https://doi.org/10.1145/3239372.3239386
47. Cuadrado, J.S., Molina, J.G., Tortosa, M.M.: RubyTL: a practical, extensible transformation language. In: Rensink, A., Warmer, J. (eds.) Model Driven Architecture-Foundations and Applications, pp. 158–172. Springer, Berlin (2006)
48. Akehurst, D.H. et al.: SiTra: simple transformations in java. In: Model Driven Engineering Languages and Systems, pp. 351–364. Springer (2006), ISBN: 978-3-540-45773-2
49. Horn, T.: Model querying with FunnyQT. In: Duddy, K., Kappel, G. (eds.) Theory and Practice of Model Transformations, pp. 56–57. Springer, Berlin (2013)
50. Zündorf, A. et al.: Story driven modeling libary (SDMLib): an Inline DSL for modeling and model transformations, the Petrinet-Statechart case. In: Sixth Transformation Tool Contest (TTC 2013), ser. EPTCS (2013)
51. Born, K., et al.: Analyzing conflicts and dependencies of rule-based transformations in henshin. In: Egyed, A., Schaefer, I. (eds.) Fundamental Approaches to Software Engineering, pp. 165–168. Springer, Berlin (2015)
52. Ehrig, H., Ermel, C., et al.: Semantical correctness and completeness of model transformations using graph and rule transformation. In: Ehrig, H. (ed.) Graph Transformations, pp. 194–210. Springer, Berlin (2008)
53. Leblebici, E. et al.: A comparison of incremental triple graph grammar tools. In: Electronic Communications of the EASST 67 (2014). https://doi.org/10.14279/tuj.eceasst.67.939
54. Bergmann, G., et al.: Viatra 3: a reactive model transformation platform. In: Kolovos, D., Wimmer, M. (eds.) Theory and Practice of Model Transformations, pp. 101–110. Springer, Cham (2015)
55. Martínez, S., Tisi, M., Douence, R.: Reactive model transformation with ATL. In: Science of Computer Programming 136, pp. 1–16 (2017). ISSN: 0167-6423. https://doi.org/10.1016/j.scico.2016.08.006. https://www.sciencedirect.com/science/article/pii/S016764231630106X
56. Le Calvar, T., et al.: Efficient ATL incremental transformations. J. Object Technol. **18**(3), 1–2 (2019)
57. van Amstel, M.F., van den Brand, M.G.J.: Using metrics for assessing the quality of ATL model transformations. In: MtATL@TOOLS (2011)
58. Vignaga, A.: Metrics for measuring ATL model transformations. In: MaTE, Department of Computer Science, Universidad de Chile, Tech. Rep (2009)
59. Getir, S. et al.: State elimination as model transformation problem. In: Transformation Tool Contest at the Conference on Software

Technologies: Applications and Foundations (TTC@STAF), pp. 65–73 (2017)
60. Beurer-Kellner, L., von Pilgrim, J., Kehrer, T.: Round-trip migration of object-oriented data model instances. In: Transformation Tool Contest at the Conference on Software Technologies: Applications and Foundations (TTC@STAF) (2020)
61. Batory, D.S., Altoyan, N.: Aocl: a pure-java constraint and transformation language for MDE. In: MODELSWARD, pp. 319–327 (2020)
62. Groner, R., et al.: A survey on the relevance of the performance of model transformations. J. Object Technol. **20**(2), 1–27 (2021). https://doi.org/10.5381/jot.2021.20.2.a5

Stefan Höppner is a Ph.D. student at Ulm University. His research is focused on topics surrounding the development and evaluation of model transformation languages. In particular, he is interested in the advantages and disadvantages that these languages offer in contrast to general purpose languages. Prior to his work as a Ph.D. student he was a student of Software Engineering at Ulm University where he received his M.Sc. in.

Timo Kehrer is professor at Humboldt-Universität zu Berlin (Germany), heading the Model-Driven Software Engineering Group at the Department of Computer Science. Before that, Kehrer was working as research assistant in the Software Engineering and Database Systems Group at University of Siegen (Germany) from 2011 to 2015, and as postdoctoral research fellow in the Dependable Evolvable Pervasive Software Engineering Group at Politecnico di Milano (Italy) from 2015 to 2016. He has active research interests in various fields of model-driven and model-based software and system engineering, with a particular focus on model evolution.

Matthias Tichy is full professor for software engineering at the University of Ulm and director of the institute of software engineering and programming languages. His main research focus is on model-driven software engineering, particularly for cyber-physical systems. He works on requirements engineering, dependability, and validation and verification complemented by empirical research techniques. He is a regular member of programme committees for conferences and workshops in the area of software engineering and model driven development. He is co-author of over 110 peer-reviewed publications.