Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams

Dissertation
zur Erlangung des Doktorgrads Dr. rer. nat.
der Fakultät für Informatik der Universität Ulm

Stefan Sarstedt
aus Gummersbach

Universität Ulm
Fakultät für Informatik
Abteilung Programmiermethodik und Compilerbau
Leiter: Prof. Dr. Helmuth Partsch

2006

Amtierender Dekan:	Prof. Dr. Helmuth Partsch
1. Gutachter:	Prof. Dr. Helmuth Partsch
2. Gutachter:	Prof. Dr. Friedrich v. Henke
3. Gutachter:	Jürgen Dingel, Ph.D., Assistant Professor
Tag der Promotion:	5. Juli 2006

Danksagung

"Indes sie forschten, röntgten, filmten, funkten, entstand von selbst die köstlichste Erfindung: der Umweg als die kürzeste Verbindung zwischen zwei Punkten."

Erich Kästner

An dieser Stelle möchte ich mich bei allen Personen bedanken, die zum Gelingen meiner Arbeit direkt oder indirekt beigetragen haben.

Zunächst möchte ich mich besonders bei Prof. Dr. Helmuth Partsch für das Ermöglichen meiner Forschungsarbeit, die langjährige Unterstützung und angenehme Arbeitsatmosphäre bedanken. Für Anliegen und Fragen aller Art hatte er stets ein offenes Ohr. Des Weiteren gilt mein Dank meinem Zweitgutachter, Herrn Prof. Dr. Friedrich v. Henke, sowie meinem Drittgutachter, Prof. Dr. Jürgen Dingel.

Ohne die Hilfe der gesamten Abteilung Programmiermethodik und Compilerbau an der Universität Ulm wäre diese Arbeit nicht möglich gewesen. Die Umgebung und der kollegiale Umgang miteinander boten mir stets einen idealen Rahmen für meine Forschung. Mein herzlicher Dank gilt besonders meinen "ActiveCharts"-Projektkollegen Jens Kohlmeyer, Alexander Raschke, Dominik Gessenharter und Matthias Schneiderhan für Fallbeispiele, Diskussionen, Ideen und Hilfe bei der Implementierung der "ActiveChartsIDE". Von meinen sehr geschätzten Kollegen ist insbesondere auch Walter Guttmann hervorzuheben, der durch zahlreiche Diskussionen und Anregungen zur Reife der Formalisierung beigetragen hat.

Ohne die moralische Hilfe meiner Freundin Jessica, meines Vaters, meiner Mutter, meines Bruders Marko und meines Freundeskreises hätte ich bereits früh aufgegeben. Für euren unerschütterlichen Glauben an mich und den täglichen Ansporn bin ich euch allen sehr dankbar.

Contents

1 Introduction **9**
- 1.1 Problems of current MDD-approaches . . . 9
- 1.2 Proposed approach . . . 10
- 1.3 Scope of this thesis . . . 10
- 1.4 Overview of this thesis . . . 11

2 UML 2 Activity Diagrams **13**
- 2.1 Introduction . . . 13
- 2.2 Informal semantics . . . 14

3 Abstract State Machines **17**
- 3.1 Basic, Structured and Asynchronous Multi-Agent ASMs . . . 17
  - 3.1.1 Basic ASMs . . . 17
  - 3.1.2 Structured ASMs . . . 18
  - 3.1.3 Asynchronous Multi-Agent ASMs . . . 18
- 3.2 ASM operators . . . 19
- 3.3 Additional ASM rules and operators . . . 20

4 Discussion of UML 2 Activity Diagram Semantics **21**
- 4.1 Targeting controversial elements . . . 21
- 4.2 Problems and enhancements of signals . . . 22
- 4.3 Problems due to errors and obscure information . . . 22
  - 4.3.1 Unclear terms . . . 23
  - 4.3.2 Where to hold control tokens . . . 24
  - 4.3.3 Confusion of the reader due to distributed information . . . 24
  - 4.3.4 Termination of accept event actions without incoming edges . . . 25
  - 4.3.5 Actions without incoming edges . . . 25
  - 4.3.6 Actions without incoming edges but with input pins . . . 25
  - 4.3.7 Data tokens outrun control tokens . . . 25
  - 4.3.8 Buffering of tokens at fork nodes . . . 26
- 4.4 Problems due to missing information . . . 26
  - 4.4.1 Context object for call behavior action . . . 26
  - 4.4.2 Which transitions to execute . . . 27
  - 4.4.3 Multiple callers with “isSingleExecution” . . . 27
  - 4.4.4 Interruptible activity regions . . . 28

5 An ASM Semantics for UML 2 Activity Diagrams **31**
- 5.1 Overview . . . 31
- 5.2 Basic definitions . . . 33
  - 5.2.1 Predefined base domains . . . 33
  - 5.2.2 UML 2 meta model to ASM mapping . . . 34
  - 5.2.3 Abbreviations . . . 38
  - 5.2.4 Configuration of activity executions . . . 38

5.3 Model-based configuration of semantics . . . 39
5.4 ASM initialization . . . 40
5.5 Activity . . . 40
5.5.1 Events . . . 41
5.5.2 Controller loop . . . 42
5.5.3 Start . . . 43
5.5.4 Termination . . . 45
5.5.5 Abort . . . 45
5.5.6 Transitions . . . 46
5.6 Action . . . 47
5.6.1 Creation . . . 48
5.6.2 Enabling . . . 49
5.6.3 Termination . . . 49
5.6.4 Abort . . . 50
5.6.5 Execution . . . 51
5.7 Computation and selection of token offers . . . 61
5.7.1 Overview . . . 61
5.7.2 Data structures . . . 61
5.7.3 Creation of token offers . . . 63
5.7.4 Propagation of token offers . . . 65
5.7.5 Selection of token offers at targets . . . 71
5.7.6 Handling interruptible activity regions . . . 75
5.7.7 Handling accept event actions . . . 78
5.7.8 Buffering of token offers . . . 79
5.7.9 Discussion of the token offer computation . . . 80
5.8 Executing transitions . . . 82
5.9 Discussion . . . 84
5.9.1 Deviations from the specification . . . 84
5.9.2 Related Work . . . 85
5.9.3 Possible extensions and further work . . . 87
5.9.4 Concluding Remarks . . . 88

6 Tool Support **89**
6.1 Architecture . . . 89
6.2 Working with the ActiveChartsIDE . . . 91
6.3 Discussion . . . 92
6.3.1 Related Work . . . 92
6.3.2 Possible Extensions . . . 93
6.3.3 Experiences . . . 93
6.3.4 Concluding Remarks . . . 93

7 Summary **95**
7.1 Contributions . . . 95
7.2 Outlook . . . 96

A Mathematical Conventions **99**

B Case Studies **101**
B.1 Alarm Device . . . 101
B.2 Molding Press . . . 104
B.3 Microwave . . . 106

Zusammenfassung **109**

List of Figures

2.1 Syntax of UML 2 activity diagrams . . . 15
2.2 Lifecycle of an *ActionExecution* . . . 15
2.3 Subset of the UML 2 meta model used . . . 16

4.1 Defining tagged values . . . 21
4.2 Tag definitions for semantic variation points for signals . . . 23
4.3 Faster data tokens . . . 26
4.4 Motivation and problem with buffering at fork nodes . . . 27
4.5 Definition of the semantic variation point for context objects . . . 27
4.6 Multiple interrupting edges . . . 28
4.7 Definition of semantic variation points for *InterruptibleActivityRegion* . . . 29

5.1 UML 2 meta model of behaviored classifiers . . . 32
5.2 Mapping UML activity diagrams to Multi-Agent ASMs . . . 33
5.3 Mapping the UML 2 meta model to ASMs . . . 35
5.4 Enabling and termination of action executions . . . 42
5.5 Creation of an action execution for action B . . . 48
5.6 Enabling of an action execution . . . 50
5.7 Termination of an action execution . . . 51
5.8 Initiation of token flow computation . . . 64
5.9 Propagation of token offers at *MergeNode* . . . 66
5.10 Propagation of token offers at *ForkNode* . . . 67
5.11 Propagation of token offers at *DecisionNode* . . . 68
5.12 Propagation of token offers at *JoinNode* . . . 70
5.13 Selection of token offers for *Action* . . . 72
5.14 Selection of token offers for *CentralBufferNode* and outgoing *ActivityParameterNode* 73
5.15 Selection of token offers for *FinalNode* . . . 73
5.16 Removal of inconsistent offers . . . 74
5.17 Effect of removing inconsistent offers at join nodes . . . 75
5.18 Leaving an interruptible activity region . . . 76
5.19 Buffering of offers . . . 80
5.20 Problematic cases if inconsistent offers were allowed . . . 81
5.21 No order of token offers . . . 82

6.1 ActiveCharts Architecture . . . 90
6.2 ActiveChartsIDE . . . 91
6.3 Execution of the “Heartbeat” example . . . 94

B.1 Alarm device static structure . . . 102
B.2 Alarm device behaviors . . . 102
B.3 Alarm device action implementations . . . 103
B.4 Alarm device object setup . . . 103
B.5 Molding press static structure . . . 104
B.6 Molding press behaviors . . . 105

B.7 Microwave static structure 106
B.8 Extract from microwave behavior 107

Chapter 1

Introduction

Modern software development processes include more or less comprehensive analysis and design phases where often various models describing the static structure and the dynamics of a system are created [Pre04, Lar05]. The primary motivation is to gain a deeper understanding (for discussions with developers and other stakeholders) and to provide a better documentation for the system. Starting from these models, the implementation is created. Unfortunately, models are often not maintained during later phases of the project which leads to divergence of technical documentation and the final implementation, which furthermore impedes system maintenance.

Model-Driven Development (MDD) [BBG05] tries to bridge this gap between analysis/design and implementation by enriching the design artifacts to enable the developer to build parts of the application out of these models automatically. The Unified Modeling Language (UML), which was released in August 2005 in its current version 2, is widely used in this area. One of the recent approaches, which especially encourages the usage of UML for modeling, is the Model-Driven Architecture (MDA) initiative [KWB03, MDA03, McN04] proposed by the Object Management Group (OMG). According to MDA, the UML design artifacts should contain all required information for building a whole application out of the models and extensions. The Executable UML [MB02] is a prominent example for this approach. The static structure of a system is modeled with UML class diagrams, and the behavior is described by UML state charts. An "action language" is used for implementing additional behavior.

1.1 Problems of current MDD-approaches

In our view, there are several shortcomings with current approaches and tool implementations, which we will describe in the following.

No formal semantics. The UML still has no formal semantics, which impedes acceptance by developers and interchangeability of complex models among different tools. Developers and tool implementors can, and frequently do, choose their own interpretation of the UML specification [Ber05]. As a consequence, often only "simple" diagram types, such as UML class diagrams, are used. To tap the full potential of the UML, i.e. to not only serve for mere documentation purposes, especially behavioral diagrams have to be employed for Model-Driven Development.

Predominance of state charts. If behavioral diagrams are supported at all, state charts are predominant because they are well-established and suitable for describing the behavior of embedded systems [Mat05, ILo05]. It is, however, worthwhile to investigate whether other types of behavioral diagrams are also suitable for this or other fields of application.

Graphical information is not sufficient. It is also obvious that graphical information itself is not sufficient to properly describe the functionality of a system [SGK+05]. Therefore, additional

languages were introduced, e.g., the Object Constraint Language (OCL) [WK03], or different syntaxes for the action semantics [UML05] (as utilized by Executable UML [MB02]). These languages are not generally accepted by developers as they do not offer as many possibilities as modern programming languages. In our view, it is necessary to use a common programming language, together with diagrams, for a modern approach to be feasible. Some developers are also still critical of using graphical models for systems development. Freedom should be left as to which functionality should be modeled, and what should not, to lead to an easy transition to Model-Driven Development.

We, therefore, want to propose the following approach to Model-Driven Development which targets these problems.

1.2 Proposed approach

We think that it is worthwhile to consider UML activity diagrams for Model-Driven Development. Activity diagrams describe the sequencing of actions and include control and data flow, parameters, decomposition and control structures such as decision nodes and parallel execution. During the analysis phase of a project, requirements are written in form of Use Cases, which describe interactions between users and a system [Lar05]. Because of the "imperative" style of Use Cases, they are often accompanied by activity diagrams that graphically present the application control flow. It would therefore be natural to directly use these diagrams for implementation, at least for prototyping purposes. Although there are products which support UML activity diagrams, only the older versions 1.x of the UML are considered. The meta model and semantics of these diagrams are based on UML state charts and include only a very restricted subset of the possibilities of the current UML 2 activity diagrams. An adequate set of elements must, however, be provided to be useful for modeling the behavior of a system. This is, in our view, given by the new version of the UML.

In our approach to Model-Driven Development the control flow (i.e., the behavior) of classes that make up an application is modeled with UML 2 activity diagrams during analysis and design phases. These diagrams are seamlessly reused for the implementation by interpreting them at runtime. It is no longer necessary to (re-)code this control flow in a programming language, since the models are executed by a runtime component. Together with generated code out of the static structure of UML 2 class diagrams, this should substantially simplify the creation of applications and lead to a continuous development process from the analysis/design phase to implementation.

Another goal of our approach is to retain the possibility of using regular "hand-written" code for a special type of activity actions. This degree of functionality described by models versus functionality described by code can, therefore, be chosen by the developer, which should improve acceptance of modeling tasks and is particularly useful because not every aspect of a system can and should be modeled graphically.

1.3 Scope of this thesis

We will not validate in this thesis whether our approach is feasible, but rather provide the prerequisites to carry out an investigation. Since the official UML 2 specification only describes the semantics in textual form, misunderstandings arise in its interpretation. To provide a reliable basis for Model-Driven Development, we discuss issues of UML 2 activity diagrams, propose solutions, and provide a formal semantics by specifying Abstract State Machine rules in this work. Based on our formalization, we also designed and implemented an interpreter for UML 2 activity diagrams, accompanied by a model importer and a simulation and debugging component, so that the behavior of an application in terms of the token flow of the activity diagrams can be executed and visualized. This facilitates creating an experimental setting for the validation of our ideas.

Part of this work has already been published in [SRKS05, SGK+05, Sar05, Sar06, SG06].

1.4 Overview of this thesis

The structure of this thesis is as follows: in chapter 2 we describe the syntax and informal semantics of UML 2 activity diagrams, and present the subset considered in this thesis. Chapter 3 gives a short overview on the Abstract State Machine (ASM) formalism we use to describe the semantics. Issues of the UML 2 specification which relate to activity diagrams, and their solutions, are discussed in chapter 4. The main part of this thesis is contained in chapter 5, where the semantics of UML 2 activity diagrams is formally defined by using Asynchronous Multi-Agent Abstract State Machines. We then discuss our tool implementation in chapter 6. A summary on contributions and an outlook are finally given in chapter 7. Mathematical conventions and case studies are contained in the appendix.

The PDF-version of this thesis has "active" links to sections and bibliographical references to ease navigation among the document. This is especially helpful in chapter 5, where ASM rules can be easily traced by using these links.

Chapter 2

UML 2 Activity Diagrams

This section provides an introduction to UML 2 activity diagrams and describes their informal semantics. We do not go into details of each UML element in this chapter. To this end, we refer to chapter 5, where the semantics is formally defined.

2.1 Introduction

Figure 2.1 shows the syntax of all supported elements of UML 2 activity diagrams. Activity diagrams are used to model control and object (or data) flow between actions. Activities can have activity parameter nodes, where incoming or outgoing data are provided. There may be multiple concurrent executions of the same activity, which act independently of each other.

Actions specify transformations on the state of the system that are not further decomposed within the given diagram. They are either implementation-dependent or more specific, e.g., used to send and receive signals or to invoke behavior specified in other diagrams. Actions can have input pins and output pins, where parameters are passed by incoming resp. outgoing object flows. We support the following kinds of actions:

CallBehaviorAction: Call behavior actions are used to invoke other behaviors, which are activities, in our case. Parameters in input or output pins are mapped to the corresponding activity parameter nodes of the invoked activity.

CallOperationAction: Call operation actions invoke methods on objects. Parameters are provided by input pins and return values are written to output pins.

AcceptEventAction: Accept event actions have triggers which indicate the type of event to wait. If such an event (a signal or time event in our case) has occurred, the action terminates. Accept event actions with a "time event"-trigger are also called "wait time actions".

SendSignalAction: These types of actions are used to send signals to other objects. The signals are received by the previously described accept event actions.

BroadcastSignalAction: Whereas send signal actions send signals to a single target only, broadcast signal actions can have multiple targets. Computation of those targets is an explicit semantic variation point in the UML specification.

Actions can have multiple pins and multiple incoming or outgoing control flows. As is the case with activities, actions can also have multiple concurrent action executions.

Object nodes allow for object flows in contrast to control flows. The previously described *input pins* and *output pins* attached to actions, are the object nodes allowing the delivery of data. On the level of activities, objects can be passed through *activity parameter nodes*, which have also been introduced above. Objects may also be buffered in *central buffer nodes*.

Edges connecting actions may pass through *control nodes* that coordinate the flows in an activity diagram. A *decision node* chooses between different outgoing edges and the corresponding *merge node* unites alternate, independent flows. On the other hand, a *fork node* splits a flow into concurrent flows along all outgoing edges and the corresponding *join node* synchronizes all incoming flows. Furthermore, flows may originate in *initial nodes* and terminate in *final nodes*. Final nodes further decompose into *flow final nodes*, which terminate flows only, and *activity final nodes*, which terminate the current activity execution. Edges may also have guards which can prevent control or object flow to occur. Guards are particularly applied on outgoing edges of decision nodes.

An interruptible activity region marks a subset of nodes and edges supporting the termination of parts of an activity diagram. If so-called *interrupting edges*, which lead out of those regions, are passed, all actions and flows inside the region are aborted.

Several levels are defined in the UML 2 specification [UML05] that support different parts of these concepts. The *fundamental level* defines activities, actions and activity groups. The *basic level* adds control sequencing and data flow between actions. We mainly address the *intermediate level* that additionally includes the before mentioned object nodes, concurrent flows with guards, and decisions. Interruptible activity regions, which are contained in the *complete level*, are discussed as an example of a useful feature with vague semantics (see chapter 4).

The UML specification provides a meta model to define the abstract syntax for activity diagrams. The subset of this meta model we use in our work is illustrated in Figure 2.3. The only change with respect to the original model is that we only allow for a single trigger to be provided for accept event actions, rather of multiple triggers. This is not a limitation, because multiple accept event actions can be used instead.

2.2 Informal semantics

The specification proposes a "petri-like semantics" for activity diagrams [UML05, p. 314]. Tokens determine the current state of execution. Control tokens can be held on initial nodes and actions, and data tokens can rest on object nodes. These tokens are *offered* to the outgoing edges of the corresponding node. If "accepted" by targets, tokens traverse the whole path from the source node to the target action or object node at once. This is called the *traverse-to-completion* principle [Boc04]. Control nodes, such as decision or join nodes, therefore only act as "traffic switches", since tokens cannot rest on those nodes.

There is also *token competition* among object flow edges. Although object nodes can have multiple outgoing edges, only one edge is actually passed, even if other edges can be traversed.

If there are tokens on all incoming edges and on all input pins of an action, an action execution can be created (see Figure 2.2 a). All data tokens are removed from the source object nodes and moved to the input pins. To enable an action execution, and thus to put it into a "running" state, data tokens are removed from the input pins and *all* control tokens on the incoming control flow edges are deleted. This is depicted in Figure 2.2 b). If an action execution has terminated (see Figure 2.2 c), control tokens are offered on all outgoing edges and data tokens are created for all output pins of the action. These new tokens are now available for the execution of consecutive actions.

Note that all data tokens are moved to the target input pins *at once*. The reason is, that otherwise a deadlock situation can arise. An example is given in [Boc04].

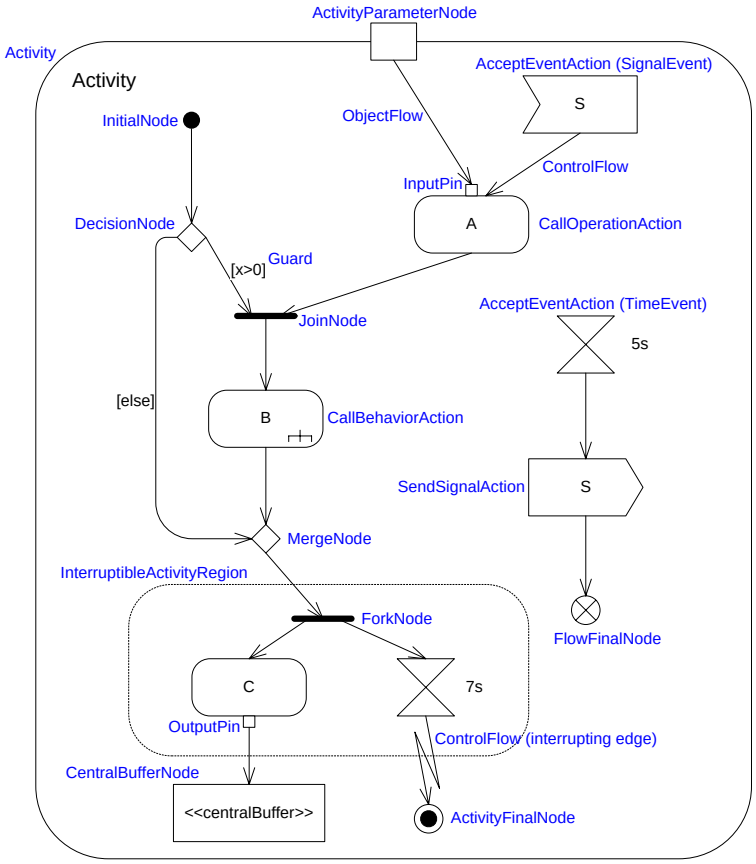

Figure 2.1: Syntax of UML 2 activity diagrams

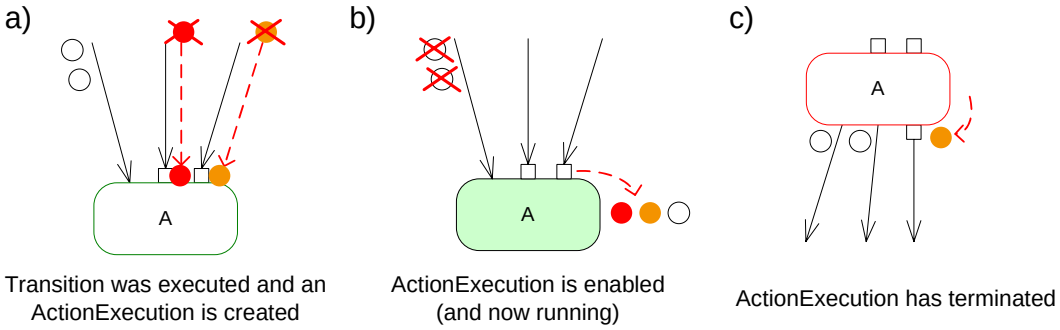

Figure 2.2: Lifecycle of an *ActionExecution*

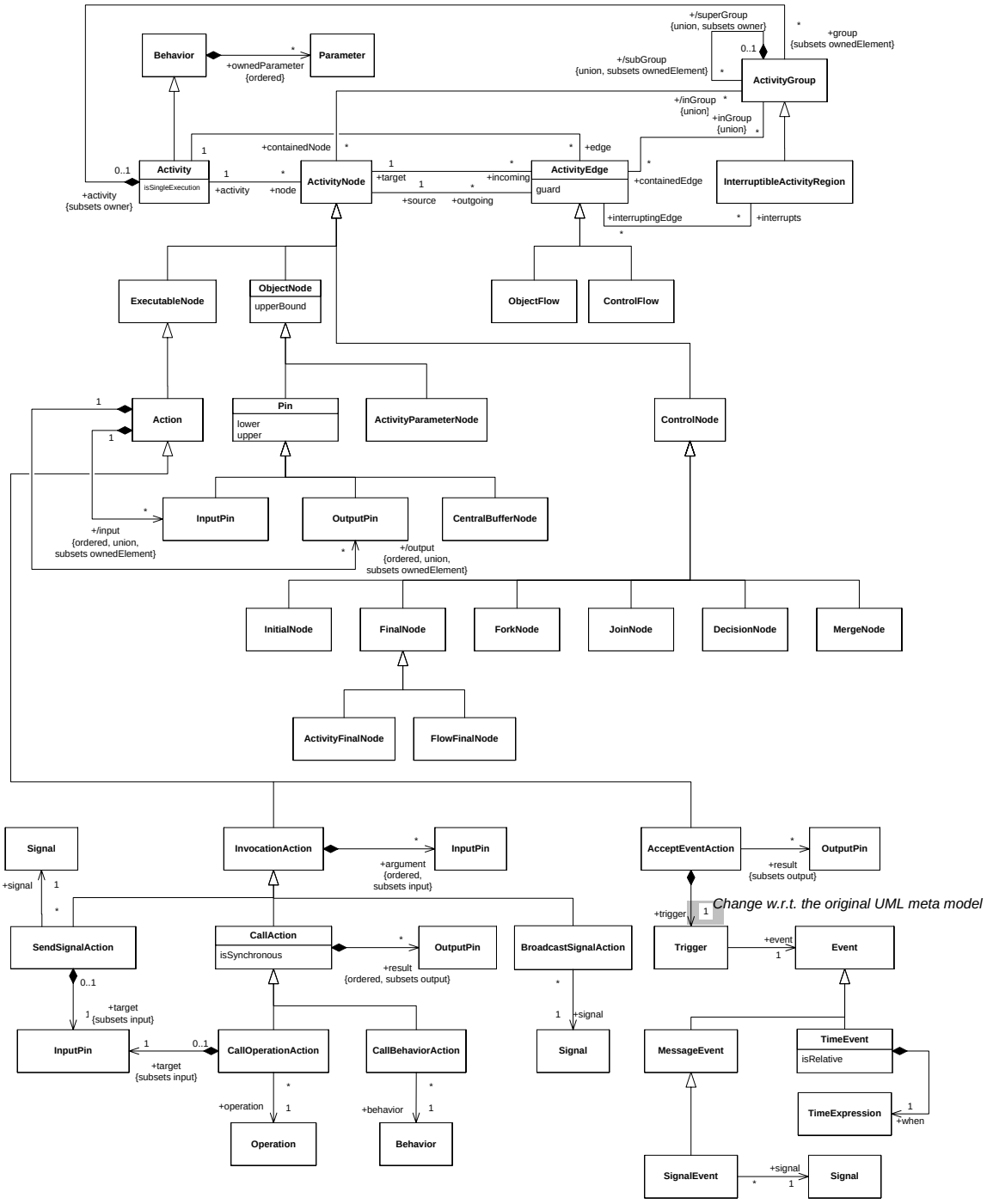

Figure 2.3: Subset of the UML 2 meta model used

Chapter 3

Abstract State Machines

An Abstract State Machine (ASM) is used to formally define the behavior of a system. The level of abstraction can be chosen freely. We will use ASMs to define the semantics of UML 2 activity diagrams in chapter 5. Here, we give a short overview of ASMs and introduce keywords that are relevant in our context. ASMs were introduced by Gurevich [Gur94]. A comprehensive introduction and overview is given in [BS03]. We also refer to the official standard of the Specification and Description Language (SDL) [ITU02], which also uses ASMs for the definition of a formal semantics. The layout of ASM rules and keywords for domains and function classifications are inspired by the SDL formalization.

In Section 3.1 we give the basics of Abstract State Machines, and present the different types of ASMs necessary in this work. Section 3.2 describes the ASM operators in details. Additional operators are finally defined in Section 3.3.

3.1 Basic, Structured and Asynchronous Multi-Agent ASMs

In this section, we introduce the different types of ASMs used.

3.1.1 Basic ASMs

A basic ASM consists of a finite set of rules of the form

if *condition* **then** *updates*

Updates is a finite set of function assignments of the form $f(x_1, \ldots, x_n) := x$. A *state* is given by interpretations of all ASM functions. The updates of all applicable rules in a state are collected in an *update-set*. All updates are then executed *at once*, resulting in a new state of the ASM. The following rules yield the update-set $\{(x, 2), (y(0), 1)\}$, if the current state of the ASM is $\{(x, 1), (y(0), 2)\}$:

if x=1 **then** x:=y(0)
y(0):=x

If an update-set contains assignments which are in conflict with each other for a given function, the update-set is considered *inconsistent*. Updates are in conflict when different values are assigned to the same location for a function. The following rules yield the inconsistent update-set $\{(x, 1), (y, 3), (x, 2)\}$, due to the conflicting updates for x:

x:=1
y:=3
x:=2

We define that an ASM *terminates* if no more rules are applicable. Note that we do not want an empty update-set to lead to termination of the ASM. The termination criterion can be freely chosen [BS03] and differs among various ASM dialects and implementations.

Multiple operators are defined for ASMs. Synchronous parallelism, for example, is supported by **forall**, and nondeterministic selection is achieved by **choose**. See Section 3.2 below for a description of all operators that are relevant in our context.

ASM functions can be classified as follows [BS03, ITU00a]:

- **static** functions do not change during execution of the ASM, i.e. their values remain constant among different states.
- **controlled** functions can be read and written by an ASM agent.
- **shared** functions can be read and written by multiple ASM agents (in case of Multi-Agent ASMs, see below).
- **monitored** functions are modified by the environment and can only be read by ASM agents.

3.1.2 Structured ASMs

Structured ASMs provide sequencing of rules by using **seq**, local variables (**local**), and return values (**result**). Sequencing enables structuring within a single rule, by making intermediate updates locally visible. Only the resulting update-set is applied to the global ASM state. The following example thus results in the update-set $\{(x, 2), (y, 1)\}$:

```
x:=1
seq
y:=x
x:=2
```

Note that our **seq**-operator separates whole blocks of statements instead of single function assignments.

Additionally, *Macros* can be defined with parameters to provide abstraction and structuring of rules:

```
SUM : P(Nat) → Nat
SUM(xs) ≡
    local sum : Nat := 0
    forall x with x ∈ xs
        sum := sum + x
    result := sum
```

3.1.3 Asynchronous Multi-Agent ASMs

Basic ASMs only execute a single ASM *agent*, i.e. only one ASM execution exists at a time. Multi-Agent ASMs enhance basic ones, by allowing *multiple* concurrent agents running parallelly, each one executing its own rule. These agents can communicate with each other by shared functions. New agents are introduced and executed by obtaining a fresh element from the reserve-set of the predefined domain *Agent*, and assigning the initial rule to *ASM(agent)*:

```
let
    myAgent = new(Agent)
in
    ASM(myAgent) := ASMRULETOEXECUTE
```

The initial agent is defined by

```
initially Agent = {init}
initially ASM(init) = ASMRULETOEXECUTE
```

3.2 ASM operators

This section gives a short overview over the ASM operators we use in our rules.

ASM operators	
forall x **with** φ P	Executes P in parallel for all elements x that satisfy φ.
choose x **with** φ P	Chooses an element x nondeterministically, that satisfies φ, and executes P. If no such element exists, the update-set of this rule is empty. **choose** works as an *angelic* choice operator, i.e. if a suitable element x that satisfies φ exists, it is chosen [WM97, BS03].
let x =*expr* **in** P	Binds variable x to expression *expr* in rule P.
let $x = new(X)$ **in** P	Introduces a new element from the reserve set *Res(X)* (see [BS03]) and binds it to x in rule P.
iterate P	Executes P repeatedly, until its update-set is empty.
P **seq** Q	Executes P, then Q with the (intermediate) update-set of P. The resulting update-set for the new state of the ASM is given by the combination $P \oplus Q$ of both update-sets, see [BS03].
if *condition* **then** P	Executes P, if *condition* is met.
if *condition* **then** P **else** Q	Executes P, if *condition* is met, else Q.
case *expr* **of** $x_1 : P_1$,$x_2 : P_2$,...,$x_n : P_n$	Executes rule P_i if *expr* evaluates to x_i.
domain X	Introduces a new domain X, see [ITU00c].
static, **controlled**, **shared**, **monitored**	For classification of functions and domains. **Static** functions do not change during execution of the ASM. **Controlled** functions are local to the current ASM agent. **Shared** functions can be read and written by multiple agents, and are, therefore, used for communication between agents. **Monitored** functions are modified by the environment and can only be read by ASM agents.
local $x : T$:=*expr*	Defines a local variable x of type T (optional) and assigns *expr* to it.
skip	Executes the empty rule. Leads to an empty update-set.
result:=*expr*	Returns result *expr* from the current macro.
add x **to** S, **delete** x **from** S	Add element x to set S, resp. removes it. Can be executed in parallel with other **add**/**remove** commands without leading to inconsistent update-sets, see [GT01].
add X **to** S, **delete** X **from** S	Enhances the ordinary **add**/**remove** constructs by adding resp. removing *all elements* $x \in X$ from S, where X denotes a set.
initially φ	Defines an initial condition φ for the ASM.
constraint φ	Defines a constraint φ for the execution of the ASM.
true, *false*, *undefined*	Predefined for all ASMs.

3.3 Additional ASM rules and operators

We introduce the following additional ASM operators, which are defined in terms of existing ASM constructs.

choose x **with** φ P ... **ifnone** Q executes rule Q, if no element x, that satisfies φ, can be chosen:

choose x **with** φ P **ifnone** $Q \equiv$
  if $\exists x : \varphi$ **then**
    choose x **with** φ
      P
  else
    Q

foreach x **in** S R executes R for each element x in set S, where R denotes a rule with $x \in \mathit{freeVar}(R)$:

foreach x **in** S $R \equiv$
  local $m := S$
  iterate
    if $|m| > 0$ **then**
      choose x **with** $x \in m$
        $m := m \setminus \{x\}$
        R

Chapter 4

Discussion of UML 2 Activity Diagram Semantics

Several issues have been identified during our formalization process of UML 2 activity diagrams. To leave some scope for elements which we consider controversial, we introduce a configuration mechanism for the semantics which we already described in [Sar05] (Section 4.1). Section 4.2 gives a short description of issues and enhancements of send signal and broadcast signal actions, which are also treated extensively in [Sar05]. The following sections describe problems that relate to errors and obscure information in the specification (Section 4.3), or missing information (Section 4.4).

4.1 Targeting controversial elements

The following sections discuss some problems where the intended semantics can only be guessed. As will be seen, multiple interpretations may be useful for different scenarios. To provide the necessary configuration properties in UML models to circumvent these deficiencies, we use tagged values, a standard mechanism for extending the UML [UML05]. Tags designate simple key/value pairs which add additional information to UML model elements. By interpreting our own tags, which will be introduced as required in the following, the execution engine can adjust its behavior at runtime accordingly. Therefore, all necessary information regarding the execution semantics is directly contained in the diagrams. This mechanism was introduced by us in [Sar05].

New tagged values are defined by introducing stereotypes [UML05]. The notation is shown in Figure 4.1, where a new stereotype with tags, which extends an existing UML meta class, is defined. We will extend various kinds of meta classes, such as “Activity”, or “InterruptibleActivityRegion”, which shall enable us to apply special tags to them. These tags will be queried by the ASM activity interpreter (see chapter 5). All semantic variation points introduced in this chapter have been implemented and tested in our tool (see chapter 6).

Figure 4.1: Defining tagged values

4.2 Problems and enhancements of signals

Problems relating to buffering, targeting and distribution of signals have already been discussed in [Sar05] and will not be treated further, here. We, therefore, will only summarize the problems and briefly describe our approach to solve them.

Specification of signal targets. Target objects for send signal actions are normally provided by an input pin. This often leads to unnecessary preceding actions, whose only purpose is to provide this object parameter, although it could often be derived from the current object configuration. We, therefore, provide the possibility to annotate send signal actions with expressions to make the target explicit. To this end, we use a subset of XPath [Kay04] for querying our current object graph, resulting in a set of nodes where the signal should be sent to. Thus, the targets are more explicit, and also guarantee flexibility by admitting parameters in the expressions. Querying object graphs by means of XPath expressions has been introduced by [Sax03] and also applied to meta models by [SG05].

Buffering of signals. Signals are always buffered in the event queue of an object, according to the UML specification [UML05]. It may, however, sometimes be desirable to also disable buffering [Sar05]. We, therefore, allow the developer to either deactivate buffering for specific kinds of signals or for all signals of an activity. Our molding-press case study (see Section B.2) makes use of this feature.

Distribution of signals. Multiple behaviors can be associated with one context object at the same time. This is due to nesting of activities by using call behavior actions, which "reuse" the current context object for its execution, unless configured otherwise (see Section 4.4.1). According to [UML05, p. 229], an event is only consumed by one action and one behavior:

> "If the accept event action is executed and the object detected an event occurrence matching one of the triggers on the action and the occurrence has not been accepted by another action or otherwise consumed by another behavior, then the accept event action completes and outputs a value describing the occurrence."

Thus, on the one hand, multiple action executions compete for event occurrences, which is correctly handled by our rules as becomes evident in chapter 5. On the other hand, there is also competition among multiple activity executions for an event, which is not realized in our implementation. This is due to the fact that we associate event queues with activity executions rather than with context objects, which store event occurrences for multiple behavior executions (the reasons for this are discussed in Section 5.9.1). To be able to send signals to "nested" activities anyway, we introduce a signal "distribution" tag, which indicates whether signal objects are distributed among nested activity executions.

Broadcasting of signals. Other issues relate to broadcast signal actions. It is not possible to model a broadcast of signals graphically with UML 2, since there is no symbol proposed for this type of action. We, therefore, propose to use a new symbol, which is a twofold SendSignalAction (see [Sar05]). Since the determination of targets of broadcast signal action is considered an "official" semantic variation point, we propose to use the same XPath-mechanism which has already been described above for send signal actions.

Figure 4.2 shows the stereotypes and tagged values, which we defined for signal configuration.

4.3 Problems due to errors and obscure information

This section deals with errors and ambiguous information in the UML specification.

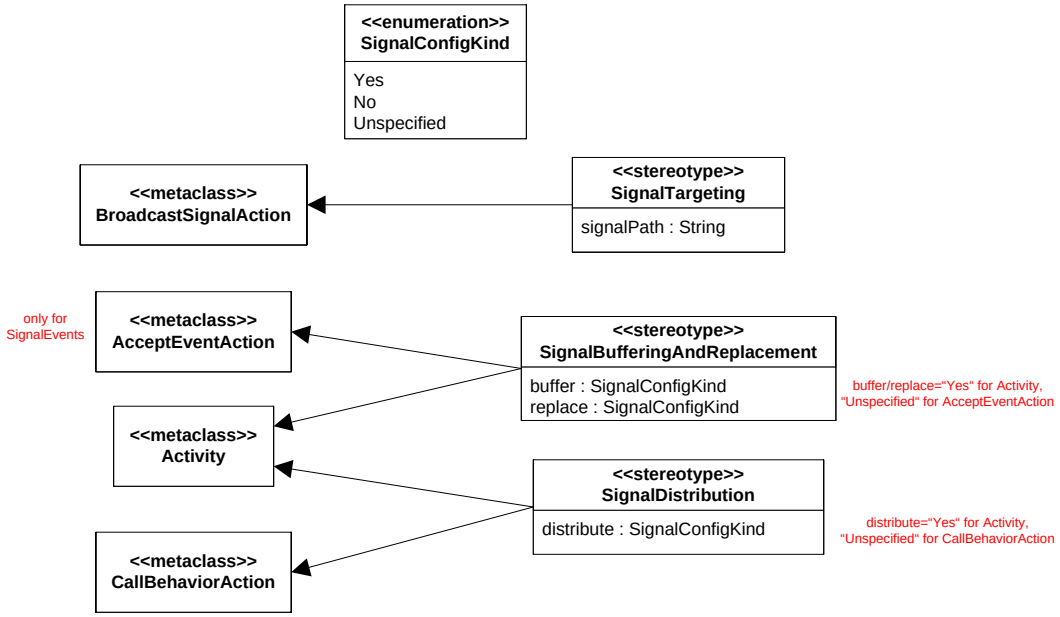

Figure 4.2: Tag definitions for semantic variation points for signals

4.3.1 Unclear terms

Multiple terms are neither clearly defined nor consistently used in the UML specification. Relevant in our context are especially the notion of "target", "traverse", and "accept", because they relate to transition computation and execution. Our ASM formalization of transitions is described in Sections 5.7 and 5.8.

The specification describes that [UML05, p. 309]

> "Edges have rules about when a token may be taken from the source node and moved to the target node. A token traverses an edge when it satisfies the rules for target node, edge, and source node all at once. This means a source node can only offer tokens to the outgoing edges, rather than force them along the edge, because the tokens may be rejected by the edge or the target node on the other side. ... Tokens cannot rest at control nodes, such as decisions and merges, waiting to move downstream. Control nodes act as traffic switches managing tokens as they make their way between object nodes and actions, which are the nodes where tokens can rest for a period of time. Initial nodes are excepted from this rule."

Since "a token traverses an edge when it satisfies the rules for target node, edge, and source node all at once" and due to the fact that tokens cannot rest at control nodes, it is not quite clear how these statements are to be applied when considering multiple consecutive edges with intermediary control nodes. No information is given on what condition the token "satisfies" the "rule" of the control node. One can imagine that all control nodes accept tokens. For join nodes, however, it may also be assumed that tokens are accepted if the *join condition* is met, i.e., there are tokens on all incoming edges of the join (see [UML05, p. 369]). Even worse, another term, "accept" comes into play [UML05, p 363]

> "Tokens arriving at a fork are duplicated across the outgoing edges. If at least one outgoing edge accepts the token, duplicates of the token are made and one copy traverses each edge that accepts the token."

In addition to edges, targets can also "accept" tokens [UML05, p. 349] (description of decision nodes)

> "Multiple edges may be offered the token, but if only one of them has a target that accepts the token, then that edge is traversed."

It would be useful to define these terms and above all provide examples of how they are used in a non-local context (i.e. not only for a single edge). The prerequisite for the execution of transitions is that the "final" target action or object node is able to receive a token. We, therefore, introduce the term *destination* for those nodes. We use the term "target" for the, intended, local context and avoid the term "accept". "Traversal" always refers to the complete path from the original source to the destination node.

4.3.2 Where to hold control tokens

For data tokens it is evident that they are held by object nodes, such as output pins, or central buffers. For control tokens, on the other hand, it is not obvious where they are actually stored. For initial nodes [UML05, p. 365] states:

> "A control token is placed at the initial node when the activity starts, ... Tokens in an initial node are offered to all outgoing edges."

Multiple edges can have an initial node as a source, and we also assume that these edges are independent from each other, though not explicitly stated (nor excluded) by the specification. Thus, according to the above quotation, the question is when to remove the single control token from an initial node. Certainly, it must be deleted if a token has moved to a destination node, but in the following it can no longer pass other outgoing edges of the initial node. We thus propose to store tokens *on* the outgoing edges of initial nodes. Each of these tokens is offered to the respective initial edge.

For actions, it even is not clear where control tokens are actually held, since no details are given by the specification, except [UML05, p. 302]:

> "When completed, an action execution offers object tokens on all its output pins and control tokens on all its outgoing control edges."

We, therefore, also hold control tokens directly on the outgoing edges of actions. These decisions are reflected by the *controlTokens*-function in our formalization which is specified in Section 5.2.

4.3.3 Confusion of the reader due to distributed information

Accept event actions without incoming edges are started with the onset of the activity. This is stated in [UML05, p. 229]:

> "If an AcceptEventAction has no incoming edges, then the action starts when the containing activity or structured node does, whichever most immediately contains the action."

The reader, however, is confused when further considering the description of interruptible activity regions [UML05, p. 367]:

> "AcceptEventActions in the region that do not have incoming edges are enabled only when a token enters the region, even if the token is not directed at the accept event action."

Thus, not *every* accept event action is activated when the activity starts. Ambiguity often arises, because related information is distributed among multiple UML elements in the specification. Thus, the ambitious reader can never be sure to have understood an element without considering the whole document.

4.3.4 Termination of accept event actions without incoming edges

As described before, accept event actions without incoming edges contained in interruptible activity regions are enabled when a flow enters the region. It is, however, not clear when these actions are *terminated*, because, according to [UML05, p. 229], they stay active (infinitely?):

> "In addition, an AcceptEventAction with no incoming edges remains enabled after it accepts an event. It does not terminate after accepting an event and outputting a value, but continues to wait for other events."

It is surely reasonable that these actions do not stay enabled after the region has been interrupted. This is reflected by our rules in Section 5.6.5.3.

4.3.5 Actions without incoming edges

Actions without incoming edges are started when the associated activity starts [UML05, p. 365]. This statement is inappropriately located in the description of initial nodes (and not in the description of actions). Besides, it runs contrary to accept event actions regarding interruptible activity regions. Accept event actions are also started when the activity starts, but only if they are *not* contained in a region as described in Section 4.3.4. Since "accept event action" subsets "action", it is no longer clear what should be done with other types of actions in interruptible activity regions. We decided to activate all actions except accept event actions without incoming edges, even if they are contained in those regions.

4.3.6 Actions without incoming edges but with input pins

Another confusion arises when considering actions without incoming edges but with input pins. It does not make sense to start these actions without any data, thus we do not start these actions when the activity starts.

4.3.7 Data tokens outrun control tokens

The steps of starting an action are detailed in [UML05, p. 302]:

> "The steps of executing an action with control and data flow are as follows:
>
> 1. An action execution is created when all its object flow and control flow prerequisites have been satisfied (implicit join). ... The flow prerequisite is satisfied when all of the input pins are offered tokens and accept them all at once, precluding them from being consumed by any other actions. This ensures that multiple action executions competing for tokens do not accept only some of the tokens they need to begin, causing deadlock as each execution waits for tokens that are already taken by others.
> 2. An action execution consumes the input control and object tokens and removes them from the sources of control edges and from input pins. The action execution is now enabled and may begin execution. If multiple control tokens are available on a single edge, they are all consumed."

Consider Figure 4.3. In a) the control and object flow prerequisites are given, thus an action execution can be created (see b). Data tokens are moved to the input pins of action B, but the control token which was used for creation is still located at the source edge. Since this control token already contributed to the activation of action B, it should be removed. Otherwise, the control token would continue to create further executions – an unwelcome effect in this case – since it has already been "taken" before. Further confusion arises when incorporating a guard that changes after an action execution has been created. In c) the guard switched to "false", and traversal of the control token over this edge when enabling the execution is not immediately obvious.

We, therefore, call this problem "outrun of control tokens". In our formalization in chapter 5, we deviate from the specification by removing control tokens from source edges when creating action executions.

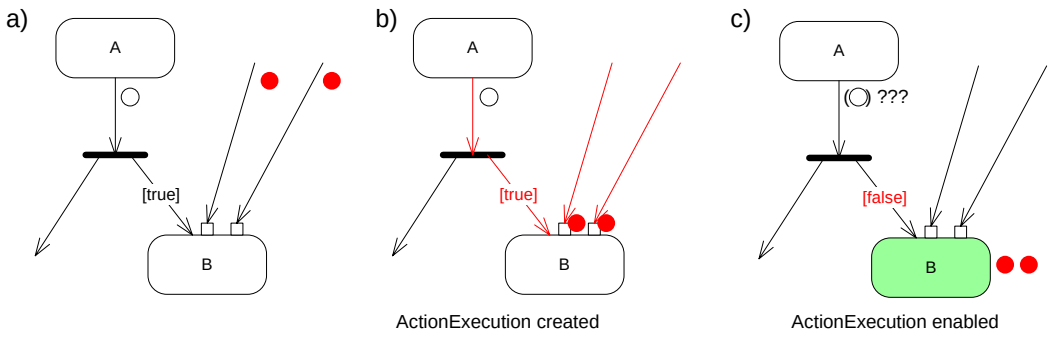

Figure 4.3: Faster data tokens

4.3.8 Buffering of tokens at fork nodes

Buffering of tokens at fork nodes has been introduced late in the specification process. In [UML03, p. 334]

> "Tokens arriving at a fork are duplicated across the outgoing edges. Tokens offered by the incoming edge are all offered to the outgoing edges. When an offered token is accepted on all the outgoing edges, duplicates of the token are made and one copy traverses each edge."

This has been changed in the subsequent intermediate [UML04] and final version of the specification. The final version describes [UML05, p. 363], that

> "Tokens arriving at a fork are duplicated across the outgoing edges. If at least one outgoing edge accepts the token, duplicates of the token are made and one copy traverses each edge that accepts the token. The outgoing edges that did not accept the token due to failure of their targets to accept it, keep their copy in an implicit FIFO queue until it can be accepted by the target. The rest of the outgoing edges do not receive a token (these are the ones with failing guards)."

Our motivation for the introduction of buffering is shown in Figure 4.4 a). Suppose that action A offers a token, and action D does not yet. Without buffering, action B starts and consumes the token, and action C cannot start if D now offers a token. This is possible, however, if the token is buffered at the right outgoing edge of the fork node. Figure 4.4 b) illustrates a problem with buffering at outgoing edges of fork nodes. If $x > 0$ when A emits a token, C cannot start due to a missing token on the right incoming edge. Action B starts, and the token is buffered at the outgoing edge of the fork node. Using the buffered token, C can start as soon as a token is provided on the other incoming edge, but the guard may have switched to "false" in the meantime. This problem can be compared to the "outrun of control tokens" presented in Section 4.3.7.

Buffering of tokens at fork nodes would also break the intuitive notion of control nodes acting as "traffic switches". Nevertheless we see that buffering is useful but propose to hold these tokens on incoming edges of destination nodes rather than on outgoing edges of fork nodes. If the whole path is open from the source node to a destination node that currently cannot consume the token, it is buffered directly at the destination node. This reduces the issues with guards to a certain extend.

4.4 Problems due to missing information

4.4.1 Context object for call behavior action

A call behavior action takes a behavior to invoke as an argument. Parameters of the action are mapped to corresponding activity parameter nodes. Although not stated explicitly, an activity

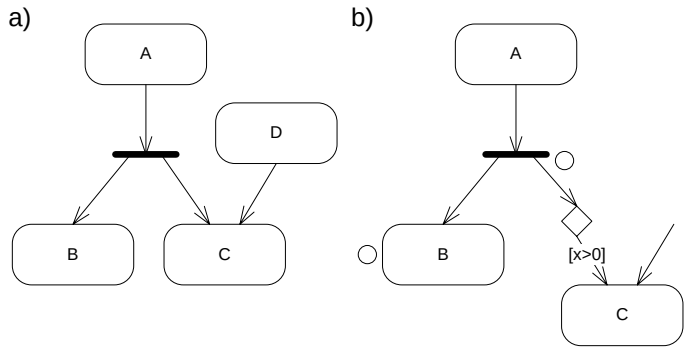

Figure 4.4: Motivation and problem with buffering at fork nodes

execution has to be created, which represents the invocation. It is, however, not clear from the specification which context object is to be used for this execution. Without any context object, guards cannot refer to its attributes and relationships. It is also not useful to create a new context object on the fly.

It is therefore necessary to identify the context object for a call behavior action. Thus, we use the context object of the *current* activity execution for the newly created execution. However, we wish to give the developer freedom by defining an explicit semantic variation point for the determination of context objects. To this end, we define a tagged value called "context", which can be applied to call behavior actions (see Figure 4.4.1). We allow for XPath [Kay04] expressions for the computation of the context in the same way as we use them for the configuration of signal targets [Sar05]. The context object to use is computed at runtime by evaluating the expression which navigates the current object graph.

Figure 4.5: Definition of the semantic variation point for context objects

4.4.2 Which transitions to execute

The specification does not include any information on how possible transitions are determined and how many are executed concurrently. We rely on the nondeterminism of our chosen formalism – Abstract State Machines – to provide the maximum freedom in our specification. We also provide a slight modification of the rules, to obtain the execution of all possible transitions (see discussion in Section 5.7.9). Our tool implementation takes this approach, due to practical reasons.

4.4.3 Multiple callers with "isSingleExecution"

If an activity is marked as "isSingleExecution" only one execution of this activity exists at a time [UML05, p. 307]. Problems arise if multiple call behavior actions call this kind of activity concurrently. When the activity execution terminates, all calling actions must terminate at once. This scenario is not discussed in the specification. Besides, the specification allows for *lower* and *upper* specifications on output pins, which determine the minimum and maximum number of tokens that have to be available for termination. These specifications must not be in conflict with each other for call behavior actions. We allow for multiple callers but, for the sake of simplicity, we define $lower = upper = 1$ for output pins of call behavior actions.

4.4.4 Interruptible activity regions

If an interrupting edge is passed, all actions contained within the interrupted region are aborted, and all tokens are removed. In Figure 4.6 a), flow 3 is removed if the interrupting edge 1 is passed. Concurrent transitions beyond interrupted regions are, however, executed (flow 2) [UML05, p. 367].

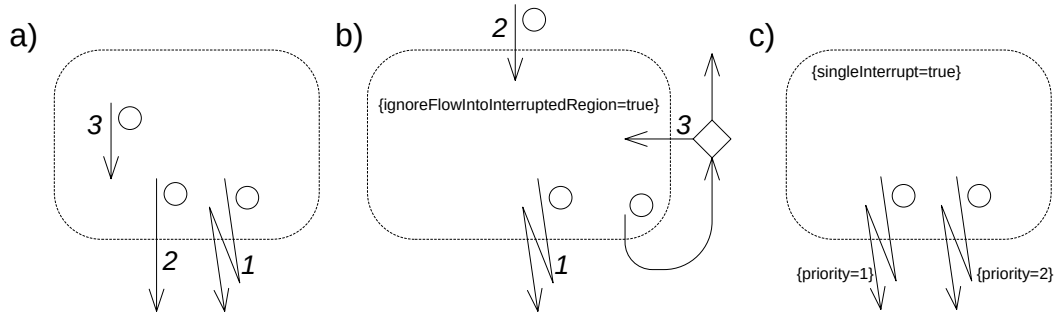

Figure 4.6: Multiple interrupting edges

4.4.4.1 Flows entering interruptible activity regions

One problem relates to flows into interrupted regions. When considering Figure 4.6 b), it is unclear whether or not the concurrent flows 2 and 3 have to be removed, when the interrupting edge 1 is being passed. Since both cases, allowing and preventing the transitions, are conceivable, we define a variation point on how to handle flows entering interrupted regions. If "ignoreFlowIntoInterruptedRegions" is set to "true", incoming flows are removed when a region is interrupted.

4.4.4.2 Multiple interrupting edges

Multiple edges can interrupt an interruptible activity region. Since, according to the specification, concurrent transitions are not interrupted, it can be assumed that concurrent interrupting edges are also passed if tokens are available. We, however, think that there are scenarios where it may be useful that only one interrupting edge is passed if mutually exclusive paths are to be taken. We thus also provide a semantic variation point for specifying the intended behavior. In Figure 4.6 c), this is shown by introducing a "singleInterrupt" tag for the region, and "priority"-tags for the edges. If "singleInterrupt" is set to "true", only the interrupting edge with the highest priority is passed.

4.4.4.3 Aborting nested interruptible activity regions

Interruptible activity regions, being activity groups, are also allowed to be nested. A major deficiency of the UML specification is missing information about how to deal with them. According to the specification [UML05, p. 323],"no node or edge in a group may be contained by its subgroups or its containing groups, transitively". This means that, when a region is aborted, its nested regions are not. To avoid this unexpected behavior we propose to interrupt all nested regions.

4.4.4.4 AcceptEventActions in nested interruptible activity regions

According to [UML05, p. 367],

> "AcceptEventActions in the region that do not have incoming edges are enabled only when a token enters the region, even if the token is not directed at the accept event action."

It is, however, not clear how to treat accept event actions without incoming edges in nested regions. We present another variation point, called "regionActivationPolicy", which has options of activating accept event actions on flows into the containing ("OnRegionFlow") or parent region ("OnParent-Flow"). A similar tagged value is applied for re-activating accept event actions: if such an action does not have any incoming edges, it remains active [UML05, p. 299]. In our opinion, this is only useful as long as there are tokens in the containing region or parent region. We thus also permit the configuration of the "regionReactivationPolicy".

4.4.4.5 All variation points for interruptible activity regions

All variation points relating to interruptible activity regions are shown in Figure 4.7.

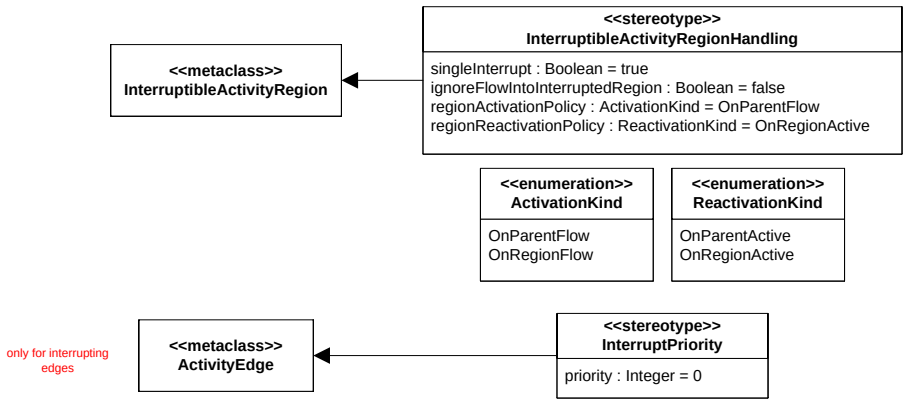

Figure 4.7: Definition of semantic variation points for *InterruptibleActivityRegion*

Chapter 5

An ASM Semantics for UML 2 Activity Diagrams

This chapter describes the ASM semantics for the subset of UML 2 activity diagrams as defined in Section 2. The resulting rules can be traced back to requirements present in or absent from the UML specification. The ASM formalization reveals topics where the UML specification is unclear or not intuitional, and serves as a basis for tool support.

Section 5.1 introduces relevant terms and gives an overview of the mapping approach. Section 5.2 defines basic ASM domains and the mapping of the UML 2 meta model to ASMs. Tags for configuring the semantics are given in Section 5.3. The initialization of the ASMs is presented in Section 5.4. Section 5.5 deals with activities and the event-handling mechanism. The life cycle of Actions, including execution of the supported kinds of actions, *CallBehaviorAction*, *CallOperationAction*, *AcceptEventAction*, *SendSignalAction* and *BroadcastSignalAction*, is described in Section 5.6. The semantics of token flow and execution of transitions is presented in Sections 5.7 and 5.8. Finally, Section 5.9 discusses related work in formalizing UML activity diagrams and gives an outlook for possible extensions and further work.

We specify the ASM domains and macros mostly in a top-down-approach, i.e. elements are often used before they are defined in our specification. In the PDF-version of this thesis, hyperlinks can be used to navigate between the ASM rules. Rule calls reference the definition of the rules' body, but only if it is *not* contained in the current section. All ASM functions are written in *italics*, and domain names are additionally colored *blue*. ASM operators are marked in **bold**.

5.1 Overview

Classifiers and behaviors

According to our vision of Model-Driven Development described in Section 1.2, we use class diagrams for modeling the static structure of a system. The control flow of the application (i.e., the behavior) is modeled with UML 2 activity diagrams, which are associated with "active" classes. Figure 5.1 shows the relevant part of the UML 2 meta model for behaviored classifiers (taken from [UML05, p. 412]). These classifiers, which are classes in our case, can have an associated behavior. Whereas UML 2 behaviors also comprise interactions and state machines, we only support activities as behaviors. Regarding the relationship between behaviored classifiers and behaviors, the UML specification also states that [UML05, p. 308]

> "The classifier, if any, is referred to as the context of the activity. At runtime, the activity has access to the attributes and operations of its context object and any objects linked to the context object, transitively."

Therefore, guards of activity edges can refer to attributes in the related context class, and actions can invoke its methods.

Actions

Activities contain actions, which specify transformations on the state of the system, and are not

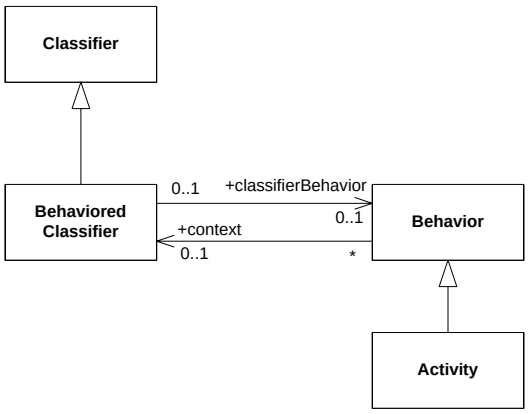

Figure 5.1: UML 2 meta model of behaviored classifiers

further decomposed. The specification proposes a multitude of different types of actions. As we want to leave ordinary programming tasks to the developer and only provide a higher-level view of the control flow by using activity diagrams, we think it is not useful to support fine-grained actions. Examples of such actions include *ReadVariableAction* [UML05, p. 267], which retrieves the values of a variable, or *CreateLinkAction* [UML05, p. 243], which is used to link objects. These tasks can be implemented more efficiently by using a normal programming language. We, therefore, support only the following types of actions:

CallBehaviorAction [UML05, p. 237]
: Invokes a behavior (i.e. an activity in our case). Recursive calls are allowed.

CallOperationAction [UML05, p. 239]
: Invokes a method. Methods contain custom code, written in an ordinary programming language.

AcceptEventAction [UML05, p. 228]
: Waits for an event to occur. We support signal and time events.

SendSignalAction [UML05, p. 273]
: Creates a signal instance from its inputs and sends it to a specific activity execution.

BroadcastSignalAction [UML05, p. 235]
: Sends a signal instance to multiple activity executions.

Section 5.6 deals with the life cycle of actions.

Executions

Similar to objects representing instances of classes at runtime, "executions" symbolize runtime entities of activities and actions [UML05, p. 285]:

> "An action execution corresponds to the execution of a particular action. Similarly, an activity execution is the execution of an activity, ultimately including the executions of actions within it. Each action in an activity may execute zero, one, or more times for each activity execution."

Each instance of an active (i.e. behavioral) class has an associated activity execution. Figure 5.2 shows the relationship between behaviored classes, activities, actions and their executions. Activity and action executions are mapped to asynchronously executing ASM agents. Each activity execution owns an event queue, which is polled by the *activity execution controller*, the ASM agent for an activity execution. Action executions use this queue to communicate with the corresponding activity execution, e.g. to inform it when the action execution has terminated. Signal events sent to an

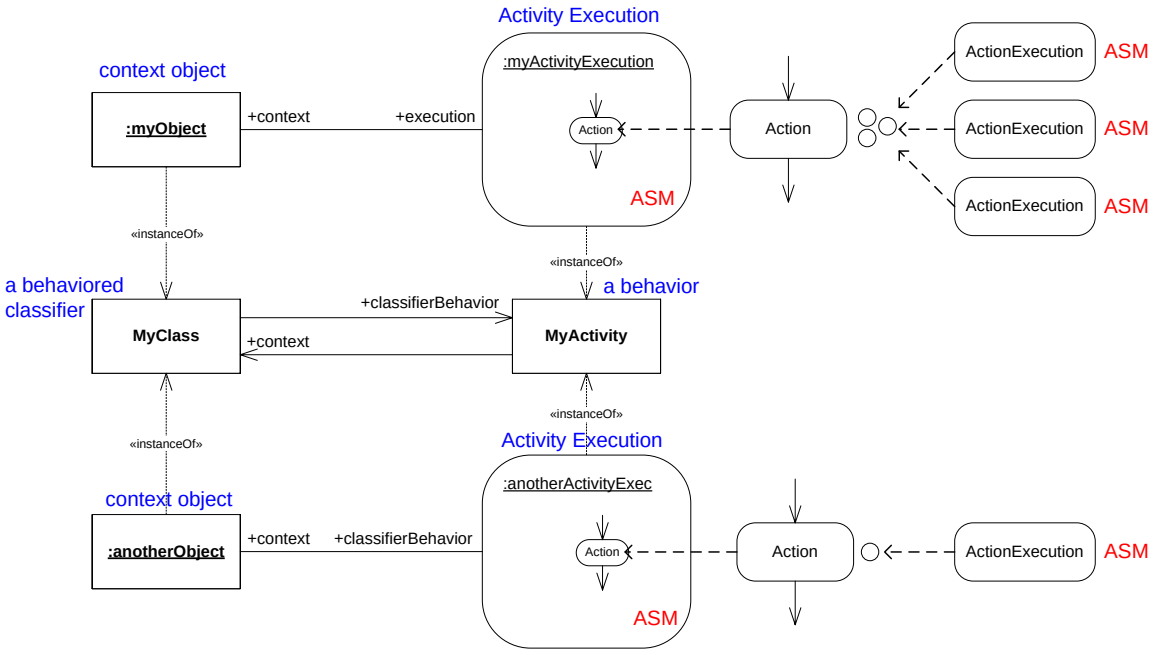

Figure 5.2: Mapping UML activity diagrams to Multi-Agent ASMs

activity execution are also stored in its event queue. Event handling and the controller loop are dealt with in Sections 5.5.1 and 5.5.2.

Interpreter approach

We use an interpreter approach for executing activities with ASMs. The considered subset of the UML 2 meta model for activity diagrams is translated to ASM domains. These domains are initialized with instances of concrete, syntactically and semantically correct activity diagrams to be executed and do not change during execution of the activities (and are therefore **static** domains). The mapping is presented in Section 5.2.2. We decided to use interpretation rather than compilation because it is easier to understand and less prone to errors. Moreover, modification of the ASM rules and establishing new ones is simplified, which is of particular importance when considering revisions and future versions of the UML specification.

Token flow

As already mentioned in chapter 2, the semantics of activities is based on token flow [UML05, p. 308]. Object nodes, like input pins for actions or central buffer nodes, can hold data tokens, which flow along object flow edges. Control tokens can rest on actions and initial nodes, and flow along control flow edges. The current positions of all data and control tokens constitute the current configuration of an activity execution. Tokens resting on nodes can lead to transitions, which in turn can lead to the execution of other actions. ASM rules for the computation and execution of transitions are described in Sections 5.7 and 5.8.

We first present basic definitions, including the mapping of the UML 2 meta model to ASM domains, in the following section.

5.2 Basic definitions

5.2.1 Predefined base domains

We introduce the following common domains for basic data types, including natural numbers, strings, boolean values and a not further specified time domain.

static domain *Nat* natural numbers
static domain *String* strings

static domain *Boolean* boolean values
static domain *Time* time values

We also define a domain for behaviored classifiers, together with a function *classifierBehavior* which maps to the association between "Behaviored Classifier" and "Behavior" shown in Figure 5.1. To model instances of a behaviored classifier we use the *BehavioredObject* domain, which is a subset of a domain for all objects. The *classOf* function returns the classifier for a specified instance. We use these domains and function for starting the interpreter in Section 5.4. New activity executions are started for elements of the domain *BehavioredObject*.

shared domain *Classifier*
shared domain $BehavioredClassifier \subseteq Classifier$
static $classifierBehavior : BehavioredClassifier \rightarrow Behavior$
shared domain *Object*
shared domain $BehavioredObject \subseteq Object$
static $classOf : Object \rightarrow Classifier$

We deliberately do not conform to the UML specification regarding the mapping of classes and instances (there is no UML element called *BehavioredObject*), because we focus on the accurate mapping of activity diagrams, and do not intend to provide a comprehensive formalization of other UML packages. The UML Semantics Project [UML06] aims at providing such "a definitive and complete formal semantics foundation for the UML2.0 standard".

Actions of type *CallBehaviorAction* invoke methods on objects. To leave open how programming language code is represented, we introduce an abstract *Code* domain for executable code.

static domain *Code*

5.2.2 UML 2 meta model to ASM mapping

This section presents the mapping of the used subset of the UML meta model, which was introduced in Figure 2.3, to ASM domains and functions. Each class from the meta model is mapped to an ASM **domain**. Attributes and associations are translated to ASM functions, which yield the particular value. An example is shown in Figure 5.3, which maps to the following ASM rules:

static domain $ActivityNode =_{def} \ldots$ *(see below for a complete definition)*
static $incoming, outgoing : ActivityNode \rightarrow \mathcal{P}(ActivityEdge)$
static domain *ControlFlow*
static domain *ObjectFlow*
static domain $Element =_{def} ActivityEdge \cup \ldots$
static domain $ActivityEdge =_{def} ControlFlow \cup ObjectFlow$
static $source, target : ActivityEdge \rightarrow ActivityNode$

Note that all domains are **static**, meaning that they do not change during the execution of the ASM rules. This is because we assume the domains to be initialized with concrete instances of activities, before the actual execution of the interpreter begins.

Activity

The following ASM functions represent the subset of the UML 2 meta model for activity diagrams, illustrated in Figure 2.3. These functions will be used by the interpreter for executing the diagrams. We start by modeling the "Activity" class, together with its collections of nodes (*node*) and edges (*edge*). If an activity is marked as "isSingleExecution" [UML05, p. 307], there exists only one execution of the activity. Further invocations do not result in the creation of new activity executions, but in issuing new tokens in the existing instance. Activities can also contain groups, interruptible activity region being the only group we support.

static domain *Activity*
static $node : Activity \rightarrow \mathcal{P}(ActivityNode)$
static $edge : Activity \rightarrow \mathcal{P}(ActivityEdge)$
static $isSingleExecution : Activity \rightarrow Boolean$
static $group : Activity \rightarrow \mathcal{P}(ActivityGroup)$

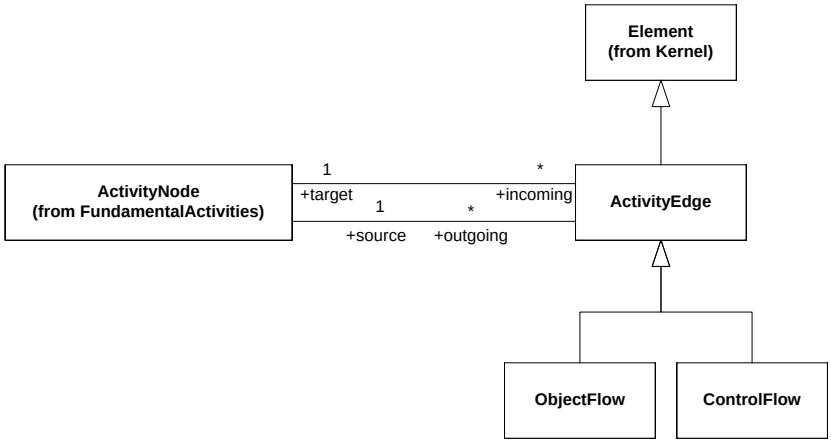

Figure 5.3: Mapping the UML 2 meta model to ASMs

We also bring in a domain for the *Behavior* class. Behaviors can have parameters [UML05, p. 417], activity parameter nodes for activities map to. *Behavior*

static domain *Behavior* $=_{def}$ *Activity*
static *ownedParameters* : *Behavior* $\rightarrow$ *Parameter**

The next paragraphs deal with actions. According to [UML05, p. 213] *Action*

> "An action is the fundamental unit of behavior specification. An action takes a set of inputs and converts them into a set of outputs, though either or both sets may be empty."

We introduce call behavior actions, which call other behaviors (i.e. activities). Call operation actions represent method invocations, the target object is provided by the *target* parameter. The operation to call is specified by *operation*. Both types of actions derive from the abstract class "CallAction", which provides an attribute indicating whether the call is synchronous or asynchronous as well as a list of results.

static domain *CallBehaviorAction*
static *behavior* : *CallBehaviorAction* $\rightarrow$ *Behavior*
static domain *CallOperationAction*
static *target* : *CallOperationAction* $\rightarrow$ *InputPin*
static *operation* : *CallOperationAction* $\rightarrow$ *Operation*
static domain *Operation*

static domain *CallAction* $=_{def}$ *CallBehaviorAction* $\cup$ *CallOperationAction*
static *result* : *CallAction* $\rightarrow$ *OutputPin**
static *isSynchronous* : *CallAction* $\rightarrow$ *Boolean*

A send signal action creates a signal instance of the type specified in *signal* and sends it to the *target* object. It inherits from "InvocationAction", which defines an *argument* function. The attributes used for creating the signal instance are provided by this function. A broadcast signal action transmits a signal to multiple targets. Since the computation of those targets is left open in the UML specification, there is no appropriate parameter specified. We will use a **monitored** function, defined in Section 5.6.5.4, which provides those target objects.

Accept event actions wait for events to occur, as determined by *trigger*. We deviate from the specification in this point by only allowing exactly one trigger for each accept event action (the meta model allows a set of triggers). The reason is, that graphical modeling of those actions with multiple triggers is not feasible.

Actions use the *input* and *output* collections of pins, constituting their input and output parameters. Data tokens can flow into input pins and out of output pins. The before-mentioned *argument* and *result* functions map to subsets of *input* and *output*. This is indicated by "subsets"-properties in the meta model, e.g. [UML05, p. 217].

static domain *SendSignalAction*
static *target* : *SendSignalAction* → *InputPin*
static *signal* : *SendSignalAction* → *Signal*
static domain *BroadcastSignalAction*
static *signal* : *BroadcastSignalAction* → *Signal*

static domain $InvocationAction =_{def} CallAction \cup SendSignalAction \cup BroadcastSignalAction$
static *argument* : *InvocationAction* → $InputPin^*$

static domain *AcceptEventAction*
static *trigger* : *AcceptEventAction* → *Trigger*

static domain $Action =_{def} InvocationAction \cup AcceptEventAction$
static *input* : *Action* → $InputPin^*$
static *output* : *Action* → $OutputPin^*$

ControlNode

Control nodes determine the token flow of activity diagrams. Token flow starts at initial nodes. Decision nodes route tokens by using guards, which refer to attributes of the context object of the current activity execution, or – in case of data tokens – to the token value. Merge nodes and join nodes merge multiple flows into one, whereas join nodes also synchronize the incoming flows. Fork nodes split one flow into multiple parallel flows, and token flow terminates at final nodes. Section 5.7 on flow computation discusses the semantics of each type of node in detail.

static domain *DecisionNode*
static domain *MergeNode*
static domain *ForkNode*
static domain *JoinNode*
static domain *InitialNode*

static domain *ActivityFinalNode*
static domain *FlowFinalNode*
static domain $FinalNode =_{def} ActivityFinalNode \cup FlowFinalNode$

static domain $ControlNode =_{def} DecisionNode \cup MergeNode \cup ForkNode \cup JoinNode \cup FinalNode \cup InitialNode$

ActivityEdge

Activity edges are used to specify the order of execution of actions. Control tokens flow along control edges, and data tokens flow along object flow edges, which target object nodes, such as input pins. The *source* and *target* functions determine the source and target node. We extend the specification by allowing edges to interrupt multiple interruptible activity regions (as specified by the *interrupts* function), rather than only one. Each edge can also have a *guard*, which is especially used for the decision nodes as described in chapter 2.

static domain *ControlFlow*
static domain *ObjectFlow*

static domain $ActivityEdge =_{def} ControlFlow \cup ObjectFlow$
static *source*, *target* : *ActivityEdge* → *ActivityNode*
static *interrupts* : *ActivityEdge* → $\mathcal{P}(InterruptibleActivityRegion)$
static *guard* : *ActivityEdge* → *String*

ObjectNode

Object nodes are sources or targets of object flows and hold data tokens. As introduced above, input pins and output pins define parameters and results of actions. Pins can have *lower* and *upper* specifications, which determine the minimum and maximum number of data tokens taken for a single execution of an action. The default values are 1. Central buffer nodes act as an intermediate buffer for data tokens, and activity parameter nodes model incoming and outgoing parameters for activities. Activity parameter nodes map to the corresponding parameters of the behavior representing the activity. All object nodes can have an *upperBound*, which defines the maximum number of tokens the node can hold. The upper bound defaults to infinite.

static domain *CentralBufferNode*

static domain *ActivityParameterNode*
static *parameter* : *ActivityParameterNode* → *Parameter*
static domain *ParameterDirectionKind* $=_{def}$ {in, inout, out, return}
static domain *Parameter*
static *direction* : *Parameter* → *ParameterDirectionKind*

static domain *InputPin*
static domain *OutputPin*
static domain *Pin* $=_{def}$ *InputPin* ∪ *OutputPin*
static *upper*, *lower* : *Pin* → *Nat*

static domain *ObjectNode* $=_{def}$ *CentralBufferNode* ∪ *Pin* ∪ *ActivityParameterNode*
monitored *upperBound* : *ObjectNode* → *Nat*

ActivityNode

ActivityNode is the common base class for actions, control nodes and object nodes. The functions *incoming* and *outgoing* return sets of incoming resp. outgoing activity edges. To determine the interruptible activity regions where a node is contained in, *inInterruptibleRegion* is used.

static domain *ActivityNode* $=_{def}$ *Action* ∪ *ControlNode* ∪ *ObjectNode*
static *incoming*, *outgoing* : *ActivityNode* → $\mathcal{P}$(*ActivityEdge*)
static *inInterruptibleRegion* : *ActivityNode* → $\mathcal{P}$(*InterruptibleActivityRegion*)

Interruptible-Activity-Regions

Interruptible activity regions are a subclass of activity group and denote a set of nodes (stored in *containedNode*) which are interrupted when interrupting edges are passed. Interrupting edges can be control or object flow edges. Interruptible regions can be nested by using *subGroup* and *superGroup*.

static domain *InterruptibleActivityRegion*
static domain *ActivityGroup* $=_{def}$ *InterruptibleActivityRegion*
static *superGroup* : *ActivityGroup* → *ActivityGroup*
static *subGroup* : *ActivityGroup* → $\mathcal{P}$(*ActivityGroup*)
static *containedNode* : *ActivityGroup* → $\mathcal{P}$(*ActivityNode*)

Triggers and Events

Accept event actions are triggered upon occurrence of a given event. We support signal and time events. A signal event is issued by send signal or broadcast signal actions, described above. Time events can be relative, i.e. the event is issued when a relative time span (specified by *when*) has elapsed. If not relative, *when* denotes an absolute time.

static domain *Signal*
static domain *Trigger*
static *event* : *Trigger* → *Event*
static domain *TimeEvent*
static *isRelative* : *TimeEvent* → *Boolean*
static *when* : *TimeEvent* → *Time*
static domain *SignalEvent*
static *signal* : *SignalEvent* → *Signal*
static domain *MessageEvent* $=_{def}$ *SignalEvent*
static domain *Event* $=_{def}$ *TimeEvent* ∪ *MessageEvent*

Miscellaneous The superclass of all UML classes is "Element". Value specifications denote arbitrary expressions, which are used when evaluating UML tags for configuring the execution semantics.

static domain *Element* $=_{def}$ *Behavior* $\cup$ *ActivityGroup* $\cup$ *ActivityEdge* $\cup$ *ActivityNode*
static domain *ValueSpecification*

5.2.3 Abbreviations

This section defines domains that act as abbreviations for a set of existing domains from the UML meta model. These domains are used throughout this chapter.

static domain *ActivityInputParameters* $=_{def}$ $\{n \in$ *ActivityParameterNode* $|$
$: n.parameter.direction \in \{\mathsf{in}, \mathsf{inout}\}\}$
static domain *ActivityOutputParameters* $=_{def}$ $\{n \in$ *ActivityParameterNode* $|$
$: n.parameter.direction \in \{\mathsf{inout}, \mathsf{out}, \mathsf{return}\}\}$

static domain *ControlFlowSource* $=_{def}$ *Action* $\cup$ *InitialNode*
static domain *ControlFlowDestination* $=_{def}$ *Action* $\cup$ *FinalNode*
static domain *ObjectFlowSource* $=_{def}$ *OutputPin* $\cup$ *CentralBufferNode*
$\cup$ *ActivityInputParameters*
static domain *ObjectFlowDestination* $=_{def}$ *InputPin* $\cup$ *CentralBufferNode*
$\cup$ *ActivityOutputParameters*
static domain *FlowSource* $=_{def}$ *ControlFlowSource* $\cup$ *ObjectFlowSource*
static domain *FlowDestination* $=_{def}$ *ControlFlowDestination* $\cup$ *ObjectFlowDestination*

We also define the following convenience functions, e.g. for accessing single input pins and edges. Note that some of our macros described later assume an arbitrary order of the incoming and outgoing edges of activity nodes. This is opposed to the UML specification but does not limit our approach in any way. The following functions $incoming(n, i)$ and $outgoing(n, i)$, which we do not define further here, return the i-th element of these ("ordered set" of) edges.

static *input* : *Action* $\times$ *Nat* $\rightarrow$ *InputPin*
$input(n, i) =_{def} elementAt(n, i)$
static *incoming*, *outgoing* : *ActivityNode* $\times$ *Nat* $\rightarrow$ *ActivityEdge*

static *predecessors* : *ActivityNode* $\rightarrow \mathcal{P}$(*ActivityNode*)
$predecessors(n) =_{def} \{p \in$ *ActivityNode* $\mid \exists e \in$ *ActivityEdge* $: source(e) = p \ \wedge \ target(e) = n\}$

static *IsControlFlow* : *ActivityEdge* $\rightarrow$ *Boolean*
$IsControlFlow(e) =_{def} e \in$ *ControlFlow*
static *IsObjectFlow* : *ActivityEdge* $\rightarrow$ *Boolean*
$IsObjectFlow(e) =_{def} e \in$ *ObjectFlow*

5.2.4 Configuration of activity executions

The current configuration of an activity execution is made up of control tokens and data tokens resting at activity nodes or edges. To model these tokens, we create appropriate domains for tokens. These domains are defined as **controlled** [BS03] to be local to a single activity execution. The function *value* yields the value for a data token. The domain *ValueSpecification* holds arbitrary values, such as numbers, strings, objects, and the like.

controlled domain *ControlToken*
controlled domain *DataToken*
static *value* : *DataToken* $\rightarrow$ *ValueSpecification*
controlled domain *Token* $=_{def}$ *ControlToken* $\cup$ *DataToken*

Control tokens can only rest on outgoing edges of actions or initial nodes, data tokens can only rest on object nodes. We, therefore, define the following functions, *controlTokens* and *dataTokens*, which model tokens on edges and nodes. Lists are used, because we assume a FIFO order [UML05, p. 380] on tokens. Initially, these functions are set to the empty list for all control flow edges resp. object nodes.

controlled $controlTokens : ControlFlow \rightarrow ControlToken^*$
initially $\forall e \in ControlFlow : controlTokens(e) = [\,]$
constraint $\forall e \in \{e \in ControlFlow \mid e.source \notin ControlFlowSource\} : controlTokens(e) = [\,]$

controlled $dataTokens : ObjectNode \rightarrow DataToken^*$
initially $\forall n \in ObjectNode : dataTokens(n) = [\,]$

To prevent conflicts in accessing these functions, we only allow the activity execution controller to move tokens. Concurrent access, therefore, is prevented. Terminating actions, for example, have to inform the controller by an "Action Termination" event. During processing of this event, the controller puts tokens on outgoing edges and output pins.

To determine whether a node offers a token (and, thus, a flow computation can be started, see Section 5.7), the function *HasToken* is used. It returns true, if control or data tokens are available on source edges or nodes.

$$HasToken : FlowSource \rightarrow Boolean$$
$$HasToken(n) =_{def} \begin{cases} |dataTokens(n)| > 0, & \textbf{if } n \in ObjectFlowSource \\ |\bigcup\{controlTokens(e) \mid e \in n.outgoing\}| > 0, & \textbf{if } n \in ControlFlowSource \\ undefined, & \textbf{otherwise} \end{cases}$$

The ASM macro Source is used to find the source node for a data token when executing transitions (Section 5.8). It returns *undefined* if the token was already removed, and thus **choose** fails to select a node. Note that there cannot be more than one object node holding the same token at any time. Since each token is cloned during transitions (i.e. moving a token produces a new one), a token is not reused for other nodes. Note that this is not true for values *stored* in data tokens (see *value* function), which are not copied during transitions.

Source : $DataToken \rightarrow ObjectFlowSource$
Source(t) $\equiv$
    choose n **with** $n \in ObjectNode \wedge t \in dataTokens(n)$
        result $:= n$

5.3 Model-based configuration of semantics

To allow for configuration of the interpreter semantics, we define the following domains. They represent the tagged values which we specified in chapter 4. The ASM function *tagValue* is used to query the model tags at runtime and adjust the interpreter behavior. This will become obvious later in the ASM rules which execute activity diagrams.

static domain *Stereotype*
initially $Stereotype = \{$SignalTargeting, SignalBufferingAndReplacement, SignalDistribution, CallContext, InterruptibleActivityRegionHandling, InterruptPriority$\}$
static domain *Tag*
initially $Tag = \{$signalPath, buffer, replace, distribute, context, singleInterrupt, ignoreFlowIntoInterruptedRegion, regionActivationPolicy, regionReactivationPolicy, priority$\}$

static $tagValue : Element \times Stereotype \times Tag \rightarrow ValueSpecification$

5.4 ASM initialization

This section deals with the initialization of the ASMs. The static domain *Program* contains all main rules which are executed by distinct ASMs.

static domain *Program*
initially *Program* = {
    INITIALIZE,
    ACTIVITYEXECUTIONCONTROLLER, (→ 5.5.2)
    EXECUTECALLBEHAVIORACTIONEXECUTION, (→ 5.6.5.1)
    EXECUTECALLOPERATIONACTIONEXECUTION, (→ 5.6.5.2)
    EXECUTEACCEPTEVENTACTIONEXECUTION, (→ 5.6.5.3)
    EXECUTESENDSIGNALACTIONEXECUTION, (→ 5.6.5.4)
    EXECUTEBROADCASTSIGNALACTIONEXECUTION (→ 5.6.5.4)}

The INITIALIZE macro is executed first and only once to create the initial activity executions for behavioral objects contained in *BehavioredObject*. We, therefore, define an initial agent called *init*, and assign the initialization macro as its main rule to the predefined *ASM* function [BS03].

initially *Agent* = {*init*}
initially *ASM*(*init*) = INITIALIZE

INITIALIZE ≡
    forall *object* **with** *object* ∈ *BehavioredObject*
        STARTNEWACTIVITYEXECUTION(*object.classOf.classifierBehavior*, *object*) (→ 5.5.3)

For each behavioral object, STARTNEWACTIVITYEXECUTION is called. The first parameter contains the behaviored classifier (i.e. the activity), an activity execution should be created for. The second parameter is the behavioral object, which will be assigned to the new execution as its context object.

The next section deals with the life cycle of activities, including creation, execution, termination and abort. The activity execution controller is described, and its event-handling mechanism.

5.5 Activity

Activity executions model instances of activities at runtime which are executed by distinct ASM agents. They, therefore, execute in parallel and have a local environment comprising their own token configuration (Section 5.2.4), event queue (Section 5.5.1) and action executions (Section 5.6). The domain *Agent* is predefined for Asynchronous Multi-Agent ASMs [BS03, p. 208] and contains all agents. We impart a domain *ActivityExecution*, a subset of *Agent*, together with some functions, which are local to each agent.

shared domain *ActivityExecution* ⊆ *Agent*

controlled *activity* : *ActivityExecution* → *Activity*
controlled *callers* : *ActivityExecution* → $\mathcal{P}$(*ActionExecution*)
controlled *terminated* : *ActivityExecution* → *Boolean*
controlled *context* : *ActivityExecution* → *BehavioredObject*

The function *activity* returns the related activity instance, this execution is for. In case the activity is marked as "isSingleExecution" [UML05, p. 307], multiple call behavior actions can call a single activity execution. Thus, a *set* of *callers* is needed to keep track of all these actions. The activity execution controller defined below executes until an "Activity Termination Event" has been processed, which sets *terminated* to true. Finally, *context* contains a link to the context object of the execution.

5.5.1 Events

This section describes the event queue, and the different event types.

5.5.1.1 Event queue

A separate event queue (ASM function *eventQueue*), is defined for each activity execution. This queue is **shared** [BS03], because other agents must have write-access to it. Action executions that are invoked by this activity execution, must e.g. inform the activity about their termination, and signals can be received from other activity executions.

shared *eventQueue* : *ActivityExecution* → $\mathcal{P}$(*ControllerEvent*)

The following macros are defined for event queues. Since there is concurrent access to the event queue from other agents, we use the ASM **add**- and **remove**-functions, to prevent inconsistent update sets [BS03]. EnqueueUniqueEvent is only defined for "Offer" and "Activity Termination" events (discussed below), to prevent multiple instances of these events in the queue. NextEvent selects an event to be processed by using a monitored function called *chooseEvent*. Thus, we leave the scheduling mechanism open, as it is specified in the UML specification [UML05].

HasEvents : *Boolean*
HasEvents $=_{def}$ *eventQueue*(*Self*) $\neq \emptyset$

EnqueueEvent : *ActivityExecution* × *ControllerEvent* → *Void*
EnqueueEvent(*exec*, *event*) ≡
    add *event* **to** *eventQueue*(*exec*)

EnqueueUniqueEvent : (*OfferEvent* ∪ *ActivityTerminationEvent*) → *Void*
EnqueueUniqueEvent(*event*) ≡
    if *event* ∉ *eventQueue*(*Self*) **then**
        EnqueueEvent(*Self*, *event*)

ReEnqueueEvent : *ControllerEvent* → *Void*
ReEnqueueEvent(*event*) ≡
    EnqueueEvent(*Self*, *event*)

monitored *chooseEvent* : *ControllerEvent*

NextEvent : *ControllerEvent*
NextEvent ≡
    local *scheduledEvent* := *chooseEvent*
    remove *scheduledEvent* **from** *eventQueue*(*Self*)
    result := *scheduledEvent*

5.5.1.2 Event types

The activity execution and action execution agents communicate by using events. We define the following event types and functions:

domain *ActivityStartEvent* $=_{def}$ {*input* : *ActivityParameterNode* → *DataToken**}
domain *ActivityTerminationEvent*
domain *OfferEvent*
domain *SignalReceivedEvent* $=_{def}$ {*signal* : *Object*}

domain *ActionEnableEvent* $=_{def}$ {*execution* : *ActionExecution*; *input* : *InputPin* $\rightarrow$ *DataToken**}
domain *ActionTerminationEvent* $=_{def}$ {*execution* : *ActionExecution*; *keepRunning* : *Boolean*;
output : *OutputPin* $\rightarrow$ *DataToken**}

domain *ControllerEvent* $\equiv$ *ActivityStartEvent* $\cup$ *ActivityTerminationEvent* $\cup$ *OfferEvent*
$\cup$*SignalReceivedEvent* $\cup$ *ActionEnableEvent* $\cup$ *ActionTerminationEvent*

ActivityStartEvent When an activity execution is created or an existing one is called (i.e. "isSingleExecution" is true), an "Activity Start" event is issued. When the event is processed, data tokens are put on incoming activity parameter nodes (as given by the *input* function) and control tokens are stored on outgoing edges of initial nodes. See Section 5.5.3.

ActivityTerminationEvent When an activity final node is reached, an "Activity Termination" event is generated. Outputs are written to the output pins of all callers and the activity execution controller is terminated. See Section 5.5.4.

OfferEvent If there are tokens on control flow or object flow sources, an "Offer" event is created. There is at most one offer event in the event queue. And with an offer event at hand, flow computation is initiated (see Section 5.7).

SignalReceivedEvent A "Signal Received" event marks the receipt of a signal instance, which is contained in *signal*. The signal was sent by send signal or broadcast signal actions included in the same or in other activities. See Section 5.6.5.4 on signals.

ActionEnableEvent An "Action Enable" event is generated if an action execution (function *execution*) is created. The subsequent enabling step removes data tokens from the input pins of the action and starts its execution (see Figure 5.4 a). The input data are contained in function *input*. Note that there can be multiple data tokens on each input pin for one action execution. See also Section 5.6.2.

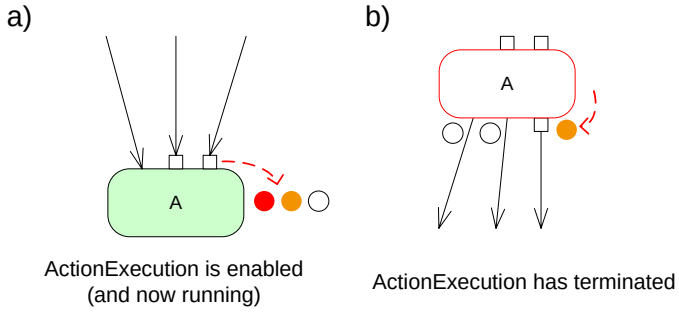

Figure 5.4: Enabling and termination of action executions

ActionTerminationEvent When an action execution finishes, it stores an "Action Termination" event in the event queue of the associated activity execution. Outputs of the terminated *execution* are stored in *output*, and control tokens are issued on the outgoing edges (see Figure 5.4 b). Accept event actions without incoming edges possibly have to be kept "running", as discussed in Section 5.6.5.3. This is indicated by *keepRunning*.

5.5.2 Controller loop

When a new activity execution has been started, it executes the ActivityExecutionController rule. The activity execution controller processes events from the associated event queue, until *terminated* is set to true by a handled "Activity Termination" event.

```
ActivityExecutionController ≡
    local event : ControllerEvent
    if ¬Self.terminated then
        if HasEvents then
            event := NextEvent (→ 5.5.1.1)
            seq
            case event of
                ActivityStartEvent :
                    HandleActivityStartEvent(event) (→ 5.5.3)
                ActivityTerminationEvent :
                    HandleActivityTerminationEvent (→ 5.5.4)
                ActionEnableEvent :
                    HandleActionEnableEvent(event) (→ 5.6.2)
                ActionTerminationEvent :
                    HandleActionTerminationEvent(event) (→ 5.6.3)
                SignalReceivedEvent :
                    HandleSignalReceivedEvent(event) (→ 5.6.5.4)
                OfferEvent :
                    HandleOfferEvent (→ 5.5.6)
```

The handling of these events is discussed in subsequent sections.

5.5.3 Start

The creation of new activity executions is handled by the StartNewActivityExecution rule. Two cases have to be differentiated:

1. "isSingleExecution" [UML05, p. 307] is false, i.e. each call of the rule creates a new, distinct execution.

2. If there is only a single execution, only the first invocation of the rule starts a new activity execution. All further calls result in reusing the existing execution and issuing new control and data tokens in it.

The else-part handles the first case. A new activity execution is created by using the ASM *new* keyword. The functions *activity* and *context* are initialized according to the parameters of the rule, and *terminated* is set to false. If there is a calling action execution (see call behavior action in Section 5.6.5.1), it is assigned to *callers*. By assigning the ActivityExecutionController-rule to the predefined *ASM*-function, the execution of the controller loop is started. Finally, an "Activity Start" event is enqueued in the event queue of the newly created activity execution. Handling of this event is discussed below.

If there is only a single execution, the existing (unique) execution is chosen by Choose- Execution. The function *callers* is updated with the new caller, and a new "Activity Start" event is issued for the existing execution. Note that, in case of only one execution, it must be guaranteed that further calls refer to the *same* context object. This is not reflected in our rules.

```
StartNewActivityExecution : Activity × BehavioredObject → ActivityExecution
StartNewActivityExecution(activity, context) ≡
    result := StartNewActivityExecution(activity, context, ∅, undefined)
```

```
StartNewActivityExecution : Activity × BehavioredObject
        ×P(ActivityParameterNode × DataToken*) × ActionExecution → ActivityExecution
StartNewActivityExecution(activity, context, input, callingExecution) ≡
    if activity.isSingleExecution ∧ IsRunning(activity) then
```

```
        let
            exec = ChooseExecution(activity)
        in
            if callingExecution ≠ undefined then
                exec.callers := exec.callers ∪ {callingExecution}
            EnqueueEvent(exec, ActivityStartEvent(input)) (→ 5.5.1.1)
            result := exec
    else
        let
            exec = new(ActivityExecution)
        in
            exec.activity := activity
            exec.terminated := false
            exec.context := context
            if callingExecution ≠ undefined then
                exec.callers := {callingExecution}
            else
                exec.callers := ∅
            ASM(exec) := ActivityExecutionController
            EnqueueEvent(exec, ActivityStartEvent(input)) (→ 5.5.1.1)
            result := exec
```

```
IsRunning : Activity → Boolean
IsRunning(activity) =def ∃exec ∈ ActivityExecution : exec.activity = activity ∧ ¬exec.terminated

ChooseExecution : Activity → ActivityExecution
ChooseExecution(activity) ≡
    choose exec with exec ∈ ActivityExecution ∧ exec.activity = activity ∧ ¬exec.terminated
        result := exec
```

The rule HandleActivityStartEvent is called by the activity execution controller 5.5.2, and performs the creation of the initial tokens. It first fills all input parameter nodes with data tokens given by the *input* function of the "Activity Start" event (see 5.5.1). According to the specification [UML05, p. 365]

> "An initial node is a starting point for executing an activity ... A control token is placed at the initial node when the activity starts, ... If an activity has more than one initial node, then invoking the activity starts multiple flows, one at each initial node."

We, therefore, create new control tokens for each outgoing edge of each initial node. The reason for putting control tokens on edges, rather than on nodes as described in the specification, is discussed in Section 4.3.2. Accept event actions without incoming edges which are not contained in any interruptible activity region also have to be activated. This, however is not evident when referring solely to the section on accept event actions in [UML05, p. 365]

> "If an AcceptEventAction has no incoming edges, then the action starts when the containing activity or structured node does, whichever most immediately contains the action."

If the section on interruptible activity regions is taken into account [UML05, p. 367], it is obvious, that we have to ignore accept event actions contained in those regions.

> "AcceptEventActions in the region that do not have incoming edges are enabled only when a token enters the region, even if the token is not directed at the accept event action."

Other kinds of actions without incoming edges are also started:

> "In addition, when an activity starts, control tokens are placed at actions and structured nodes that have no incoming edges" [UML05, p. 365]

The specification is missing that actions can also have input pins only. Data tokens can start these actions. Thus, we also have to make sure that the set of input pins is empty when implementing this rule.

HandleActivityStartEvent : *ActivityStartEvent* → *Void*
HandleActivityStartEvent(*event*) ≡
forall *parameterNode* **with** *parameterNode* ∈ (*ActivityInputParameters* ∩ *Self.activity.node*)
dataTokens(*parameterNode*) := *dataTokens*(*parameterNode*) ⊎ *parameterNode.event.input*
forall *n* **with** *n* ∈ (*InitialNode* ∩ *Self.activity.node*)
forall *e* **with** *e* ∈ *n.outgoing*
controlTokens(*e*) := *controlTokens*(*e*) ⊕ *new*(*ControlToken*)
forall *n* **with** *n* ∈ (*AcceptEventAction* ∩ *Self.activity.node*) ∧ *n.inInterruptibleRegion* = ∅
∧ *n.incoming* = ∅
CreateActionExecution(*n*, ∅) (→ 5.6.1)
forall *n* **with** *n* ∈ ((*Action* \ *AcceptEventAction*) ∩ *Self.activity.node*)
∧ *n.incoming* = ∅ ∧ *n.input* = ∅
CreateActionExecution(*n*, ∅) (→ 5.6.1)
seq
EnqueueUniqueEvent(*OfferEvent*) (→ 5.5.1.1)

At the end of the rule, an "Offer" event is issued, to indicate that tokens are available for possible transitions.

5.5.4 Termination

If an activity final node is reached, an "Activity Termination" event is enqueued. The event is handled by the following rule.

HandleActivityTerminationEvent ≡
forall *n* **with** *n* ∈ (*Action* ∩ *Self.activity.node*)
AbortAllActionExecutions(*n*, *Self*) (→ 5.6.4)
forall *caller* **with** *caller* ∈ *Self.callers*
forall *n* **with** *n* ∈ (*ActivityOutputParameters* ∩ *Self.activity.node*)
caller.callResult(*n*) := *dataTokens*(*n*)
dataTokens(*n*) := [ ]
seq
Self.terminated := *true*

All running actions are aborted. Data tokens contained in output activity parameter nodes are saved in *callResult* for each caller of the activity. If "isSingleExecution" is false, which is the normal case, there is only one caller. The call result is used by the call behavior action to provide the data tokens for the action termination (and, thus, for the output pins of the call action). Finally, *terminated* is set to true, resulting in termination of the current activity execution controller.

5.5.5 Abort

When call behavior actions are aborted, the corresponding activity executions (which were created by these call actions) must also be terminated. This is achieved by the following rule. Except for setting the call results, it works like the HandleActivityTerminationEvent rule.

AbortActivityExecution : *ActivityExecution* → *Void*
AbortActivityExecution(*exec*) ≡
    forall n **with** $n \in (Action \cap Self.activity.node)$
        AbortAllActionExecutions(n, *exec*) (→ 5.6.4)
    seq
    $exec.terminated := true$

5.5.6 Transitions

In this section we present our main ASM rule for the computation and execution of transitions, describing the structure of the token flow semantics.

According to the UML specification [UML05], nodes offer tokens on outgoing edges. The exact working of propagation of token offers and their selection at destination actions, final and object nodes, however, is neither formally defined, nor adequately discussed in the specification. Our proposal for transition computation and execution consists of the following main ASM rule that is executed repeatedly as long as offer events are generated, i.e. control or data tokens are available for the current activity execution:

HandleOfferEvent ≡
    ComputeTokenOffers (→ 5.7.3)
    seq
    FetchBufferedTokens (→ 5.7.8)
    seq
    SelectTokenOffers (→ 5.7.5)
    seq
    RemoveFlowsInInterruptedRegions (→ 5.7.6)
    seq
    ActivateAcceptEventActions (→ 5.7.7)
    seq
    BufferTokens (→ 5.7.8)
    seq
    ExecuteTransition (→ 5.8)
    seq
    if $\exists n \in FlowSource : n \in Self.activity.node \ \wedge \ HasToken(n)$ **then**
        EnqueueUniqueEvent(*OfferEvent*) (→ 5.5.1.1)

The ComputeTokenOffers macro computes the spreading of token offers through the activity graph. After all possible offers have been computed, previously buffered offers are fetched by FetchBufferedTokens. Then, subsets of all offers are selected by SelectTokenOffers at destination actions, object and final nodes, preparing the traversal of the associated tokens. Note that selecting token offers can invalidate other, conflicting token offers. Since aborting interruptible activity regions can prevent tokens from traversal, the rule RemoveFlowsInInterruptedRegions removes those selections. Accept event actions without incoming edges contained in interruptible activity regions are initialized by ActivateAcceptEventActions. Finally, BufferTokens stores offers, which have to be delayed due to the buffering semantics of fork nodes. All these rules are described in detail in Section 5.7.

After all relevant offers have been selected, ExecuteTransition executes the token traversal and performs the termination of actions and removal of tokens in interrupted regions, as discussed in Section 5.8.

Finally, since ExecuteTransition selects transitions nondeterministically, and false guards and missing tokens for join synchronizations can prevent other tokens from flowing, a new offer event is generated if there are still tokens available.

5.6 Action

This section describes the life cycle of action executions. The first subsections deal with creation, enabling, termination and aborting of action executions. Section 5.6.5 presents the execution rules for call behavior, call operation, accept event, send signal and broadcast signal actions.

Similar to activity executions, each action execution is modeled as a separate ASM agent. We introduce a domain *ActionExecution* which is a subset of the predefined domain *Agent*.

shared domain *ActionExecution* $\subseteq$ *Agent*

A set of ASM functions is defined for all types of action executions. The function *node* refers to the action this execution was created for. Input data, which is taken from the input pins of the action, is provided by *input*. To obtain the activity execution, where this action execution runs in, *activityExecution* is used.

controlled *node* : *ActionExecution* $\rightarrow$ *Action*
controlled *input* : *ActionExecution* $\rightarrow$ (*InputPin* $\rightarrow$ *DataToken**)
controlled *activityExecution* : *ActionExecution* $\rightarrow$ *ActivityExecution*

The steps of executing an action are detailed in [UML05, p. 302]:

1. "An action execution is created when all its object flow and control flow prerequisites have been satisfied (implicit join). ... The flow prerequisite is satisfied when all of the input pins are offered tokens and accept them all at once, precluding them from being consumed by any other actions. This ensures that multiple action executions competing for tokens do not accept only some of the tokens they need to begin, causing deadlock as each execution waits for tokens that are already taken by others.
2. An action execution consumes the input control and object tokens and removes them from the sources of control edges and from input pins. The action execution is now enabled and may begin execution. If multiple control tokens are available on a single edge, they are all consumed.
3. An action continues executing until it has completed. ... The detailed semantic of execution an action and definition of completion depends on the particular subclass of action.
4. When completed, an action execution offers object tokens on all its output pins and control tokens on all its outgoing control edges (implicit fork), and it terminates. ... The output tokens are now available to satisfy the control or object flow prerequisites for other action executions."

Item 1 is assured by our flow computation and transition execution algorithm presented in Sections 5.7 and 5.8. When a transition is executed, the macro CreateActionExecution, which will be discussed later in this section, is called.

After an action execution has been created, an "Action Enable" event is stored in the event queue (see item 2). HandleActionEnableEvent removes the data tokens from the input pins and enables the action to start executing. The ASM rules for execution (item 3) of the supported types of actions are discussed in Section 5.6.5. In contrast to the specification, we already remove all control tokens from source edges, when creating an action execution. The reason for this deviation is discussed in Section 4.3.7.

When the action has completed (item 4), an "Action Termination" event is enqueued. HandleActionTerminationEvent stores the results in the output pins and puts new control tokens on the outgoing edges of the terminated action. This macro is described in Section 5.6.3.

To reflect these (and additional) states of an action execution, the function *mode* is used:

controlled *mode* : *ActionExecution* $\rightarrow$ *ActionExecutionMode*

static domain *ActionExecutionMode*
initially *ActionExecutionMode* = {created, enabled, running, completed, terminating, terminated, abortRequested}

created When an action execution is created by using CreateActionExecution 5.6.1, its mode is set to "created". It does not begin execution until it is enabled. Thus, each action execution in this mode only executes **skip** (see Section 5.6.5).

enabled After processing of an "Action Enable" event (see HandleActionEnableEvent in Section 5.6.2), the execution mode is set to "enabled". Some initialization tasks are performed for some types of action and execution mode is set to "running".

running The actual task of the action is performed in this mode. After an action has finished executing, its mode is set to "completed".

completed In most cases, the results are prepared, and an "Action Termination" event is issued in this mode. The action execution goes into "terminating" state afterwards.

terminating When terminating, the ASM execution rule is no more applicable and therefore the corresponding agent is terminated. The execution is, however, still considered not "terminated" as long as the "Action Termination" event has not been handled. After handling this event, mode is set to "terminated".

terminated The execution has terminated and outputs have been written to the output pins.

abortRequested When leaving interruptible activity regions or terminating activity executions, all contained action executions have to be aborted. In this case, AbortActionExecution (see Section 5.6.4) is called which sets the execution state to "abortRequested". If this is observed by the execution rule, its agent is terminated immediately, by going into "terminated" mode. Note that in this case, no "Action Termination" event is issued.

5.6.1 Creation

When transitions are executed, control tokens are removed from source edges and data tokens are moved from source object nodes to targets. If the target of a transition is an action, a new execution for this action is created [UML05, p. 302]. Our tool (see chapter 6) illustrates this by highlighting the action border and traversed edges, and by moving control and data tokens to the destination. An example is shown in Figure 5.5.

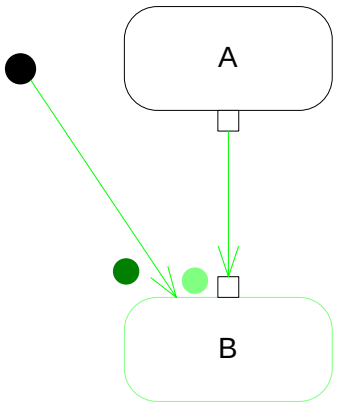

Figure 5.5: Creation of an action execution for action B

The following rule creates new executions for actions. Parameters are the type of action and input data which are provided through its input pins. A new ASM agent is created by using the *new* keyword, and its initial *mode* is set to "created".

CreateActionExecution : *Action* × (*InputPin* → *DataToken**) → *Void*
CreateActionExecution(n, *input*) ≡
let
execution = *new*(*ActionExecution*)
in
execution.node := n
execution.input := *input*
execution.activityExecution := *Self*
execution.mode := created
if $n \in$ *CallBehaviorAction* **then**
ASM(*execution*) := ExecuteCallBehaviorActionExecution (→ 5.6.5.1)
if $n \in$ *CallOperationAction* **then**
ASM(*execution*) := ExecuteCallOperationActionExecution (→ 5.6.5.2)
if $n \in$ *AcceptEventAction* **then**
ASM(*execution*) := ExecuteAcceptEventActionExecution (→ 5.6.5.3)
if $n \in$ *SendSignalAction* **then**
ASM(*execution*) := ExecuteSendSignalActionExecution (→ 5.6.5.4)
if $n \in$ *BroadcastSignalAction* **then**
ASM(*execution*) := ExecuteBroadcastSignalActionExecution (→ 5.6.5.4)
EnqueueEvent(*Self*, *ActionEnableEvent*(*execution*, *input*)) (→ 5.5.1.1)

Depending on the concrete type of action, a corresponding rule is assigned to the *ASM* function, leading to the start of the agent. Additionally, an "Action Enable" event is enqueued. Handling this event is discussed next.

5.6.2 Enabling

"Action Enable" events are handled by the following rule. As described at the beginning of Section 5.6, the specification states that all data tokens have to be removed from the input pins of the action:

HandleActionEnableEvent : *ActionEnableEvent* → *Void*
HandleActionEnableEvent(ev) ≡
forall *inputPin* **with** *inputPin* ∈ *ev.execution.node.input*
dataTokens(*inputPin*) := *dataTokens*(*inputPin*) ⊘ *ev.input*(*inputPin*)
ev.execution.mode := enabled

Control tokens have already been removed from the source edges, as discussed in Section 4.3.7. The execution is enabled by setting its *mode* accordingly.

Figure 5.6 shows how our tool (see chapter 6) illustrates the enabling of an action execution. The Action is highlighted and a token is put alongside.

5.6.3 Termination

When an action execution terminates, control tokens are offered on the outgoing edges and data tokens for the results are stored in the output pins. An example is illustrated in Figure 5.7. Our tool (see chapter 6) highlights terminated actions with a red border.

Since we allow *lower* and *upper* attributes for pins (pin is a multiplicity element [UML05, p. 90]), we must put the following constraint on action terminations:

constraint
$\forall ev \in$ *ActionTerminationEvent* : $\forall$ *outputPin* ∈ *ev.execution.node.output* :
lower(*outputPin*) ≤ |*ev.output*(*outputPin*)| ≤ *upper*(*outputPin*)

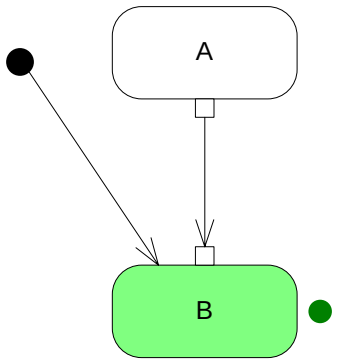

Figure 5.6: Enabling of an action execution

We, therefore, assume that each action produces "enough" results to be able to terminate. The actual "Action Termination" event is processed as follows. If there is not enough space left in the output pins, the event processing has to be deferred by calling ReEnqueueEvent. If there is space available, control and data tokens are stored. If there is at least one outgoing edge or output pin, an "Offer" event is generated, indicating possible transitions. Accept event actions without incoming edges must eventually be kept running. Thus, in this case, *mode* is not set to "terminated".

HandleActionTerminationEvent : *ActionTerminationEvent* → *Void*
HandleActionTerminationEvent(*ev*) ≡
    if ∀*pin* ∈ *ev.execution.node.output* : (|*dataTokens*(*pin*)| + |*ev.output*(*pin*)|)
            ≤ *upperBound*(*pin*) **then**
        forall *e* **with** *e* ∈ *ev.execution.node.outgoing*
            controlTokens(*e*) := *controlTokens*(*e*) ⊕ *new*(*ControlToken*)
        forall *outputPin* **with** *outputPin* ∈ *ev.execution.node.output*
            dataTokens(*outputPin*) := *dataTokens*(*outputPin*) ⊎ *ev.output*(*outputPin*)
        if |*ev.execution.node.outgoing*| > 0 ∨ |*ev.execution.node.output*| > 0 **then**
            EnqueueUniqueEvent(*OfferEvent*) (→ 5.5.1.1)
        if ¬*ev.keepRunning* **then**
            ev.execution.mode := terminated
    else
        ReEnqueueEvent(*ev*) (→ 5.5.1.1)

5.6.4 Abort

If an interruptible activity region or an activity execution is terminated, all contained actions must also be aborted. This is implemented by the following rules.

AbortAllActionExecutions : *Action* × *ActivityExecution* → *Void*
AbortAllActionExecutions(*n*, *activityExecution*) ≡
    forall *exec* **with** *exec* ∈ *ActionExecution* ∧ *exec.mode* ∉ {terminating, terminated}
            ∧ *exec.node* = *n* ∧ *exec.activityExecution* = *activityExecution*
        AbortActionExecution(*exec*)

AbortActionExecution : *ActionExecution* → *Void*
AbortActionExecution(*exec*) ≡
    if *exec.mode* ≠ abortRequested **then**
        exec.mode := abortRequested

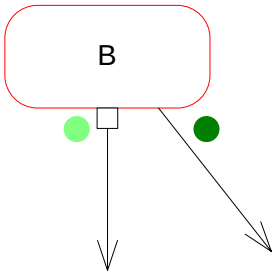

Figure 5.7: Termination of an action execution

else
  if *exec.mode* $\neq$ terminated **then**
    skip

To abort an action execution, its *mode* is set to "abortRequested". The "Abort" rule then waits for the execution to terminate, which is indicated by its mode set to "terminated".

5.6.5 Execution

Each type of action fulfills a specific task. This section describes the main rule for each "action"-agent.

5.6.5.1 CallBehaviorAction

The semantics of a call behavior action is as follows [UML05, p, 238]:

1. "When all the prerequisites of the action execution are satisfied, CallBehaviorAction invokes its specified behavior with the values on the input pins as arguments. When the behavior is finished, the output values are put on the output pins. Each parameter of the behavior of the action provides output to a pin or takes input from one. ...
2. If the call is asynchronous, the action completes immediately. ...
3. An asynchronous invocation completes when its behavior is started, or is at least ensured to be started at some point. Any return or out values from the invoked behavior are not passed back to the containing behavior. When an asynchronous invocation is done, the containing behavior continues regardless of the status of the invoked behavior. For example, the containing behavior may complete even though the invoked behavior is not finished.
4. If the call is synchronous, execution of the calling action is blocked until it receives a reply from the invoked behavior. The reply includes values for any return, out, or inout parameters.
5. If the call is synchronous, when the execution of the invoked behavior completes, the result values are placed on the result pins of the call behavior action, and the execution of the action is complete ..."

As stated earlier, we only support activities as valid behaviors. Other possibilities would include UML interactions and state machines.

Creating an execution

The invocation does not start until the execution is in "running" *mode*. It creates a new activity execution by using the StartNewActivityExecution, which was described in Section 5.5.3. Inputs are provided by the input pins of the call action. Before calling the macro, the inputs must

be mapped to the corresponding input activity parameter nodes of the called activity. This is implemented by PinToParameter. Mapping is done by matching the index of the input pin and parameter, as specified in [UML05, p. 238]:

> "The type, ordering, and multiplicity of an argument or result pin is derived from the corresponding parameter of the behavior."

Context object

As discussed in Section 4.4.1, it is not clear which object should be supplied as the "context object" for a new activity execution of a call behavior action. We leave this option open, by using a "context" tag, introduced in Section 4.4.1. If this tag is not defined for a call behavior action, the context object of the current activity execution is used as the context object for the called activity. If the tag is provided, the following function is used to compute the context object:

monitored *computeContext* : *BehavioredObject* × *String* → *BehavioredObject*

It takes the current context object and a string as input and computes the object to use as the context. We leave the meaning of the string parameter, and how computation is actually performed, open at this point. In Section 4.4.1 we propose to use path expressions, loosely based on XPath [Kay04], to navigate the current object graph. Context objects are then determined by querying associations between objects. This is also implemented by our tool described in chapter 6.

isSynchronous

If in "running" mode and if the call is synchronous, the action execution waits until the called activity execution has terminated. If asynchronous, the action execution immediately goes into "completed" state. We deviate from the specification (see item 4 above), because we do not model explicit "reply" messages from objects. This is because we do not want to provide a formalization for all aspects of the UML 2, including event handling and messaging, as specified by the "Common Behaviors" packages [UML05].

If an abort is requested (by AbortActionExecution 5.6.4), the invoked activity execution is terminated. If mode is "completed", the output activity parameter nodes are mapped to the output pins of the calling action, by using ParameterToPin. Finally, an "Activity Termination" event is issued.

controlled *calledExecution* : *ActionExecution* → *ActivityExecution*
controlled *callResult* : *ActionExecution* → (*ActivityParameterNode* → *DataToken**)

ExecuteCallBehaviorActionExecution ≡
  local *result* : *OutputPin* → *DataToken**
  local *context* : *BehavioredObject*
  if *Self*.*mode* = created **then**
    skip
  if *Self*.*mode* = enabled **then**
    if *tagValue*(*Self*.*node*, CallContext, context) = *undefined* **then**
      context := *Self*.*activityExecution*.*context*
    else
      context := *computeContext*(*Self*.*activityExecution*.*context*,
                    tagValue(*Self*.*node*, CallContext, context))
    seq
    Self.*calledExecution* := StartNewActivityExecution(*Self*.*node*.*behavior*,
        context, {(PinToParameter(*n*), *ts*) | (*n*, *ts*) ∈ *Self*.*input*}, *Self*) (→ 5.5.3)
    Self.*mode* := running
  if *Self*.*mode* = running **then**
    if *Self*.*node*.*isSynchronous* **then**
      if ¬*Self*.*calledExecution*.*terminated* **then**
        skip
      else
        Self.*mode* := completed
    else

```
        Self.mode := completed
    if Self.mode = abortRequested then
        AbortActivityExecution(Self.calledExecution) (→ 5.5.5)
        Self.mode := terminated
    if Self.mode = completed then
        forall n with n ∈ (ActivityOutputParameters ∩ Self.node.behavior.node)
            result(ParameterToPin(n, Self.node)) := Self.callResult(n)
        seq
        EnqueueEvent(Self.activityExecution,
                     ActionTerminationEvent(Self, false, result)) (→ 5.5.1.1)
        Self.mode := terminating


ParameterToPin : ActivityParameterNode × CallBehaviorAction → OutputPin
ParameterToPin(n, callAction) ≡
    let
        outParameters = [p ∈ callAction.behavior.ownedParameters |
                         p.direction ∈ {inout, out, return}]
    in
        choose outputPin with outputPin ∈ callAction.result ∧ indexOf(callAction.result,
                              outputPin) = indexOf(outParameters, n.parameter)
            result := outputPin


PinToParameter : InputPin → ActivityParameterNode
PinToParameter(inputPin) ≡
    choose callAction with callAction ∈ CallBehaviorAction
                           ∧ inputPin ∈ callAction.argument
        let
            inParameters = [p ∈ callAction.behavior.ownedParameters |
                            p.direction ∈ {in, inout}]
        in
            choose n with n ∈ ActivityParameterNode ∧ indexOf(callAction.argument,
                          inputPin) = indexOf(inParameters, n.parameter)
                result := n
```

5.6.5.2 CallOperationAction

Call operation actions are used for method invocations [UML05, p. 239]:

> "The inputs to the action determine the target object and additional actual arguments of the call.
>
> 1. When all the prerequisites of the action execution are satisfied, information comprising the operation and the argument pin values of the action execution is created and transmitted to the target object. The target objects may be local or remote. The manner of transmitting the call, the amount of time required to transmit it, the order in which the transmissions reach the various target objects, and the path for reaching the target objects are undefined.
> 2. When a call arrives at a target object, it may invoke a behavior in the target object. ...
> 3. If the call is synchronous, when the execution of the invoked behavior completes, its return results are transmitted back as a reply to the calling action execution. ...

4. If the call is asynchronous, the caller proceeds immediately and the execution of the call operation action is complete. Any return or out values from the invoked operation are not passed back to the containing behavior. If the call is synchronous, the caller is blocked from further execution until it receives a reply from the invoked behavior.
5. When the reply transmission arrives at the invoking action execution, the return result values are placed on the result pins of the call operation action, and the execution of the action is complete."

Since we do not provide a semantics for messages and invocations, as specified by the "Common Behaviors" package of the UML, we do not care about other call mechanisms but direct invocations of the code. We use the following monitored function to locate a method. A context (i.e., target) object and the operation to call are provided. An instance of "Code", which is not further specified here, is returned:

monitored *locateMethod* : *Object* × *Operation* → *Code*

Besides, we define two macros, that execute code synchronously resp. asynchronously. ExecCode executes one "step" (e.g. one statement) and then returns. A program counter and additional context information is handled internally. If the code terminates, its outputs are returned. ExecCodeAsync starts execution of the code and returns immediately, without yielding any output. The function *terminated* indicates whether execution of the code has finished. In [BFGS05] and [SB04], an ASM semantics for C#, in terms of an abstract interpreter, is given, that could be used in our rule. In our tool implementation (see chapter 6), user code is invoked by creating a new thread. If the call action is synchronous, it waits for the thread to terminate.

ExecCode : *Code* → (*OutputPin* → *DataToken**)
ExecCodeAsync : *Code* → *Void*

monitored *terminated* : *Code* → *Boolean*

The following rule is executed by the agent for a call operation action. The context object to use is computed the same way, as it is in the execution rule for call behavior actions (see 5.6.5.1). When completed, an "Action Termination" event is issued, which includes the result of the (synchronous) call.

```
ExecuteCallOperationActionExecution ≡
    local code : Code
    local output : OutputPin → DataToken*
    if Self.mode = created then
        skip
    if Self.mode = enabled then
        if tagValue(Self.node, CallContext, context) = undefined then
            code := locateMethod(Self.node.target, Self.node.operation)
        else
            code := locateMethod(computeContext(Self.activityExecution.context,
                                                tagValue(Self.node, CallContext, context)),
                                 Self.node.operation)
        Self.mode := running
    if Self.mode = running then
        if Self.node.isSynchronous then
            if ¬code.terminated then
                output := ExecCode(code)
            else
                Self.mode := completed
        else
```

```
        ExecCodeAsync(code)
        Self.mode := completed
    if Self.mode = abortRequested then
        Self.mode := terminated
    if Self.mode = completed then
        EnqueueEvent(Self.activityExecution,
                     ActionTerminationEvent(Self, false, output)) (→ 5.5.1.1)
        Self.mode := terminating
```

5.6.5.3 AcceptEventAction

Accept event actions wait for the occurrence of an event meeting a specified condition [UML05, p. 229]:

> "Accept event actions handle event occurrences detected by the object owning the behavior ... Event occurrences are detected by objects independently of actions and the occurrences are stored by the object. The arrangement of detected event occurrences is not defined, but it is expected that extensions or profiles will specify such arrangements."

Since we do not provide a semantics for UML messaging mechanisms (see Section 5.9), we define event types that are only needed in our context and also store those events at activity executions instead of context objects mentioned above. We, however, also leave the arrangement of detected event occurrences open by using the monitored function *chooseEvent* defined in Section 5.5.1.1.

Regarding the behavior of accept event actions, [UML05, p. 229] also states, that

> "If the accept event action is executed and the object detected an event occurrence matching one of the triggers on the action and the occurrence has not been accepted by another action or otherwise consumed by another behavior, then the accept event action completes and outputs a value describing the occurrence. If such a matching occurrence is not available, the action waits until such an occurrence becomes available, at which point the action may accept it. In a system with concurrency, several actions or other behaviors might contend for an available event occurrence. Unless otherwise specified by an extension or profile, only one action accepts a given occurrence, even if the occurrence would satisfy multiple concurrently executing actions. If the occurrence is a signal event occurrence and unmarshall is false, the result value contains a signal object whose reception by the owning object caused the occurrence. ... If the occurrence is a time event occurrence, the result value contains the time at which the occurrence transpired. ..."

We only support time and signal events as valid triggers. Although the UML meta model allows for multiple triggers to be specified, we use only one trigger. As discussed in Section 5.9, this is useful, because it is impossible to model multiple triggers graphically for one accept event action.

The execution rule for accept event actions first determines the type of event, and calls subrules accordingly:

```
ExecuteAcceptEventActionExecution ≡
    let
        event = Self.node.trigger.event
    in
        if event ∈ TimeEvent then
            ExecuteWaitTimeActionExecution(event)
        else
            ExecuteAcceptSignalActionExecution(event)
```

For *time events* we introduce a domain for the current time:

monitored *now* : *Time*

The execution rule is as follows. When enabled, *now* is used to compute the time to elapse, which is stored in the function *endTime*. The computation depends on the meta model function "*isRelative*", which indicates whether the event is relative or absolute. When running, **skip** is executed until the timer has elapsed. The "time at which the occurrence transpired" is stored in the output pin of the accept event action, as required by the UML specification, and an "Action Termination" event is issued.

controlled *endTime* : *ActionExecution* → *Time*

```
ExecuteWaitTimeActionExecution(event) ≡
    local result : OutputPin → DataToken*
    if Self.mode = created then
        skip
    if Self.mode = enabled then
        if event.isRelative then
            Self.endTime := now + event.when
        else
            Self.endTime := event.when
        Self.mode := running
    if Self.mode = running then
        if now < Self.endTime then
            skip
        else
            forall outputPin with outputPin ∈ Self.node.output
                result(outputPin) := [new(DataToken(now))]
            seq
            EnqueueEvent(Self.activityExecution, ActionTerminationEvent(Self,
                                        MustStayActive, result)) (→ 5.5.1.1)
            if MustStayActive ∧ event.isRelative then
                Self.endTime := now + event.when
            else
                Self.mode := completed
    if Self.mode = completed then
        Self.mode := terminating
    if Self.mode = abortRequested then
        Self.mode := terminated
```

Accept event actions eventually must stay active after activation [UML05, p. 299]

> "... an AcceptEventAction with no incoming edges remains enabled after it accepts an event. It does not terminate after accepting an event and outputting a value, but continues to wait for other events."

It is not immediately clear from the specification, when the action has to terminate at all (see discussion in Section 4.3.4). There, it is also stated that "An AcceptEventAction with no incoming edges and contained by a structured node is terminated when its container is terminated.". Unfortunately, this statement cannot be applied to interruptible activity regions, because they are not a UML "structured node". It is, however, reasonable to use the same rule for interruptible regions. We, therefore, define the macro MustStayActive. If an accept event action is not contained in any region, it remains active until termination of the activity execution. If it is contained in a region, it checks whether the region or any parent regions are active. Parent regions are only tested if the UML tag "OnParentActive" is set. This is a semantic variation point introduced by us, as described in Section 4.4.4.

$MustStayActive : Boolean$
$MustStayActive =_{def}$
$Self.node.incoming = \emptyset \ \wedge \ (Self.node.inInterruptibleRegion = \emptyset \vee$
$(\exists r \in InterruptibleActivityRegion : Self.node \in r.containedNode \ \wedge \ (IsActive(r)$
$\vee \ (\exists r' \in r.parents : IsActive(r')$
$\wedge \forall r'' \in regionsFromTo(r, r') :$
$tagValue(r'', \mathsf{InterruptibleActivityRegionHandling}, \mathsf{regionReactivationPolicy})$
$= \ \mathsf{OnParentActive}))))$

A region is "active", if it contains tokens or active actions. An action, in turn, is "active" if it has not been terminated or aborted yet.

$IsActive : InterruptibleActivityRegion \rightarrow Boolean$
$IsActive(r) =_{def} \exists n \in innerNodes(r) : (n \in FlowSource \ \wedge \ HasToken(n))$
$\vee \ (n \in Action \ \wedge \ IsActive(n))$

$IsActive : Action \rightarrow Boolean$
$IsActive(n) =_{def} \exists exec \in ActionExecution : exec.node = n \ \wedge \ exec.activityExecution = Self$
$\wedge \ exec.mode \in \ \{\mathsf{created}, \mathsf{enabled}, \mathsf{running}, \mathsf{completed}, \mathsf{terminating}\}$

Note that, when dealing with time events, MustStayActive is only useful if the time event is relative, because absolute event expire only once. This is not indicated in the UML specification.

The other event type we support, are *signal events*. The execution rule behaves much the same as ExecuteWaitTimeActionExecution, but waits for a signal instance to be "received". This is achieved by setting the function *receivedSignal*, when processing received signals in HandleSignalReceivedEvent (see Section 5.6.5.4). The function is **shared** to allow the activity execution agent to access it.

shared $receivedSignal : ActionExecution \rightarrow Object$

The following rule handles accept event actions for signal events. ASM **skip** is executed until *receivedSignal* is set. An "Action Termination" event is issued, and the resulting signal instance is prepared for the output pin.

ExecuteAcceptSignalActionExecution($event$) $\equiv$
local $result : OutputPin \rightarrow DataToken^*$
if $Self.mode =$ created **then**
skip
if $Self.mode =$ enabled **then**
$Self.receivedSignal := undefined$
$Self.mode :=$ running
if $Self.mode =$ running **then**
if $Self.receivedSignal = undefined$ **then**
skip
else
forall $outputPin$ **with** $outputPin \in Self.node.output$
$result(outputPin) := [new(DataToken(Self.receivedSignal))]$
seq
EnqueueEvent($Self.activityExecution$, $ActionTerminationEvent(Self,$
$MustStayActive, result)$) ($\rightarrow$ 5.5.1.1)
if $MustStayActive$ **then**
$Self.receivedSignal := undefined$
else
$Self.mode :=$ completed
if $Self.mode =$ completed **then**
$Self.mode :=$ terminating

if *Self*.*mode* = abortRequested **then**
  Self.*mode* := terminated

5.6.5.4 SendSignalAction and BroadcastSignalAction

This section deals with send signal and broadcast signal actions, which send signal instances to objects. The specification writes [UML05, p. 274]

1. "When all the prerequisites of the action execution are satisfied, a signal instance of the type specified by signal is generated from the argument values and his signal instance is transmitted to the identified target object. ...
2. When a transmission arrives at a target object, it may invoke behavior in the target object. ...
3. A send signal action receives no reply from the invoked behavior; any attempted reply is simply ignored, and no transmission is performed to the requestor."

Thus, the following ASM rule for send signal action execution, creates a signal instance and uses the SendSignal macro discussed below to send it to the specified *target*:

ExecuteSendSignalActionExecution ≡
  if *Self*.*mode* = created **then**
    skip
  if *Self*.*mode* = enabled **then**
    Self.*mode* := running
  if *Self*.*mode* = running **then**
    let
      signal = *createInstance*(*Self*.*node*.*signal*, [*head*(*Self*.*input*(*pin*)).*value* |
                  pin ∈ *Self*.*node*.*argument*])
      target = *head*(*Self*.*input*(*Self*.*node*.*target*)).*value*
    in
      SendSignal(*signal*, *target*)
      Self.*mode* := completed
  if *Self*.*mode* = completed **then**
    EnqueueEvent(*Self*.*activityExecution*,
                 ActionTerminationEvent(*Self*, *false*, ∅)) (→ 5.5.1.1)
    Self.*mode* := terminating
  if *Self*.*mode* = abortRequested **then**
    Self.*mode* := terminated

Creation of a signal instance is performed by the following monitored function *createInstance*. The above rule uses data from the input pins to set the attributes of the signal instance, as required by the specification.

monitored *createInstance* : *Signal* × *Object** → *Object*

The macro SendSignal is defined as follows. It takes a signal object and an activity execution, which is the target of the send action, as parameters. If the activity cannot handle the signal, i.e. there are no accept event actions for signal events of the type of signal that is sent, it is discarded. The function *CanHandleSignal* checks this prerequisite. Note, that accept event actions also accept signals of any subtype of the specified signal type, as stated by [UML05, p. 228]

> "For triggers with signal events, a signal of the specified type or any subtype of the specified signal type is accepted."

We, therefore, compare the signal types by using the *classOf* function, that was defined in Section 5.2.

If the signal can be handled, SendSignal uses the UML tag configuration to determine whether buffering is to be used (see Section 4.2). If buffering is not wanted, HandleSignal (described below) is called, which tries to handle the signal immediately. If there is no active accept event action, HandleSignal discards the signal. If buffering is enabled, the "replace" tag is examined, denoting whether the arrived signal replaces old signals in the event queue. If signals are to be replaced (see Section 4.2), all matching previous signal objects are deleted. Finally, a new "Signal Received" event is created for the signal, and the event is stored in the event queue of the target activity execution.

Due to the reasons described in Section 4.2, we also introduce a "distribution" tag, which indicates if a signal object is to be distributed to nested activity executions. This is implemented by the last part of the SendSignal rule, which verifies this tag and sends the signal to all activity executions, that are called by the current one, by calling SendSignal recursively.

SendSignal : *Object* × *ActivityExecution* → *Void*
SendSignal(*signal*, *exec*) ≡
  if *CanHandleSignal*(*signal*, *exec.activity*) **then**
    if *tagValue*(*exec.activity*, SignalBufferingAndReplacement, buffer) = No
      ∨ (∃*n* ∈ *handlingActions*(*signal*, *exec.activity*) :
        tagValue(*n*, SignalBufferingAndReplacement, buffer) = No) **then**
      HandleSignal(*signal*, *exec*)
    else
      if (*tagValue*(*exec.activity*, SignalBufferingAndReplacement, replace) = Yes
          ∧ (∄*n* ∈ *handlingActions*(*signal*, *exec.activity*) :
            tagValue(*n*, SignalBufferingAndReplacement, replace) = No))
          ∨ (*tagValue*(*exec.activity*, SignalBufferingAndReplacement, replace) = No
          ∧ (∃*n* ∈ *handlingActions*(*signal*, *exec.activity*) :
            tagValue(*n*, SignalBufferingAndReplacement, replace) = Yes)) **then**
        remove {*e* ∈ *eventQueue*(*exec*) | *e* ∈ *SignalReceivedEvent*
          ∧ (*classOf*(*signal*) ⪯ *classOf*(*e.signal*)
            ∨ *classOf*(*e.signal*) ⪯ *classOf*(*signal*))} **from** *eventQueue*(*exec*)
      seq
      EnqueueEvent(*exec*, *SignalReceivedEvent*(*signal*)) (→ 5.5.1.1)
  forall *subExec* **with** *subExec* ∈ {*e.calledExecution* | *e* ∈ *ActionExecution* ∧
      e.node ∈ (*CallBehaviorAction* ∩ *exec.activity.node*)
      ∧ (*tagValue*(*exec.activity*, SignalDistribution, distribute) = Yes
        ∨ *tagValue*(*e.node*, SignalDistribution, distribute) = Yes)}
    SendSignal(*signal*, *subExec*)

CanHandleSignal : *Object* × *Activity* → *Boolean*
CanHandleSignal(*signal*, *activity*) =$_{def}$ *handlingActions*(*signal*, *activity*) ≠ ∅

handlingActions : *Object* × *Activity* → $\mathcal{P}$(*AcceptEventAction*)
handlingActions(*signal*, *activity*) =$_{def}$ {*n* ∈ (*AcceptEventAction* ∩ *activity.node*) |
  n.trigger.event ∈ *SignalEvent* ∧ *classOf*(*signal*) ⪯ *n.trigger.event.signal*}

Handling of a signal is implemented by HandleSignal. The rule chooses an action execution for a suitable accept event action and sets its function *receivedSignal* to the provided signal instance. Note that, by using "choose", only one execution is chosen nondeterministically, as required by the specification. If no such execution exists, the signal is discarded. The rule HandleSignalReceivedEvent processes received signals (i.e., "Signal Receive" events), by simply calling HandleSignal.

HandleSignal : *Object* × *ActivityExecution* → *Void*

```
HandleSignal(signal, activityExecution) ≡
    choose exec with exec ∈ ActionExecution
            ∧ exec.node ∈ handlingActions(signal, activityExecution.activity)
            ∧ exec.activityExecution = activityExecution
            ∧ exec.mode ∈ {created, enabled, running}
            ∧ exec.receivedSignal = undefined
        exec.receivedSignal := signal


HandleSignalReceivedEvent : SignalReceivedEvent → Void
HandleSignalReceivedEvent(e) ≡
    HandleSignal(e.signal, Self)
```

Another send action we support, is *BroadcastSignalAction*. According to [UML05, p. 236]

> "When all the prerequisites of the action execution are satisfied, a signal object is generated from the argument values according to signal and this signal object is transmitted concurrently to each of the implicit broadcast target objects in the system. The manner of identifying the set of objects that are broadcast targets is a semantic variation point and may be limited to some subset of all the objects that exist. There is no restriction on the location of target objects. The manner of transmitting the signal object, the amount of time required to transmit it, the order in which the transmissions reach the various target objects, and the path for reaching the target objects are undefined."

Thus, the only difference to send signal actions is the determination of targets. This computation is performed by the function *computeTargets*, which uses a signal path expression (see Section 4.2) and the current context object to compute a set of target activity executions. Signal paths are implemented by our tool which is presented in chapter 6.

monitored *computeTargets* : *String* × *BehavioredObject* → $\mathcal{P}$(*ActivityExecution*)

The following rule, ExecuteBroadcastSignalActionExecution, executes broadcast signal actions. SendSignal is used to send the provided signal instance to the computed target executions.

```
ExecuteBroadcastSignalActionExecution ≡
    if Self.mode = created then
        skip
    if Self.mode = enabled then
        Self.mode := running
    if Self.mode = running then
        let
            signal = createInstance(Self.node.signal, [head(Self.input(pin)).value |
                                    pin ∈ Self.node.argument])
            targets = computeTargets(tagValue(Self.node, SignalTargeting, signalPath),
                                     Self.activityExecution.context)
        in
            forall target with target ∈ targets
                SendSignal(signal, target)
            Self.mode := completed
    if Self.mode = completed then
        EnqueueEvent(Self.activityExecution,
                     ActionTerminationEvent(Self, false, ∅)) (→ 5.5.1.1)
        Self.mode := terminating
    if Self.mode = abortRequested then
        Self.mode := terminated
```

5.7 Computation and selection of token offers

This section deals with computation of token flow in UML 2 activity diagrams. After giving an overview in the next section, we introduce the relevant data structures in Section 5.7.2. The following sections describe the different aspects of the token flow semantics.

5.7.1 Overview

According to the UML specification [UML05] (e.g. on p. 302 or on p. 365), nodes offer tokens on outgoing edges. Further propagation of these offers and their selection at destinations is, however, not discussed in the specification. Our approach to computation of relevant token offers is divided into five steps:

1. Creation of initial token offers at flow sources which currently hold control or data tokens.
2. Propagation of token offers to flow destinations.
3. Selection of suitable token offers at those destinations.
4. Determining interruptible activity regions to be aborted and removing offers in these regions.
5. Creating additional selections for accept event actions without incoming edges and input pins in interruptible activity regions.

First, new token offers are created for tokens resting on outgoing edges of actions or initial nodes (being sources of control flows) or at object nodes (being sources of object flows). This is elaborated in Section 5.7.3. Second, the token offers are propagated through the activity graph towards the consuming destination nodes, namely actions, object nodes, and final nodes (Section 5.7.4). Third, offers can now be selected at those destination nodes (Section 5.7.5), preparing their flow through the activity graph. Fourth, some of these still preliminary offers have to be removed if they are contained in interruptible activity regions which are interrupted by other selections (Section 5.7.6). Finally, accept event actions without incoming edges and input pins contained in interruptible activity regions have possibly to be activated (Section 5.7.7), if there is a flow entering the region.

Buffering of offers, which complements to the steps described before, is discussed in Section 5.7.8.

5.7.2 Data structures

The specification states that multiple token offers may exist on a single edge [UML05, p. 369], therefore we use the function

controlled *offers* : *ActivityEdge* $\rightarrow \mathcal{P}($ *TokenOffer*$)$

to store temporary token offers. This function has to be **controlled**, since computation of offers must not be shared among different activity executions, and, therefore, it must act locally to an ASM agent.

We also define a function for buffering unsuccessful offers, which is used for the implementation of fork node semantics. Buffers are only relevant at flow destination nodes, as discussed in Section 4.3.8. ASM Rules which relate to buffering are described in Section 5.7.8.

controlled *buffer* : *ActivityEdge* $\rightarrow \mathcal{P}($ *TokenOffer*$)$
initially $\forall e \in$ *ActivityEdge* : $buffer(e) = \emptyset$

5.7.2.1 TokenOffer

An instance of the data structure *TokenOffer* represents the existence of a single data token on a source object node or multiple control tokens from a single source edge which are able to flow to the current position of this offer. We combine multiple control tokens in a single one because they act as one if starting actions is concerned [UML05, p. 302]:

"If multiple control tokens are available on a single edge, they are all consumed."

A token offer consists of the following components:

controlled domain $TokenOffer =_{def} \{offeredToken : Token; paths : \mathcal{P}(ActivityEdge^*); exclude : \mathcal{P}(TokenOffer); include : \mathcal{P}(TokenOffer); buffered : Boolean\}$

In the case of object flows, the function *offeredToken* holds the actual data token, whose possible traversal is represented by this offer. For control flows, a dummy control token is stored which acts as a substitute for all source control tokens. The component *paths* represents the path starting from the source node of the token, leading to the current position of the offer. Actually, a set of paths is needed, since control flows must be included when combined with object flows by a join node, as described in Section 5.7.4.4. The function *paths* serves three purposes:

1. To remove control tokens from source edges in case of control flows or object flows joined with control flows, see Section 5.8 on executing transitions.
2. To eliminate token selections in interruptible activity regions, see Section 5.7.6.
3. To highlight the edges being traversed if animation in a tool is desired (see Section 6).

According to the specification [UML05, p. 381], "a token in an object node can traverse only one of the outgoing edges". Thus, our algorithm must ensure that these offers exclude each other, as is also the case in Section 5.7.4.3 for decision nodes with overlapping guards. Efficiency dictates that we avoid to try all combinations possible for several nodes with competing edges. To this end, the component *exclude* of *TokenOffer* collects all conflicting offers. The component *include*, finally, contains those offers that have contributed to the current one. We use offers as static information, i.e. they do not move along edges but result from previously computed offers on predecessor edges of the current path. Both the *exclude* and *include* sets are used when selecting token offers at destination nodes in Section 5.7.5.

The function *buffered* finally indicates whether the offer has been fetched by a buffer, which is used to implement the fork node semantics, or else has recently been computed by the propagation algorithm.

The following convenience functions are used by the token selection and transition execution macros.

$IsObjectFlowOffer : TokenOffer \to Boolean$
$IsObjectFlowOffer(o) =_{def} o.offeredToken \in DataToken$
$IsControlFlowOffer : TokenOffer \to Boolean$
$IsControlFlowOffer(o) =_{def} o.offeredToken \in ControlToken$

$offersForNode : ActivityNode \to \mathcal{P}(TokenOffer)$
$offersForNode(n) =_{def} \bigcup\{offers(e) \mid e \in incoming(n)\}$

$sourceEdges : TokenOffer \to \mathcal{P}(ActivityEdge)$
$sourceEdges(o) =_{def} \{head(p) \mid p \in o.paths\}$
$sources : TokenOffer \to \mathcal{P}(FlowSource)$
$sources(o) =_{def} \{e.source \mid e \in sourceEdges(o)\}$
$target : TokenOffer \to FlowDestination$
$target(o) =_{def} last(elementAt(o.paths.asList, 1)).target$

5.7.2.2 TokenSelection

After token offers have been computed, token selections are created by the rule SelectTokenOffers described in Section 5.7.5. A token selection consists of a flow destination and a set of

token offers which may activate this destination node. A destination node comprises the domains *Action*, *FinalNode*, *CentralBufferNode* and *ActivityParameterNode* with outgoing parameters. The domain *InputPin* is omitted as a destination node for selections, because token offers for input pins are included in the selection for the corresponding action.

Notice that the set of token offers can also be empty. This is the case if actions without incoming edges and input pins have to be activated.

The domain *TokenSelection* represents token selections:

controlled domain $TokenSelection =_{def} (FlowDestination \setminus InputPin) \times \mathcal{P}(TokenOffer)$

$sources : TokenSelection \to \mathcal{P}(FlowSource)$
$sources((\_, O)) =_{def} \bigcup_{o \in O} sources(o)$
$targets : TokenSelection \to \mathcal{P}(FlowDestination)$
$targets((\_, O)) =_{def} \{target(o) \mid o \in O\}$
$paths : TokenSelection \to \mathcal{P}(ActivityEdge^*)$
$paths((\_, O)) =_{def} \bigcup_{o \in O} o.paths$

The functions *sources*, *targets* and *paths* aggregate the related values of all token offers contained in the specified selection.

5.7.3 Creation of token offers

The first step in computation of token offers is the creation of initial offers at source edges of flow sources. The rule ComputeTokenOffers creates these offers and calls the propagation rule afterwards.

ComputeTokenOffers ≡
    ClearFlowInformation
    seq
    InitializeFlowsForControlFlowSources
    InitializeFlowsForObjectFlowSources
    seq
    PropagateFlowInformation (→ 5.7.4)

First, all previous computation results are cleared by calling ClearFlowInformation. To ensure that the propagation algorithm visits each node only once, a *visited*-flag is introduced for each node. This flag is initialized with *false* for control nodes and set to *true* for all other activity nodes.

controlled $visited : ActivityNode \to Boolean$
$ArePredecessorsVisited : ActivityNode \to Boolean$
$ArePredecessorsVisited(n) =_{def} \forall p \in predecessors(n) : visited(p)$

ClearOffers : *ActivityNode* → *Void*
ClearOffers(n) ≡
    forall e **with** $e \in n.outgoing$
        $offers(e) := \emptyset$

ClearFlowInformation ≡
    forall n **with** $n \in Self.activity.node$
        $visited(n) := n \in (ActivityNode \setminus ControlNode)$
        ClearOffers(n)

There are two InitializeFlows macros, one for control flows and one for object flows. Figures 5.8 a) and b) show two examples for control flow offers. In a), each outgoing edge of action A holds a token, depicted by ○. These – resting – tokens are contained in the functions *controlTokens*. Only the left token is offered (⊜) to its edge, since the false guard prevents the other from flow ability. In b), both outgoing edges of the initial node hold control tokens, and thus, two offers are created.

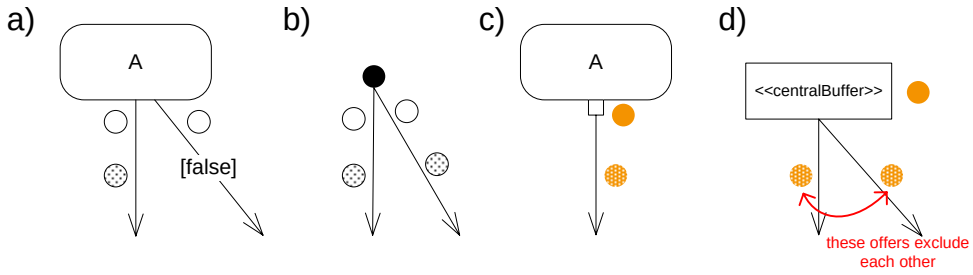

Figure 5.8: Initiation of token flow computation

The following rule implements control flow initialization. It processes all sources of control flows, namely *Action* and *InitialNode*, in parallel and creates new token offers on outgoing edges if they hold at least one token and their guards evaluate to true. Only nodes of the current activity have to be taken into account. Therefore the intersection is with $Self.activity.node$. *Self* denotes the current activity execution and *node*, coming from the UML meta model 5.2.2, returns all nodes of an activity.

InitializeFlowsForControlFlowSources ≡
forall n **with** $n \in (ControlFlowSource \cap Self.activity.node)$
$visited(n) := true$
forall e **with** $e \in n.outgoing$
if $|controlTokens(e)| > 0 \ \wedge \ IsGuardTrue(e, Self.context)$ **then**
let
$t = new(TokenOffer)$
in
$t.offeredToken := new(ControlToken)$
$t.paths := \{[e]\}$
$t.exclude := \emptyset$
$t.include := \emptyset$
$t.buffered := false$
$offers(e) := \{t\}$

A dummy control token is assigned for each token offer, which represents all control tokens at the source edge (when flowing, all tokens will be removed). *Paths* is assigned a list with the current edge and, being the first offer of a token, *include* is initialized as empty. Since tokens on outgoing edges of control flow sources do not compete for traversal (in contrast to object flows), *exclude* is also set to empty. The function *buffered* is initialized with false, as is the case for all other macros relating to computation of token offer propagation. Finally, the offer is stored in the *offers* set of the edge.

For checking the guards, the following monitored (and thus not further specified [BS03]) function *IsGuardTrue* is defined, which evaluates the guard of the given edge. As a guard can refer to attributes of the context class and the current data token offered on the edge (see Section 5.1), the corresponding arguments *BehavioredObject* and *Token* are provided. If the offered token is a control token, the guard must not refer to the token value. This can be assured syntactically.

monitored $IsGuardTrue : ActivityEdge \times BehavioredObject \times Token \rightarrow Boolean$
$IsGuardTrue : ActivityEdge \times BehavioredObject \rightarrow Boolean$
$IsGuardTrue(e, o) =_{def} IsGuardTrue(e, o, undefined)$

For sources of object flow – we assume a FIFO order of data tokens on object nodes – [UML05, p. 380], *offeredToken* holds the first data token from the function *dataTokens* of the node considered. Further, according to p. 381

> "A token in an object node can traverse only one of the outgoing edges."

Therefore, *exclude* is initialized to contain all edges except the current, since all outgoing edges of object nodes compete with each other. All other functions are treated similarly to control flows. Figures 5.8 c) and d) show two examples for object flow offers. The offers in d) exclude each other, since they originate from the same object node.

controlled $t : Nat \rightarrow TokenOffer$

InitializeFlowsForObjectFlowSources $\equiv$
forall n **with** $n \in (ObjectFlowSource \cap Self.activity.node) \ \wedge \ |dataTokens(n)| > 0$
$visited(n) := true$
let
$m = |n.outgoing|$
in
forall i **with** $1 \leq i \leq m$
$t(i) := new(TokenOffer)$
seq
forall i **with** $1 \leq i \leq m$
let
$e = outgoing(n, i)$
in
if $IsGuardTrue(e, Self.context, head(dataTokens(n)))$ **then**
$t(i).offeredToken := head(dataTokens(n))$
$t(i).paths := \{[e]\}$
$t(i).exclude := \{t(j) \mid 1 \leq j \leq m \ \wedge \ i \neq j\}$
$t(i).include := \emptyset$
$t(i).buffered := false$
$offers(e) := \{t(i)\}$

5.7.4 Propagation of token offers

After all initial offers have been created, PropagateFlowInformation distributes them by iteratively calling rules for the join, decision, merge, and fork nodes. Each node is processed once, and only if all predecessors have been handled. The ASM **iterate** command executes all rules in parallel until the update set is empty, i.e. no more changes occur.

PropagateFlowInformation $\equiv$
iterate
PropagateFlowForMergeNode ($\rightarrow$ 5.7.4.1)
PropagateFlowForForkNode ($\rightarrow$ 5.7.4.2)
PropagateFlowForDecisionNode ($\rightarrow$ 5.7.4.3)
PropagateFlowForJoinNode ($\rightarrow$ 5.7.4.4)

5.7.4.1 MergeNode

The propagation for the merge nodes is simple, since [UML05, p. 374]

> "All tokens offered on incoming edges are offered to the outgoing edge."

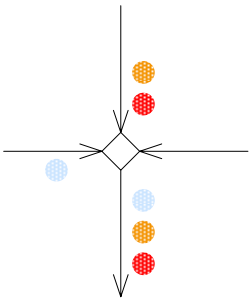

Figure 5.9: Propagation of token offers at *MergeNode*

which is illustrated in Figure 5.9. We check whether the token satisfies the guard of the outgoing edge, and calculate the new token offers as shown in the following rule.

PropagateFlowForMergeNode ≡
forall n **with** $n \in (\mathit{MergeNode} \cap \mathit{Self}.\mathit{activity}.\mathit{node}) \ \wedge \ \mathit{ArePredecessorsVisited}(n)$
$\wedge \neg \mathit{visited}(n)$
ClearOffers(n)
seq
$\mathit{visited}(n) := \mathit{true}$
forall e **with** $e \in \mathit{incoming}(n)$
forall t **with** $t \in \mathit{offers}(e)$
if $\mathit{IsGuardTrue}(\mathit{outgoing}(n,1), \mathit{Self}.\mathit{context}, t.\mathit{offeredToken})$ **then**
let
$t_{out} = \mathit{new}(\mathit{TokenOffer})$
in
$t_{out}.\mathit{offeredToken} := t.\mathit{offeredToken}$
$t_{out}.\mathit{paths} := \{p \oplus \mathit{outgoing}(n,1) \mid p \in t.\mathit{paths}\}$
$t_{out}.\mathit{exclude} := t.\mathit{exclude}$
$t_{out}.\mathit{include} := t.\mathit{include} \cup \{t\}$
$t_{out}.\mathit{buffered} := \mathit{false}$
add t_{out} **to** $\mathit{offers}(\mathit{outgoing}(n,1))$

Each propagation rule first clears all offers on all outgoing edges of the current node by calling ClearOffers. This is because some offers must be re-computed when selecting tokens which invalidate a join condition (see Section 5.7.5).

All paths from the function *paths* are extended by the current outgoing edge, and the base token offer is added to the *include* set. Finally, the new offers are stored in the *offers* set of the outgoing edge. The ASM **add** [GT01] command is used because of possibly concurrent updates, caused by using **forall** for parallel execution of the rules' body.

5.7.4.2 ForkNode

The behavior of fork nodes is illustrated in Figure 5.10. The specification states that [UML05, p. 363]

> "Tokens arriving at a fork are duplicated across the outgoing edges. If at least one outgoing edge accepts the token, duplicates of the token are made and one copy traverses each edge that accepts the token."

PropagateFlowForForkNode ≡

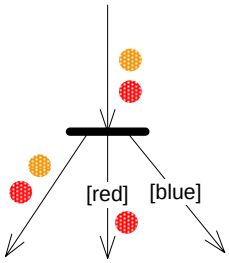

Figure 5.10: Propagation of token offers at *ForkNode*

forall n **with** $n \in (\mathit{ForkNode} \cap \mathit{Self}.\mathit{activity}.\mathit{node}) \ \wedge \ \mathit{ArePredecessorsVisited}(n)$
$\wedge \neg \mathit{visited}(n)$
ClearOffers(n)
seq
$\mathit{visited}(n) := \mathit{true}$
forall t **with** $t \in \mathit{offers}(\mathit{incoming}(n, 1))$
forall e **with** $e \in n.\mathit{outgoing}$
if $\mathit{IsGuardTrue}(e, \mathit{Self}.\mathit{context}, t.\mathit{offeredToken})$ **then**
let
$t_{out} = \mathit{new}(\mathit{TokenOffer})$
in
$t_{out}.\mathit{offeredToken} := t.\mathit{offeredToken}$
$t_{out}.\mathit{paths} := \{p \oplus e \mid p \in t.\mathit{paths}\}$
$t_{out}.\mathit{exclude} := t.\mathit{exclude}$
$t_{out}.\mathit{include} := t.\mathit{include} \cup \{t\}$
$t_{out}.\mathit{buffered} := \mathit{false}$
add t_{out} **to** $\mathit{offers}(e)$

The calculation of the offers is almost identical to merge nodes. In this respect we deviate from the specification [UML05, p. 363] that requires tokens to be buffered at intermediate locations if the guard is true but they are not accepted by target nodes, as stated by

> "The outgoing edges that did not accept the token due to failure of their targets to accept it, keep their copy in an implicit FIFO queue until it can be accepted by the target. The rest of the outgoing edges do not receive a token (these are the ones with failing guards). This is an exception to the rule that control nodes cannot hold tokens if they are blocked from moving downstream."

This requirement was introduced late in the specification process, after the final adopted version of the specification [UML03] had been published. We decided against incorporating this functionality in our ASM rules, because fork buffering may lead to unexpected behavior in combination with guards, as is discussed in Section 4.3.8.

5.7.4.3 DecisionNode

The specification of decision nodes [UML05, p. 349] requires that

> "Each token arriving at a decision node can traverse only one outgoing edge. [...] Each token offered by the incoming edge is offered to the outgoing edges. Most commonly, guards of the outgoing edges are evaluated to determine which edge should be traversed."

The modeler, however, must ensure that only one outgoing edge is actually traversed. Additionally, [UML05, p. 349] states that

> "Notice that the semantics only requires that the token traverse one edge, rather than be offered to only one edge. Multiple edges may be offered the token, but if only one of them has a target that accepts the token, then that edge is traversed. If multiple edges accept the token and have approval from their targets for traversal at the same time, then the semantics is not defined."

It is, however, left unspecified what the "approval" proviso means. As long as two outgoing edges are not traversed simultaneously by the same token, we propose that any selection of token offers may be chosen. This is achieved by filling the *exclude* sets of the outgoing offers accordingly.

An example is shown in Figure 5.11. A single offer is generated for whereas two competing offers are created for . Since all ordinary guards fail for , it is offered to the "else" edge. The predefined guard "else" may be defined for at most one outgoing edge. It succeeds if the token is refused by all the other edges outgoing from the decision node.

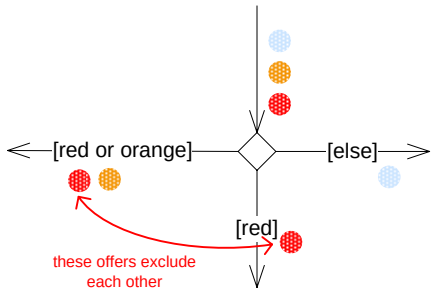

Figure 5.11: Propagation of token offers at *DecisionNode*

The following rules implement the distribution of token offers at decision nodes.

ElseEdge : *ActivityNode* → *ActivityEdge*
ElseEdge(n) ≡
    choose e **with** $e \in n.outgoing : e.guard = 'else'$
        result := e

HasElseEdge : *ActivityNode* → *Boolean*
HasElseEdge(n) = ElseEdge(n) ≠ *undefined*

PropagateFlowForDecisionNode ≡
    forall n **with** $n \in (DecisionNode \cap Self.activity.node) \ \wedge \ ArePredecessorsVisited(n)$
                $\wedge \neg visited(n)$
        ClearOffers(n)
        seq
        $visited(n) := true$
        forall t **with** $t \in offers(incoming(n, 1))$
            let
                $acceptingEdges = [e \in n.outgoing \mid e \neq \text{ElseEdge}(n) \wedge$
                    $IsGuardTrue(e, Self.context, t.offeredToken)]$
            in
                if $|acceptingEdges| > 0$ **then**
                    forall i **with** $1 \leq i \leq |acceptingEdges|$

```
        t(i) := new(TokenOffer)
    seq
    forall i with 1 ≤ i ≤ |acceptingEdges|
        t(i).offeredToken := t.offeredToken
        t(i).paths := {p ⊕ elementAt(acceptingEdges, i) | p ∈ t.paths}
        t(i).exclude := {t(j) | 1 ≤ j ≤ |acceptingEdges| ∧ i ≠ j} ∪ t.exclude
        t(i).include := t.include ∪ {t}
        t(i).buffered := false
        add t(i) to offers(elementAt(acceptingEdges, i))
else
    if HasElseEdge(n) then
        let
            t_out = new(TokenOffer)
        in
            t_out.offeredToken := t.offeredToken
            t_out.paths := {p ⊕ ELSEEDGE(n) | p ∈ t.paths}
            t_out.exclude := t.exclude
            t_out.include := t.include ∪ {t}
            t_out.buffered := false
            add t_out to offers(ELSEEDGE(n))
```

The variable *acceptingEdges* holds all outgoing edges, except the "else" edge, whose guards evaluate to true. New, mutually exclusive, token offers are then created for these edges. The exclusion is guaranteed by storing the competing accepting offers in the *exclude* set of the current offer. In case of no edges accepting an offer, it is forwarded to the optional "else" edge.

5.7.4.4 JoinNode

Join nodes are the most complex kind of control nodes [UML05, p. 369]:

> "If there is a token offered on all incoming edges, then tokens are offered on the outgoing edge according to the following join rules:
>
> 1. If all the tokens offered on the incoming edges are control tokens, then one control token is offered on the outgoing edge.
> 2. If some of the tokens offered on the incoming edges are control tokens and others are data tokens, then only the data tokens are offered on the outgoing edge. Tokens are offered on the outgoing edge in the same order they were offered to the join.
>
> Multiple control tokens offered on the same incoming edge are combined into one before applying the above rules."

The first case is shown in Figure 5.12 a), where only data tokens are joined. Figure b) shows the joining of control tokens. Note that we ignore any "order" of tokens, since this term is not clearly described in the specification, as discussed in Section 5.9.

```
controlFlowOffers : ActivityNode → P(TokenOffer)
controlFlowOffers(n) =_def ⋃{offers(e) | e ∈ incoming(n) ∧ IsControlFlow(e)}

PROPAGATEFLOWFORJOINNODE ≡
    forall n with n ∈ (JoinNode ∩ Self.activity.node) ∧ ArePredecessorsVisited(n)
                  ∧ ¬visited(n)
        CLEAROFFERS(n)
        seq
```

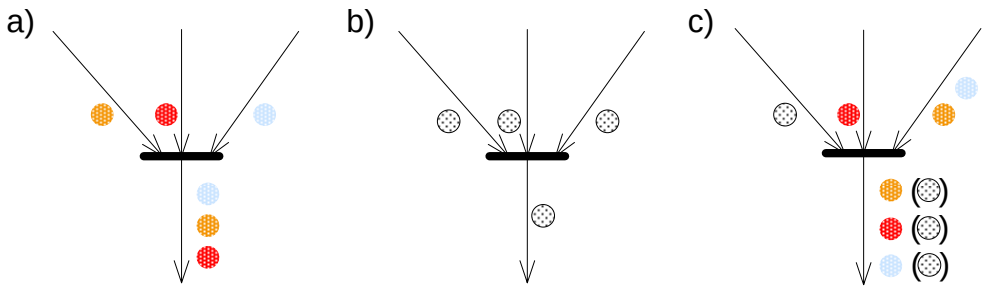

Figure 5.12: Propagation of token offers at *JoinNode*

$visited(n) := true$
if $\forall e \in incoming(n) : offers(e) \neq \emptyset$ **then**
  let
    $o = outgoing(n, 1)$
  in
    if $IsControlFlow(o)$ **then**
      if $IsGuardTrue(o, Self.context)$ **then**
        let
          $t_{out} = new(TokenOffer)$
        in
          $t_{out}.offeredToken := new(ControlToken)$
          $t_{out}.paths := \{p \oplus o \mid p \in \bigcup\{t.paths \mid t \in offersForNode(n)\}\}$
          $t_{out}.exclude := \bigcup\{t.exclude \mid t \in offersForNode(n)\}$
          $t_{out}.include := \bigcup\{t.include \cup \{t\} \mid t \in offersForNode(n)\}$
          $t_{out}.buffered := false$
          add t_{out} **to** $offers(o)$
    else
      forall e **with** $e \in incoming(n) \ \wedge \ IsObjectFlow(e)$
        forall t **with** $t \in offers(e)$
          if $IsGuardTrue(o, Self.context, t.offeredToken)$ **then**
            let
              $t_{out} = new(TokenOffer)$
            in
              $t_{out}.offeredToken := t.offeredToken$
              $t_{out}.paths := \{p \oplus o \mid p \in t.paths\} \cup$
                $\{p \oplus o \mid p \in \bigcup\{t'.paths \mid t' \in controlFlowOffers(n)\}\}$
              $t_{out}.exclude := \bigcup\{t'.exclude \mid t' \in offersForNode(n)\}$
              $t_{out}.include := \bigcup\{t'.include \cup \{t'\} \mid t' \in offersForNode(n)\}$
              $t_{out}.buffered := false$
              add t_{out} **to** $offers(o)$

If the outgoing edge is a control flow, all incoming edges are control flows [UML05, p. 369], and only one control token has to be offered to the outgoing edge if its guard permits. The new *exclude* and *include* sets consist of the union of all incoming exclude and include sets, respectively, to prevent conflicting offers to flow that might invalidate the join condition. The function *offersForNode*, defined in Section 5.7.2, is used for this calculation. It collects all offers from all incoming edges of a node. The outgoing edge is appended to all incoming control flow paths that are joined to form the new *paths*. This is done to be able to remove all control tokens on all source edges [UML05, p. 302] if the token offer is selected at a destination node.

If some or all incoming edges are object flows, all offers from these flows have to be forwarded. By joining all incoming control flows to the paths set of each transmitted token offer (illustrated in Figure 5.12 c), they can be removed if the actual transition of the object token takes place. It is reasonable to share the control flows among each offer, because they contribute to the join condition of each data token. As in the case of having control flows only, the include and exclude sets of each offer contain all flows to exclude competing offers. Examples of usage of these sets when join nodes are involved in flow computation, are given in Section 5.7.5.

For efficiency reasons and to keep the algorithm concise, we assume that the incoming token offers are consistent, i.e. they must not compete with each other, as discussed in Sections 5.7.5 and 5.7.9 in detail.

5.7.5 Selection of token offers at targets

Once the token offers have been computed, we select subsets of them to participate in the planned transition. The transition may lead, e.g., to the start of a new action as described by the specification [UML05, p. 302]:

> "An action execution is created when all its object flow and control flow prerequisites have been satisfied (implicit join). [...] The flow prerequisite is satisfied when all of the input pins are offered tokens and accept them all at once, precluding them from being consumed by any other actions."

Other possibilities are moving tokens to central buffer nodes, outgoing activity parameter nodes, and final nodes. The specification, however, does not indicate what to perform if there are enough token offers to conduct several of these operations. We model this by using the ASM iteration and choice constructs in our selection rule SelectTokenOffers now discussed.

The selected subsets are accumulated in *tokenSelections*, which is used afterwards for executing the transitions in Section 5.8.

controlled *tokenSelections* : $\mathcal{P}$(*TokenSelection*)

The local function *taken* keeps track of the selections for each object node to make sure they do not overflow. The iteration may be stopped by choosing n = skipSelection at any stage. Otherwise, we select token offers depending on the kind of node.

```
SelectTokenOffers ≡
    local taken : ObjectNode → Nat
    forall n with n ∈ (ObjectNode ∩ Self.activity.node)
        taken(n) := 0
    tokenSelections := ∅
    seq
    iterate
        choose n with n ∈ ((Action ∪ CentralBufferNode ∪ {n' ∈ ActivityParameterNode |
                n'.parameter.direction ∈ {out, inout, return}} ∪ FinalNode)
                ∩ Self.activity.node) ∪ {skipSelection}
            if n ∈ Action then
                SelectTokenOffersForAction(n, taken)
            if n ∈ (CentralBufferNode ∪ ActivityParameterNode) then
                SelectTokenOffersForCentralBufferAndActParameterNode(n, taken)
            if n ∈ FinalNode then
                SelectTokenOffersForFinalNode(n, taken)
```

We do not specify if and how many transitions should be executed, by using a combination of **iterate**, **choose** and skipSelection. This conforms to the specification as there is no information on this topic (see Section 4.4.2). The ASM semantics of **iterate** can lead to termination of the rule,

although offers to select are still available. We propose a solution to this by introducing a slightly modified **choose** construct in Section 5.7.9.

The selection for action nodes is performed according to the specification cited above. We select a subset of token offers S_i for each input pin i of the action. Conditions for the acceptance of tokens by input pins are that the number of selected tokens is between *lower* and *upper* [UML05, p. 249], and that the total number of tokens resting on each pin does not exceed its upper bound [UML05, p. 380]. If an appropriate selection has been found, we commit to it and update the remaining offers according to the specification. To this end, the UpdateOffers discussed below removes all selected and competing offers from the activity graph.

SelectTokenOffersForAction$(n, taken) \equiv$

let

$m = |incoming(n)|$

$p = |input(n)|$

in

if $\forall 1 \leq i \leq m : offers(incoming(n,i)) \neq \emptyset$ **then**

choose $S_1, \ldots, S_p$ **with** $\forall 1 \leq i \leq p : S_i \subseteq offersForNode(input(n,i))$
$\wedge\ lower(input(n,i)) \leq |S_i| \leq \min(upper(input(n,i)),$
$upperBound(input(n,i)) - |dataTokens(input(n,i))|$
$-taken(input(n,i)))$

let

$selection = \bigcup_{1 \leq i \leq m} offers(incoming(n,i)) \cup \bigcup_{1 \leq i \leq p} S_i$

in

UpdateOffers$(selection)$

$tokenSelections := tokenSelections \cup \{(n, selection)\}$

forall i **with** $1 \leq i \leq p$

$taken(input(n,i)) := taken(input(n,i)) + |S_i|$

Figure 5.13 shows some scenarios for token selections. In a), all incoming control flow offers (⊜) and either of the data token offers (● or ●) may be chosen for a single selection. The input pin only consumes a single data token, because *lower* and *upper* default to 1. The remaining data token is not used for the creation of the execution for action A. In b), we have a single input pin with *lower*=2 and *upper*=3. Therefore, we can choose any of the subsets {●,●}, {●,●}, {●,●} or {●,●,●} for a selection.

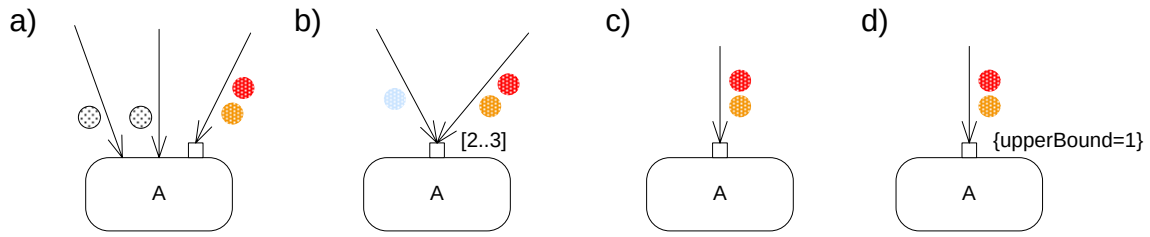

Figure 5.13: Selection of token offers for *Action*

Case c) yields *two* selections causing action A to start twice. Finally, in d), if the input pin does not currently hold a token, only one selection (either with ● or ●) is generated due to *upperBound*=1.

The selection rule for central buffer nodes and outgoing activity parameter nodes is considerably simpler. It can be viewed as a special case of the rule for actions with only one input pin and no incoming control flows.

SelectTokenOffersForCentralBufferAndActParameterNode$(n, taken) \equiv$

choose S **with** $S \subseteq offersForNode(n)$
$\wedge\ 0 < |S| \leq upperBound(n) - |dataTokens(n)| - taken(n)$
UPDATEOFFERS(S)
$tokenSelections := tokenSelections \cup \{(n, S)\}$
$taken(n) := taken(n) + |S|$

For Figure 5.14 a), any subset may be chosen for a selection, and, due to the iteration over all nodes in SELECTTOKENOFFERS, multiple selections can be generated. In b), the central buffer node has *upperBound*=2 and already holds token, therefore only one of the offers and may be selected.

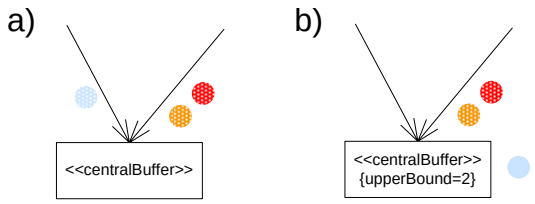

Figure 5.14: Selection of token offers for *CentralBufferNode* and outgoing *ActivityParameterNode*

The last selection rule refers to final nodes, subsuming *ActivityFinalNode* and *FlowFinalNode*. Each offer on any incoming edge can yield a selection. Examples of final nodes with offers are shown in Figure 5.15.

SELECTTOKENOFFERSFORFINALNODE($n, taken$) $\equiv$
choose e **with** $e \in incoming(n)\ \wedge\ |offers(e)| > 0$
choose d **with** $d \in offers(e)$
UPDATEOFFERS($\{d\}$)
$tokenSelections := tokenSelections \cup \{(n, \{d\})\}$

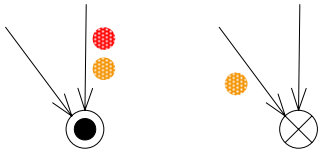

Figure 5.15: Selection of token offers for *FinalNode*

To avoid an inefficient search and further dependencies at destination and join nodes, we assume that the incoming token offers are consistent. The implications on the algorithm if allowing inconsistent offers is discussed in Section 5.7.9. We define the following constraints for the ASM:

$AreConsistent : TokenOffer \times TokenOffer \rightarrow Boolean$
$AreConsistent(t_1, t_2) =_{def} (t_1.exclude \cap (t_2.include \cup \{t_2\}) = \emptyset)$
$\wedge\ (t_2.exclude \cap (t_1.include \cup \{t_1\}) = \emptyset)$

constraint $\forall n \in ObjectNode \cup JoinNode : \forall t_1, t_2 \in offersForNode(n) : AreConsistent(t_1, t_2)$
constraint $\forall n \in Action : \forall t_1, t_2 \in (offersForNode(n) \cup \bigcup_{i \in input(n)} offersForNode(i)) : AreConsistent(t_1, t_2)$

The modeler can ensure this, e.g., by placing appropriate guards on competing edges that lead to the same destination nodes. A stronger, syntactic condition is the absence of two paths from the same decision or object node to the same action, join, or object node. Here, an action together with its input pins is considered as one node.

The macro UpdateOffers, which is called after a selection has been made, removes the selection of token offers and all offers inconsistent to it. This is implemented by the subrule RemoveInconsistentAndSelectedOffers which in turn calls a rule for removing buffered offers:

UpdateOffers : $\mathcal{P}(\mathit{TokenOffer}) \rightarrow \mathit{Void}$
UpdateOffers($\mathit{selection}$) $\equiv$
    RemoveInconsistentAndSelectedOffers($\mathit{selection}$)
    InvalidateJoinsForInconsistentOffers($\mathit{selection}$)
    seq
    PropagateFlowInformation ($\rightarrow$ 5.7.4)
    seq
    FetchBufferedTokens

RemoveInconsistentAndSelectedOffers : $\mathcal{P}(\mathit{TokenOffer}) \rightarrow \mathit{Void}$
RemoveInconsistentAndSelectedOffers($\mathit{selection}$) $\equiv$
    forall e **with** $e \in (\mathit{ActivityEdge} \cap \mathit{Self}.\mathit{activity}.\mathit{edge})$
        $\mathit{offers}(e) := \{t \in \mathit{offers}(e) \mid t \notin \mathit{selection} \ \wedge \ \forall t' \in \mathit{selection} : \mathit{AreConsistent}(t, t')\}$
    RemoveInconsistentAndSelectedBufferedOffers($\mathit{selection}$) ($\rightarrow$ 5.7.8)

Figure 5.16 shows an example of an object node with several destinations. After the offer has been accepted at action A, the competing offers for action B and the central buffer node are deleted by this rule. Note that successful offers, from which the selected offer derives, do not need to be removed from the graph. Therefore, the offer on the leftmost outgoing edge of the source object node can be left untouched.

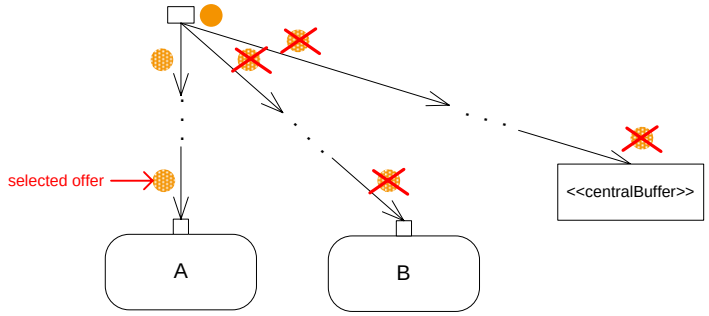

Figure 5.16: Removal of inconsistent offers

If any inconsistent offers are removed from a join node, we re-propagate this information since it may affect other token offers originating at that node. We, therefore, first reset the *visited* flag of these join nodes and all successor control nodes to false. This is achieved by the rule InvalidateJoinsForInconsistentOffers. Predecessor nodes remain unaffected. The propagation rule PropagateFlowInformation rule introduced in Section 5.7.4 is called afterwards, recomputing the flow information.

InvalidateJoinsForInconsistentOffers : $\mathcal{P}(\mathit{TokenOffer}) \rightarrow \mathit{Void}$
InvalidateJoinsForInconsistentOffers($\mathit{selection}$) $\equiv$

forall n **with** $n \in (\mathit{JoinNode} \cap \mathit{Self}.\mathit{activity}.\mathit{node}) \ \wedge \ \exists e \in n.\mathit{incoming} : \exists t \in \mathit{offers}(e) :$
$\exists t' \in \mathit{selection} : \neg \mathit{AreConsistent}(t, t')$
forall n' **with** $n' \in (\textsc{AllControlNodeSuccessors}(n) \cup \{n\})$
$\mathit{visited}(n') := \mathit{false}$

$\textsc{AllControlNodeSuccessors} : \mathit{ActivityNode} \rightarrow \mathcal{P}(\mathit{ControlNode})$
$\textsc{AllControlNodeSuccessors}(n) \equiv$
local $\mathit{nodes} : \mathcal{P}(\mathit{ControlNode}) := \emptyset$
forall e **with** $e \in n.\mathit{outgoing}$
if $e.\mathit{target} \in \mathit{ControlNode}$ **then**
add $e.\mathit{target}$ **to** nodes
add $\textsc{AllControlNodeSuccessors}(e.\mathit{target})$ **to** nodes
seq
result := nodes

We discuss the necessity for re-propagation starting from join nodes by means of the example in Figure 5.17. In a), the join condition is met and offers 4 and 5 are put on the outgoing edge of the join node. The effect of selecting offer 2, which competes with offer 1, and contributes to the join condition, is shown in b): offer 1 must be removed, and thus the join condition is no longer valid. Offer 5 must be withdrawn.

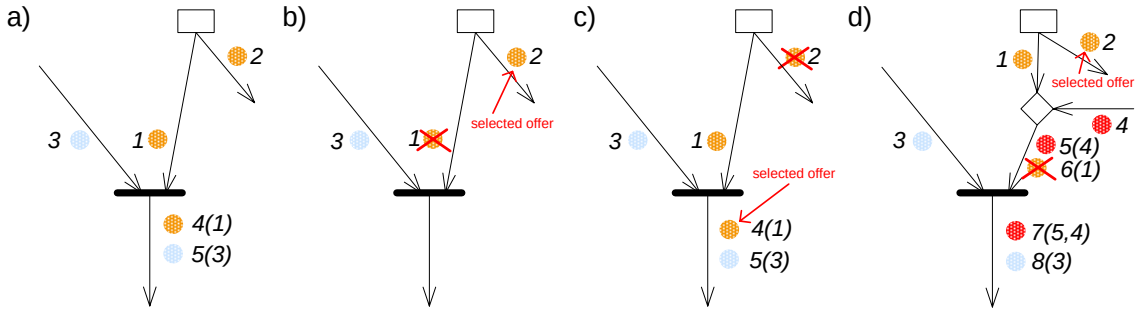

Figure 5.17: Effect of removing inconsistent offers at join nodes

The effect of selecting offer 4, on the other hand, is depicted in c), where the competing offer 2 has to be removed. This is implemented by collecting all these offers in the *exclude* set, compare the propagation rule for join nodes in Section 5.7.4.4.

A scenario where a removal of offers at join nodes does not prevent other tokens from flowing, is illustrated in d). Although offer 6 is removed for selecting the competing offer 2, the join condition is still valid since there is another offer on the same incoming edge of the join node (offer 5). Offers 7 and 8 (stemming from offer 3 resp. offer 5) can therefore be created.

5.7.6 Handling interruptible activity regions

The selections yielded by SelectTokenOffers are still preliminary. Some of the selections chosen may contain interrupting edges that abort interruptible activity regions. Executing a transition for such a selection aborts all inner actions and can invalidate other concurrent transitions in these regions, as stated in [UML05, p. 367]:

> "When a token leaves an interruptible region via edges designated by the region as interrupting edges, all tokens and behaviors in the region are terminated."

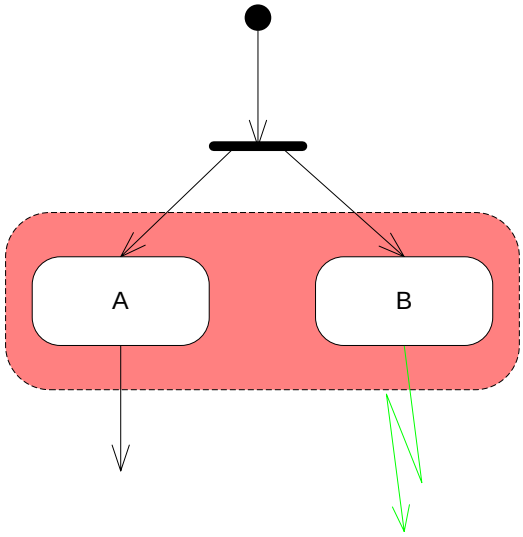

Figure 5.18: Leaving an interruptible activity region

Figure 5.18 shows how our tool (see chapter 6) illustrates the abort of an interruptible activity region.

We, therefore, must – besides determining the regions to be interrupted – remove all selections conflicting with interrupting selections. Concurrent non-interrupting flows to nodes outside the region, however, are executed [UML05, p. 367]:

> "If a non-interrupting edge is passing a token from a source node in the region to target node outside the region, then the transfer is completed and the token arrives at the target even if an interruption occurs during the traversal."

In addition to the informal semantics of interruptible activity regions discussed in the specification, we also have to handle the following issues detailed in Section 4.4:

1. The specification gives no information on concurrent flows leading into aborted regions. Offers along such flows may originate from a node outside the region, or may re-enter the region after leaving it. Since either keeping or destroying such tokens can be useful, our algorithm can be adapted to both alternatives. For this purpose we use the ignoreFlowIntoInterruptedRegion configuration tag, as introduced in Section 4.4.4.

2. The passing of multiple interrupting edges is not discussed. There are scenarios where this behavior should be avoided. By introducing a singleInterrupt tag for interruptible activity regions and priority tags for interrupting edges, the required behavior can also be adjusted.

3. According to the definition of included nodes in *ActivityGroup*, nested regions are *not* aborted. Instead of this unexpected behavior we propose to interrupt all nested regions.

The following macro handles the removal of token selections in interrupted regions, as well as the determination of the regions to be interrupted.

RemoveFlowsInInterruptedRegions ≡
  $regionsToInterrupt := \emptyset$
  seq
  forall r **with** $r \in (InterruptibleActivityRegion \cap Self.activity.group)$
    $\wedge\ IsInterrupted(r)\ \wedge\ (\nexists r' \in parents(r) : IsInterrupted(r'))$
    remove $\{s \in tokenSelections \mid HasInnerFlow(s, r)\}$ **from** $tokenSelections$

if $tagValue(r, \mathsf{InterruptibleActivityRegionHandling}, \mathsf{ignoreFlowIntoInterruptedRegion}) = true$ **then**
  remove $\{s \in tokenSelections \mid HasFlowInto(s, r)\}$ **from** $tokenSelections$
let
  $e = \textsc{ChooseInterruptingEdge}(\bigcup_{s \in tokenSelections} interruptingEdge(s, r))$
in
  if $tagValue(r, \mathsf{InterruptibleActivityRegionHandling}, \mathsf{singleInterrupt}) = true$ **then**
    remove $\{s \in tokenSelections \mid Interrupts(s, r') \wedge e \notin interruptingEdge(s, r')\}$
    from $tokenSelections$
add $(\{r\} \cup children(r))$ **to** $regionsToInterrupt$

We iterate all outmost interruptible activity regions which are interrupted by at least one token offer. We then remove all token selections that contain at least one "inner flow", denoting a flow which is located completely inside the region or any subregion. To this end, we define the ASM functions *IsInterrupted* and *HasInnerFlow*:

$IsInterrupted : InterruptibleActivityRegion \to Boolean$
$IsInterrupted(r) =_{def} \exists s \in tokenSelections : Interrupts(s, r)$

$Interrupts : TokenSelection \times InterruptibleActivityRegion \to Boolean$
$Interrupts(s, r) =_{def} interruptingEdge(s, r) \neq \emptyset$

$interruptingEdge : TokenSelection \times InterruptibleActivityRegion \to \mathcal{P}(ActivityEdge)$
$interruptingEdge(s, r) =_{def} \{e \in \uplus asList(paths(s)) \mid r \in e.interrupts\}$

$HasInnerFlow : TokenSelection \times InterruptibleActivityRegion \to Boolean$
$HasInnerFlow(s, r) =_{def} \exists p \in paths(s) : nodesOnPath(p) \subseteq innerNodes(r)$

$innerNodes : InterruptibleActivityRegion \to \mathcal{P}(ActivityNode)$
$innerNodes(r) =_{def} containedNode(r) \cup \bigcup_{r' \in children(r)} containedNode(r')$

$nodesOnPath : ActivityEdge^* \to \mathcal{P}(ActivityNode)$
$nodesOnPath(p) =_{def} \{e.source \mid e \in p\} \cup \{e.target \mid e \in p\}$

The following functions *parents* and *children* collect all parent- resp. subregions and are used for determining the outermost interrupted region and to collect all nodes including those in nested regions (see function *innerNodes* above).

$parents : InterruptibleActivityRegion \to \mathcal{P}(InterruptibleActivityRegion)$
$$parents(r) =_{def} \begin{cases} \{superGroup(r)\} \cup parents(superGroup(r)), & \textbf{if } superGroup(r) \neq undefined \\ \emptyset, & \textbf{otherwise} \end{cases}$$
$children : InterruptibleActivityRegion \to \mathcal{P}(InterruptibleActivityRegion)$
$$children(r) =_{def} \begin{cases} subGroup(r) \cup \bigcup_{r' \in subGroup(r)} children(r'), & \textbf{if } subGroup(r) \neq \emptyset \\ \emptyset, & \textbf{otherwise} \end{cases}$$

After removing all inner flows in RemoveFlowsInInterruptedRegions, we eliminate all selected offers that do not leave the region, as checked by *HasInnerFlow*. This happens only if the corresponding configuration tag is set.

$HasFlowInto : TokenSelection \times InterruptibleActivityRegion \to Boolean$
$HasFlowInto(s, r) =_{def} \exists p \in paths(s) : last(p).target \in innerNodes(r)$
$\qquad \wedge\ nodesOnPath(p) \setminus innerNodes(r) \neq \emptyset$

Furthermore, the singleInterrupt tag is checked and, if set, an interrupting edge with the highest priority is chosen by ChooseInterruptingEdge. For the removal of token selections for concurrent interrupting edges to work, we impose the following constraint:

constraint $\forall s \in \mathit{TokenSelection} : \forall r \in \mathit{InterruptibleActivityRegion} : |\mathit{interruptingEdge}(s, r)| \leq 1$

It states that each token selection must only pass at most one interrupting edge. This would not be the case if multiple interrupting flows were joined outside the interruptible region. This, however, is without loss of generality, because these join nodes can be moved inside the region. This constraint can be ensured by checking the syntax of the model accordingly.

ChooseInterruptingEdge : $\mathcal{P}(\mathit{ActivityEdge}) \rightarrow \mathit{ActivityEdge}$
ChooseInterruptingEdge(edges) $\equiv$
choose e **with** $e \in \mathit{edges} \ \wedge \ \mathit{tagValue}(e, \mathsf{InterruptPriority}, \mathsf{priority}) = \max_{e' \in \mathit{edges}} (\mathit{tagValue}(e', \mathsf{InterruptPriority}, \mathsf{priority}))$
result := e

Finally, we collect all regions and subregions to interrupt in *regionsToInterrupt*. This will be used to remove all tokens and to abort all nodes, when executing the transitions in Section 5.8.

controlled $\mathit{regionsToInterrupt} : \mathcal{P}(\mathit{InterruptibleActivityRegion})$

5.7.7 Handling accept event actions

The next step of computing transitions is the activation of accept event actions contained in interruptible activity regions. According to the specification [UML05, p. 367]:

> "AcceptEventActions in the region that do not have incoming edges are enabled only when a token enters the region, even if the token is not directed at the accept event action."

We, therefore, check whether there is any token selection entering a region. To this end, the ASM function *HasFlowInto* defined in the previous section is used. We then create additional token selections for accept event actions if the following conditions hold:

- The action has no incoming edges.
- It is not already executing, as determined by *IsRunning*, defined in Section 5.6.
- It is contained immediately in the entered region (by checking *containedNode*, which comes from the UML meta model) or it is contained in a child region and the regionActivationPolicy tag is set to OnParentFlow for all regions in between. The latter targets the problem of nested regions with accept event actions, see 4.4.4.

$\mathit{IsRunning} : \mathit{Action} \rightarrow \mathit{Boolean}$
$\mathit{IsRunning}(n) =_{def} \exists \mathit{exec} \in \mathit{ActionExecution} : \mathit{exec.node} = n \ \wedge \ \mathit{exec.activityExecution} = \mathit{Self} \wedge \mathit{exec.mode} = \mathsf{running}$

ActivateAcceptEventActions $\equiv$
forall r **with** $r \in (\mathit{InterruptibleActivityRegion} \cap \mathit{Self.activity.group}) \wedge \exists s \in \mathit{tokenSelections} : \mathit{HasFlowInto}(s, r)$
add $\{(n, \emptyset) \mid n \in (\mathit{AcceptEventAction} \cap \mathit{Self.activity.node}) \wedge n.\mathit{incoming} = \emptyset \ \wedge \ (\neg \mathit{IsRunning}(n) \ \vee \ \mathit{IsInterrupted}(r)) \wedge (n \in r.\mathit{containedNode} \ \vee \ (\exists r' \in r.\mathit{children} : n \in r'.\mathit{containedNode} \wedge \forall r'' \in \mathit{regionsFromTo}(r', r) : \mathit{tagValue}(r'', \mathsf{InterruptibleActivityRegionHandling}, \mathsf{regionActivationPolicy})$

$= \text{OnParentFlow}))\}$
to *tokenSelections*

$$regionsFromTo : InterruptibleActivityRegion \times InterruptibleActivityRegion \rightarrow \mathcal{P}(InterruptibleActivityRegion)$$
$$regionsFromTo(r, s) =_{def} \begin{cases} \{r\}, & \textbf{if } r.superGroup = s \\ \{r\} \cup regionsFromTo(r.superGroup, s), & \textbf{otherwise} \end{cases}$$

5.7.8 Buffering of token offers

Buffering of tokens due to the fork node semantics is adjacent to some of the flow computation rules described in the previous sections. The specification [UML05, p. 363] proposes the following semantics:

> "Tokens arriving at a fork are duplicated across the outgoing edges. If at least one outgoing edge accepts the token, duplicates of the token are made and one copy traverses each edge that accepts the token. The outgoing edges that did not accept the token due to failure of their targets to accept it, keep their copy in an implicit FIFO queue until it can be accepted by the target. The rest of the outgoing edges do not receive a token (these are the ones with failing guards)."

As discussed in Section 4.3.8, this semantics can lead to unexpected behavior in combination with guards. Moreover, the intuition of control nodes acting solely as "traffic switches" [UML05, p. 309] is negated. To alleviate these issues to a certain extend, we buffer tokens on incoming edges of destination nodes instead of outgoing edges of fork nodes. To this end, the ASM function *buffer*, which is introduced in Section 5.7.2, holds token offers which must be buffered.

In order to make buffered offers available for SelectTokenOffers, we add these token offers to the *offers* of the incoming edges of destination nodes. This is done after computation of token offers, as well as after their re-computation due to invalidated join nodes. See calling of the FetchBufferedTokens rule in Sections 5.5.6 and 5.7.5.

FetchBufferedTokens ≡
  forall n **with** $n \in (FlowDestination \cap Self.activity.node)$
    forall e **with** $e \in n.incoming$
      $offers(e) := offers(e) \cup buffer(e)$

In addition to removing selected and inconsistent offers from *offers*, UpdateOffers also has to eliminate buffered offers. This is achieved by the following rule:

RemoveInconsistentAndSelectedBufferedOffers : $\mathcal{P}(TokenOffer) \rightarrow Void$
RemoveInconsistentAndSelectedBufferedOffers(*selection*) ≡
  forall e **with** $e \in (ActivityEdge \cap Self.activity.edge) \wedge e.target \in FlowDestination$
    $buffer(e) := \{t \in buffer(e) \mid t \notin selection \wedge \forall t' \in selection : AreConsistent(t, t')\}$

Finally, after offers have been selected and further processed by RemoveFlowsInInterruptedRegions and ActivateAcceptEventActions, token offers to be buffered must be determined. Figure 5.19 shows some scenarios where buffering is needed. In a), the offer on the incoming edge of action A is selected. Since action B cannot start due to a missing token on its right edge, the duplicated offer must be buffered on the left incoming edge. Thus, to determine whether a token has to be buffered, the base tokens of offers have to be compared. Since join nodes also include all incoming control flows with each outgoing token offer, all control tokens must also be incorporated. This is illustrated in b), where a data token is selected that passed a join node with an incoming control flow. Action A currently cannot accept the control token, which must be buffered, because its flow is included with the flow of the selected data token to action B.

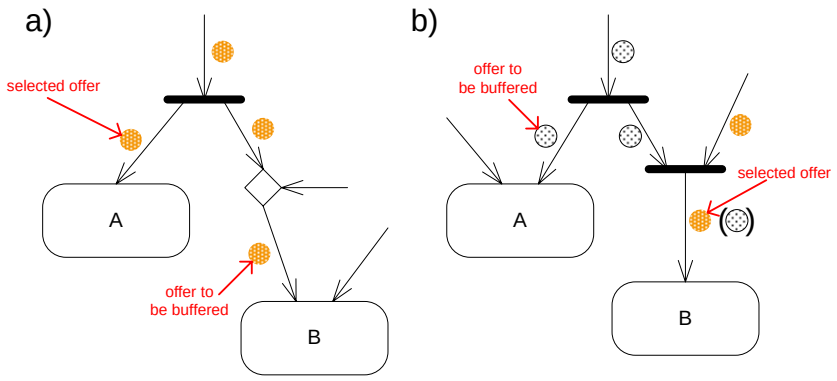

Figure 5.19: Buffering of offers

We use the following function *allTokens* to obtain all control tokens of an offer, together with the actually offered base token:

$allTokens : TokenOffer \rightarrow \mathcal{P}(Token)$
$allTokens(t) =_{def} \{t'.offeredToken \mid t' \in t.include \wedge IsControlFlowOffer(t')\} \cup \{t.offeredToken\}$

The ASM rule which buffers tokens is defined as follows. Besides storing relevant offers in *buffer*, it has to mark all these offers as "buffered', by setting *buffered* to true. This is necessary to prevent these "delayed" transitions from removing tokens from the source node(s) (see Section 5.8). Otherwise, control tokens that have been generated after a token offer has been buffered, were removed although they did not contribute to the buffered flow.

BUFFERTOKENS ≡
forall n **with** $n \in (FlowDestination \cap Self.activity.node)$
forall e **with** $e \in n.incoming$
let
$tokensToBuffer = \{t \in offers(e) \mid \exists(\_, O) \in tokenSelections : \exists o \in O :$
$t.offeredToken \in allTokens(o)\}$
in
forall t **with** $t \in tokensToBuffer$
$t.buffered := true$
$buffer(e) := buffer(e) \cup tokensToBuffer$

5.7.9 Discussion of the token offer computation

In this section, we discuss implications and constraints of the flow propagation algorithm. Hints towards solutions are given for some issues. Incorporating these solutions would lead to unnecessary constraints or to a higher complexity of the algorithm.

Only subsets of possible transitions are executed. To provide the maximum freedom for executing transitions (see 4.4.2), our selection algorithm in Section 5.7.5 relies on nondeterminism by using ASM **iterate** and **choice** constructs. This may lead to unnecessary buffering and re-computation of selections that would, otherwise, be possible. The reason is (besides skipSelection) that **iterate** terminates as soon as a node with no possible selection is chosen which yields an empty update set. To make **iterate** execute until no further selections are possible, we remove skipSelection from the **choose** construct in SELECTTOKENOFFERS. Additionally, we introduce the following modified **choose** command, called **choose'**:

choose' x **with** φ $P \equiv$
local z
if $\exists x : \varphi$ **then**
choose x **with** φ
P
else
$z := 0$
$z := 1$

All calls of **choose** in the SELECTTOKENOFFERFOR* rules are to be substituted with **choose'**. Since **choose** works as an angelic choice operator [WM97, BS03], which delays generating the inconsistent update set (incorporated in **choose'**) as much as possible, all successful selections are made first.

Requirement of consistent offers. As described in Section 5.7.5, we require all incoming offers for actions, object nodes and join nodes to be consistent. This considerably simplifies the selection algorithm, because interdependencies between incoming token offers can be neglected. Figure 5.20 a) shows a problematic scenario in case inconsistent offers would be allowed: the selection would fail if token ● were chosen for the left input pin of action B, because it would invalidate the conflicting offer on the other input pin. Backtracking would have to be used for a successful combination (● on the left and ● on the right) to be found, which makes the algorithm inefficient and harder to comprehend. The same goes for join nodes, if

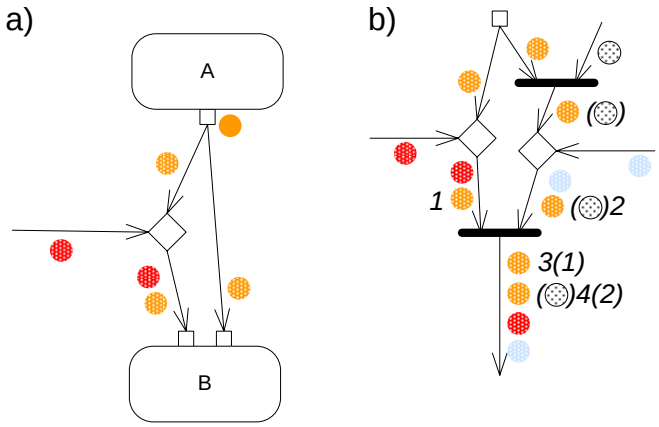

Figure 5.20: Problematic cases if inconsistent offers were allowed

inconsistent offers were allowed. Backtracking would have to be used to determine whether the join condition is met. Even worse, additional dependencies would arise between token offers outgoing from the join node, as shown in b). If ● were not present, only ● and offer 4 of ● could flow and offer 3 must be excluded. If, however, ● is present at the join node, no such dependency would exist.

Inappropriate order of selections. There is no order on token offers, meaning that no offer is preferred over another. In Figure 5.21, offers 1 and 2 may be chosen, leading to buffering of offer 3, because it is blocked by the upper bound of the central buffer node. Intuitively, one would rather select offers 2 and 3, because they stem from the same base token. To overcome this dilemma, the selection algorithm would have to act "globally", incorporating selections on other nodes, which would further complicate the computation.

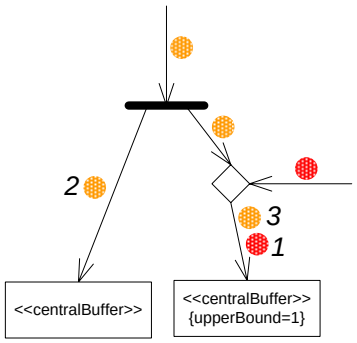

Figure 5.21: No order of token offers

5.8 Executing transitions

After computing and selecting relevant offers (see Section 5.7), the actual transitions can be executed. This results in interrupting regions and in moving base tokens from source nodes to target nodes. The following ExecuteTransition macro performs these steps:

ExecuteTransition ≡
    InterruptRegions
    seq
    MoveTokensAndCreateExecutions
    seq
    DetermineActivityTermination

The first step is to interrupt regions. The specification [UML05, p. 367] states, that

> "When a token leaves an interruptible region via edges designated by the region as interrupting edges, all tokens and behaviors in the region are terminated."

The following macro performs the interrupt:

InterruptRegions ≡
    foreach r **in** *regionsToInterrupt*
        forall n **with** $n \in (\mathit{ControlFlowSource} \cap r.\mathit{containedNode})$
            forall e **with** $e \in n.\mathit{outgoing}$
                $\mathit{controlTokens}(e) := [\,]$
        forall n **with** $n \in (\mathit{FlowDestination} \cap r.\mathit{containedNode})$
            forall e **with** $e \in n.\mathit{incoming}$
                $\mathit{buffer}(e) := \emptyset$
        forall n **with** $n \in (\mathit{ObjectNode} \cap r.\mathit{containedNode})$
            $\mathit{dataTokens}(n) := [\,]$
        foreach n **in** $(\mathit{Action} \cap r.\mathit{containedNode})$
            AbortAllActionExecutions(n, Self) ($\rightarrow$ 5.6.4)
            RemovePendingActionEvents(n)

Regions to be aborted have been determined by RemoveFlowsInInterruptedRegions 5.7.6 and stored in *regionsToInterrupt*. All control tokens, data tokens, and buffered offers are removed. Additionally, all action executions of all actions which are contained in such regions, are aborted. Pending "Termination"- and "Enable"-events for those actions must also be deleted from the event queue of the current activity execution. Otherwise, the later processing of those events can lead to the creation of new (but obsolete) tokens or executions.

RemovePendingActionEvents : *Action* → *Void*
RemovePendingActionEvents(n) ≡
    remove *pendingActionEvents*(n) **from** *eventQueue*(*Self*)

pendingActionEvents : *Action* → *ControllerEvent**
pendingActionEvents(n) =$_{def}$ [$ev \in$ *eventQueue*(*Self*) | $ev \in$ (*ActionTerminationEvent* ∪*ActionEnableEvent*) ∧ ev.*execution*.*node* = n]

The next step of ExecuteTransition is to remove base tokens and to store them in target nodes. In addition to that, new action executions are created for flow destinations being actions. Our selection algorithm already ensured, that [UML05, p. 302]

> "An action execution is created when all its object flow and control flow prerequisites have been satisfied (implicit join). Exceptions to this are listed below. The flow prerequisite is satisfied when all of the input pins are offered tokens and accept them all at once, precluding them from being consumed by any other actions."

In addition to accepting all data tokens for (i.e. moving them to) input pins, we also remove control tokens from source edges, as discussed in Section 4.3.7. We have to remove all control tokens, since [UML05, p. 302]

> "If multiple control tokens are available on a single edge, they are all consumed.."

Note that tokens are only removed if they do not stem from a buffered offer (ensured by ¬o.*buffered*). Otherwise, this would lead to removal of control tokens appearing later at the source nodes, which is not wanted. Since a base token can have flows to multiple targets (see fork node semantics), copies of these tokens are created by using the Clone macro, before storing them in targets. Instead of cloning at fork nodes, we decided to clone at the time of transition execution, because this avoids another field for cloned tokens of *TokenOffer*. The original base tokens would have to be included anyway. Each data token therefore is only held at exactly one object node and never moved (uncloned) to other destinations.

```
MoveTokensAndCreateExecutions ≡
    local activationData : InputPin → DataToken*
    foreach (n, O) in tokenSelections
      if n ∈ Action then
        forall i with i ∈ n.input
          activationData(i) := [ ]
      seq
      foreach o in O
        if ¬o.buffered then
          forall e with e ∈ sourceEdges(o) ∧ e.source ∈ ControlFlowSource
            controlTokens(e) := [ ]
        if IsObjectFlowOffer(o) then
          let
            t = o.offeredToken
            t' = Clone(o.offeredToken)
          in
            if ¬o.buffered ∧ Source(t) ≠ undefined then
              dataTokens(Source(t)) := dataTokens(Source(t)) ⊘ [t]
            dataTokens(target(o)) := dataTokens(target(o)) ⊕ t'
            if target(o) ∈ InputPin then
              activationData(target(o)) := activationData(target(o)) ⊕ t'
      seq
      if n ∈ Action then
```

CreateActionExecution(n, *activationData*) ($\rightarrow$ 5.6.1)

Clone : *DataToken* $\rightarrow$ *DataToken*
Clone(t) $\equiv$
let
$t' = new(DataToken)$
in
$t'.value := t.value$
result $:= t'$

A token selection can lead to the creation of a new action execution. All relevant input data are collected in *activationData* and handed on to CreateActionExection.

The last step of ExecuteTransition is to determine whether an "Activity Termination"-event has to be generated. The specification [UML05, p. 320] includes the following:

> "A token reaching an activity final node terminates the activity [...] Any object nodes declared as outputs are passed out of the containing activity, using the null token for object nodes that have nothing in them."

The following macros handle this case.

DetermineActivityTermination $\equiv$
if $\exists(n, \_) \in tokenSelections : n \in (ActivityFinalNode \cap Self.activity.node)$ **then**
forall n **with** $n \in (ActivityOutputParameters \cap Self.activity.node) \ \wedge \ |dataTokens(n)| = 0$
$dataTokens(n) := dataTokens(n) \oplus$ [NewNullToken]
EnqueueUniqueEvent(*ActivityTerminationEvent*) ($\rightarrow$ 5.5.1.1)

NewNullToken : *DataToken*
NewNullToken $\equiv$
let
$t = new(DataToken)$
in
$t.value := undefined$
result $:= t$

5.9 Discussion

This section discusses our ASM semantics for UML 2 activity diagrams. First, important deviations from the specification and their reasons are described. Related work concerning semantics of activity diagrams is presented afterwards. Finally, extensions, experiences and concluding remarks are given.

5.9.1 Deviations from the specification

One of the most apparent deviations from the official UML 2 specification refers to messaging. Although we provide a comprehensive execution environment for activity diagrams, we do not want to give a broad formalization of the UML 2. Thus, we ignore the UML 2 "Common Behaviors" and other basic packages. Normally, each behaviored object manages its own event queue, as described in [UML05, p. 418]:

> "The behavior executes within its context object, independently of and concurrently with any existing behavior executions. The object that is the context of the behavior manages the input pool holding the event occurrences to which a behavior may respond ... As an

object may have a number of behaviors associated, all these behaviors may access the same input pool. The object ensures that each event occurrence on the input pool is consumed by only one behavior."

We use event queues for activity executions, which handle their corresponding action, activity and offer-computation events. Event types as defined by the UML 2 are not used. As we do not store signal events at context objects, we need to impart the signal distribution tag [Sar05], to be able to route signals to nested activity executions. There is no competition among event occurrences when there are multiple activity executions for one context object. Nevertheless, it is easy to introduce additional queues for each context object, for handling "Signal Receive" events. It would then be straight forward to rewrite the rules dealing with signal handling to reflect the intention (namely, event competition among executions) of the UML 2 specification.

Another important difference relates to the creation and enabling of action executions. In the specification, control tokens are kept until the execution is enabled, although data tokens have already been moved to the target input pins of the action. Since control tokens are reserved for the execution anyway, we already remove these tokens upon creation of the execution (see Section 4.3.7). In our tool, this is shown by moving the control tokens to the edges at the destination nodes.

We also do not buffer tokens at fork nodes, as discussed in Section 4.3.8. The reason is, that changing guards can confuse the modeler when debugging and executing. In addition, the intuitive semantics of control nodes acting solely as "traffic switches" is softened. We, therefore, store tokens that are refused directly at the edges of the destination node, which avoids these problems. Our solution implies that tokens are not buffered if there is a join node on the path which does not forward offers due to a missing join condition (for an example, see [Hau05]). This is because we allow buffering only if the whole path towards the destination node is "open", in contrast to [Hau05]. Since the term "target" is not clearly defined in the specification, both views are permitted.

A minor deviation relates to triggers for accept event actions. For mere practicability we allow only a single trigger to be defined. When modeling accept event actions graphically with a tool (see chapter 6), multiple triggers cannot be modeled or would confuse the developer.

5.9.2 Related Work

We basically discuss only UML 2.0 related work in the following, because of activity diagrams having been completely redefined in the current version of the specification. UML 1.x activity diagrams are special kinds of state charts: They inherit the semantics but provide only a special notation. The semantics of UML 1.x activity diagrams is, for example, treated by Eshuis and Wieringa, who provide a formalization for the execution of activity diagrams for workflow modeling [EW01, Esh02, EW02, EW04]. For a comprehensive overview of work on UML 1.x semantics, see [Stö04c] and [Hau05].

Börger et al define an ASM semantics for UML 1.x activity diagrams, but only present excerpts in [BCR00a]. We use this work as an inspiration, but our approach is completely different, in that we use agents for action and activity executions rather than for sub-diagrams. We provide a flow computation algorithm and a more comprehensive treatment of activities and actions as a whole, which is demanded by the raised complexity of the UML 2 specification. Other work which uses Abstract State Machines for specifying the semantics of other types of UML diagrams, includes, for example [BCR00b, Jür02, JEJ02] which formalize UML state charts.

Ober [Obe03] proposes a general approach to define the UML semantics in terms of ASMs. She describes, for example, the semantics of classifiers, associations, operations and actions. An automated mapping of the UML meta model to ASM functions is suggested as the basis for a formalization. We also use agents to implement the execution of actions, but do not provide a comprehensive semantics of UML basic elements. Besides, our mapping to ASMs is not performed automatically. Although Ober defines a semantics for UML 1.4 [UML01], it could also be adapted to UML 2.0, to serve as a basis for behavior diagrams, such as activity diagrams or state charts.

The work of [BFGS05] and [SB04] on the ASM semantics for C# can be used to obtain a better and more complete formalization for call operation actions. We leave the integration of a concrete programming language open in our mapping to ASMs, by using an abstract "ExecCode" macro.

However, we provide such an integration in our tool (see Section 6). Actions are implemented by C# code, which is invoked by reflection.

The official documents for the Specification and Description Language (SDL) [ITU02, ITU00a, ITU00b, ITU00c], served as a starting point for our formalization. We mainly use their syntax for ASM domains and function classifications in our work. The SDL uses a compilation approach [EGGP00], as opposed to our interpreter approach. Discussion on how the signal handling mechanisms of SDL compare to the UML signals can be found in our other work [Sar05].

Regarding UML 2.0 activity diagrams, Störrle [Stö04b,Stö04c,Stö04a,Stö05] proposes a mapping to Petri-nets, which is manifest, because the UML specification envisions a "Petri-like semantics" for activity diagrams [UML05, p. 314]. Different variants of Petri-nets are used, e.g., colored Petri-nets for data flow, and procedural Petri-nets for activities. The treatment of join nodes having mixed object and control flows is, however, neither discussed nor obvious. The development culminates in [SH05] concluding that Petri-nets might, after all, not be appropriate for formalizing activity diagrams. Especially mapping advanced concepts, such as interruptible activity regions, is found not to be intuitive. Moreover, the lack of a unified Petri-net formalism, integrating the different variants used to map different concepts, is observed. Ensuring the traverse-to-completion semantics is identified as another problem. The specification demands that the whole path from the source node to the destination be traversed at once. Since the mapping of control nodes to Petri-nets results in having multiple intermediate places, traverse-to-completion is not given. Störrle proposes to use yet another variant of Petri-nets, called "Zero-safe nets" [BM97], for this purpose.

Barros et al [BG03] also translate to Petri-nets. They omit mappings for the basic elements, such as control nodes and edges and focus on call behavior actions with parameters, which they convert to activities. They also do not take activity creation and destruction, signals, and token termination into account.

Vitolins and Kalnins [VK05] present an algorithm for computing the token flow, proposing a forward and backward search by using so-called "push" and "pull" engines. Several far-reaching restrictions are, however, imposed on activity diagrams. Decision nodes must have mutually exclusive guards, and object nodes must not have any outgoing concurrent edges. This simplifies their algorithm, since they do not have to pull all input tokens in one atomic step – traverse-to-completion is thus not observed. Fork and join nodes must not be on the same path between two actions. Tokens resulting from join nodes are grouped, which is neither excluded nor stated in the specification. Additionally, guards can only reference token values but not attributes of the context class. Central buffer nodes are not supported, and pins can have no upper bounds. Only call behavior actions are taken into account.

Hausmann [Hau05] formalizes activity diagrams using "Dynamic Meta Modeling", where graph transformation rules operate on an instance of the UML meta model. The transformation engine is responsible for resolving the nondeterminism occurring at competing edges of object and decision nodes. This renders the approach too inefficient to serve as a basis for tool support. The semantics of a large part of activity diagrams is described in great detail and problems of the UML specification are discussed. Apart from this, several restrictions apply also to this work. Only one offer is allowed per edge, and – as a consequence – when different data tokens are offered to a join node, only one of them is forwarded. Guards and interruptible activity regions are not supported.

Bock and Gruninger [BG04, BG05] use the Process Specification Language (PSL) to define the semantics of activity diagrams as constraints on runtime sequences of behavior execution. The term "occurrence" is used as a substitute for "execution". Only simple diagrams with call actions are discussed.

The ongoing UML Semantics Project [UML06] aims at formalizing a subset of UML by providing "a strong foundation for the definition of a UML virtual machine that is capable of executing UML 2.0 models". The Modelware Project [Mod05, Hea06] implements a tool capable of simulating basic activity diagrams, but only with control flows. Currently, no formalization of the algorithms behind their execution engine is available.

All previously discussed work for UML 2.0 focusses on the preliminary versions of the UML specification [UML03, UML04]. Besides being based on the final version [UML05], our formalization shows how to deal with the restrictions mentioned before. To the best of our knowledge, there

is no other work that includes a comprehensive treatment of concurrent activities and discusses the implementation of a variety of different types of actions, as shown in Section 5.6. Due to the operational nature of ASM rules, an implementation can be derived with relative ease. Moreover, none of the works discussed so far, and none that we know of, handles the problems presented in Section 4.4.4 related to interruptible activity regions, including incoming flows, multiple interrupting edges, and nested regions. The useful feature of *lower* and *upper* multiplicity bounds on pins, which enable multiple data tokens to be consumed at once by one input pin, is also not treated elsewhere. Like [Hau05] and [Obe03], we rely heavily on terms of the UML specification, as far as possible. This especially includes the direct usage of the UML meta model for the interpreter and the concepts of "activity execution", "action execution", "offers", and the notion of "creation" and "enabling" of executions. The ASM rules are traced back to requirements present in or absent from the UML specification.

5.9.3 Possible extensions and further work

A tool implementation has been derived from the ASM specification (see chapter 6). It has, however, not been generated automatically by an ASM tool. Although the partly operational style of our ASM specification leads to a largely straightforward implementation, it cannot be assured that the tool implements the ASM rules in a correct way. We, therefore, currently investigate the usage of AsmL [GRS04, BGN+03, BS01] and its successor, Spec Explorer [CGN+05], which enable the direct execution of the ASM rules [Fre06]. This way, syntactic and more or less trivial semantic errors have already been revealed. Examples of severe errors found by [Fre06], are:

- a wrong behavior when keeping accept event actions active, due to an issue with the *IsActive* predicate
- an unwanted duplication of data tokens in output pins of call behavior actions, due to a problem in the HandleActivityTerminationEvent macro
- a missing **seq**-keyword in AllControlNodeSuccessors led to a false computation of the return value

Another goal is to generate test cases, at least for the flow computation rules, to guarantee their proper working in our implementation.

Other interesting areas to explore are test case generation from activity diagrams (see, for example, [WYY+04]) or model checking [dMGMP02, Win01, CW00]. A translator from ASMs to PVS, including a comprehensive example for multi-agent-ASMs, is discussed in [GR00], which could serve as a starting point.

Extending the supported subset of UML 2 activity diagrams should also be considered. Important elements to examine are, e.g. weight specifications for activity edges. There are contradictory statements regarding the semantics of "weight". In [UML05, p. 315], "weight" denotes the

> "Number of objects consumed from the source node on each traversals"

On the next page it says, that

> "The weight attribute dictates the minimum number of tokens that must traverse the edge at the same time. It is a value specification evaluated every time a new token becomes available at the source. It must evaluate to a positive LiteralUnlimitedNatural, and may be a constant. When the minimum number of tokens are offered, all the tokens at the source are offered to the target all at once."

Due to these contradictions, the exact semantics has first to be clarified by the OMG. This also includes constraints on permissible weight specifications of consecutive edges.

Another issue relates to reentrant behaviors, where it is not clear when a delayed call action should be invoked (data tokens collect at input pins as stated by [UML05, p. 302]). This is especially important when considering *lower* and *upper* specifications, which are not even discussed in the semantics of "Action" on page 301.

Order of tokens is another problematic area. According to [UML05, p. 369] "Tokens are offered on the outgoing edge in the same order they were offered to the join". It is unclear what "order" means and there is no statement on order of tokens at other places in the specification.

5.9.4 Concluding Remarks

In this section, we define the execution semantics of a subset of UML 2 activity diagrams by using Abstract State Machines. The formalization provides insight into problems with the UML specification, and their solutions. The resulting rules can be traced back to requirements present in or absent from the UML specification. Semantic variation points have been used for missing or controversial parts of UML activity diagrams, such as interruptible activity regions and signals.

The rigor of ASMs forces us to precisely define the semantics of UML activity diagram elements. Thus, imprecise terms such as "offer", "traversal" or "action execution" become more comprehensible. There are, however, parts of the specification, the meaning of which can only be guessed. The OMG should, therefore, consider the definition of a semantics for the complete UML, a task that has – to some extend – just begun by the UML semantics project [UML06]. Without a formal semantics, the UML specification is of only limited use. This is especially true for the behavior part of the UML. The specification also does not have a direct model of runtime execution yet, although terms like "execution" are used throughout the document. It would certainly be useful to incorporate runtime concepts into the UML meta model to clarify these terms.

Besides discussing the UML semantics, the operational style of the ASM rules provides a good starting point for tool implementations. Our tool, which implements our proposed approach to Model-Driven Development, is presented in the next chapter.

Chapter 6

Tool Support

This section shows some aspects of our tool implementation, comprising the interpreter runtime and the "ActiveChartsIDE". The IDE is used for diagram import, animation and debugging of activity diagrams. Detailed information on the usage of our tool is given in [Sar06]. Further information about our IDE is contained in [SGK+05]. The information in this chapter is partly taken from these publications.

Section 6.1 describes the architecture of our implementation. A short survey of how the IDE is used for Model-Driven Development is given in Section 6.2.

6.1 Architecture

Figure 6.1 shows an overview of our tool approach. We build on Microsoft Visio 2003 [Bia04, WE04] for modeling and animation and use Microsoft C# [NEG05] as the implementation language for the software under development. As is common in modern software development processes, the static structure of a system is modeled using UML 2 class diagrams. These models – drawn in Microsoft Visio 2003 – are translated into C# code by a generator (shown as the "Class Generator"-tool in the Figure). The generated code implements all attributes and associations shown in the diagram, including code to handle modifications (addition and removal of objects) of those relationships. The implementation involves that when updating one end of an association, the opposite end is automatically modified.

Since partial classes [Mic05] are used, additional C# code adding methods or other attributes can (and should) be written in separate files. The ordinary Microsoft C# compiler merges all matching class definitions when compiling the files. This leads to easier development cycles, because modifications of the static structure and therefore regeneration of its code leaves custom C# code untouched.

Application control flow is modeled using UML 2 activity diagrams (see "Dynamics" in the Figure). Therefore, each class that should have its own behavior has an associated activity, describing its functionality. UML 2 call operation actions (see Section 5.6.5.2) are used to invoke custom C# code written by the developer. In Figure 6.1 this is indicated by an action named "DoSomething" and its associated method declaration "`void DoSomething()`" in the lower left.

To connect classes to activities we again make use of UML tags. Activities are also drawn in Visio 2003 and translated into a XML representation by our tool. When the compiled program is finally executed, a runtime component ("ActiveCharts Runtime", implemented as a dynamic link library) reads the model file and executes the activities when needed. During execution, custom code is called if an UML call operation action is reached in an activity. To integrate import, visualization and debugging of activity execution, the "ActiveChartsIIDE" has been developed, which is discussed in the following section.

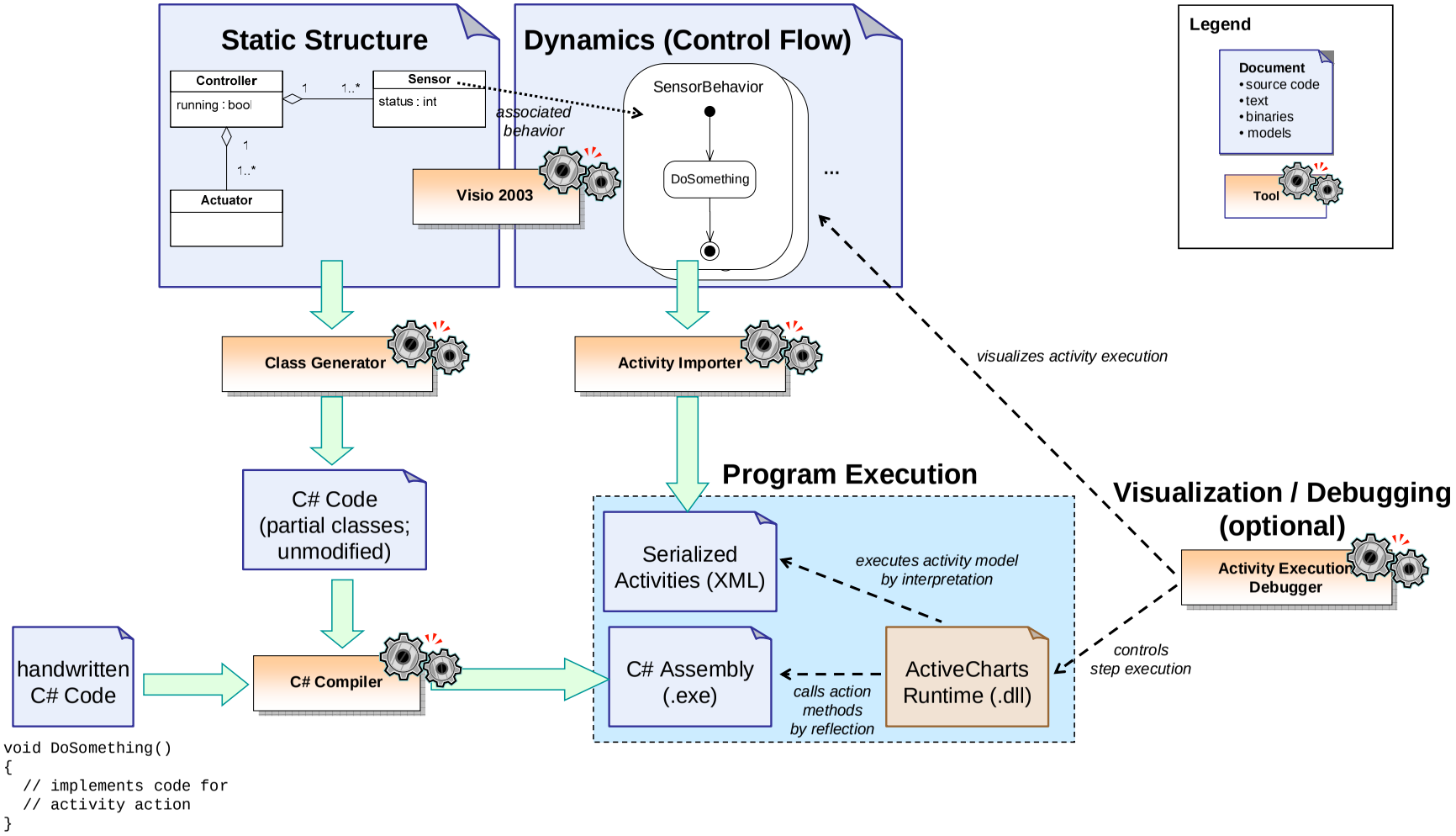

Figure 6.1: ActiveCharts Architecture

6.2 Working with the ActiveChartsIDE

The "ActiveChartsIDE" (see Figure 6.2) is used for class and activity diagram import and for visualizing and debugging activity executions [Ges05]. When started, executions can be controlled and shown in the diagram previously drawn. Values of data tokens flowing between actions can also be examined.

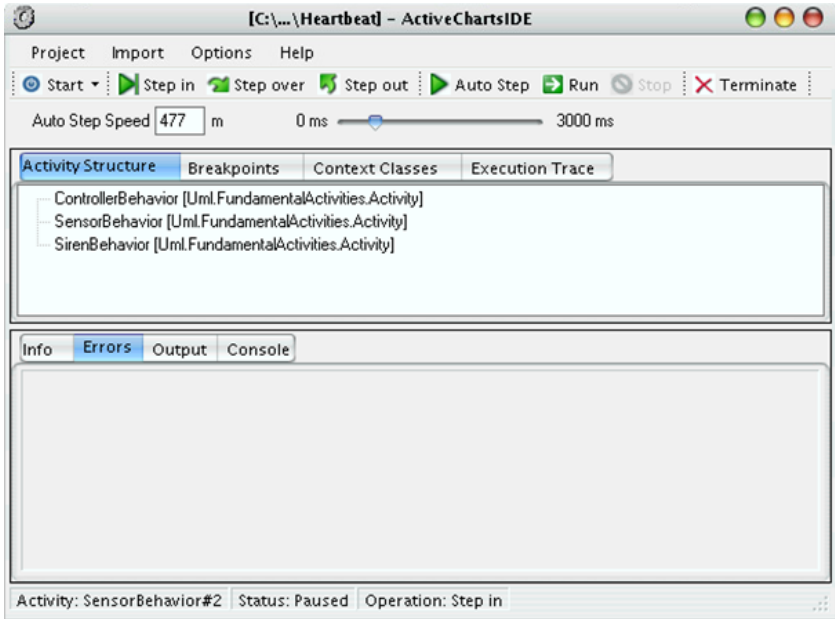

Figure 6.2: ActiveChartsIDE

Adjustments had to be made in the interpreter and tool to be convenient for the developer. According to the UML 2 meta model, call operation actions determine the operation to call, and the target object to use, by the *operation* resp. *target* relationships (see [UML05, p. 239]). It is not clear how to provide this information in a graphical way. We, therefore, assume the *target* to be the context object of the current activity execution and determine the operation by matching the name of the call operation action with the methods of the context class. Operation parameters are matched with input pins by their name, and type names must be annotated at the pins. A similar principle is applied for signals, where the signal type is derived from the name of the send signal or accept signal action. For wait time actions, valid time specifications have been defined, and *isRelative* is computed automatically. A graphical symbol for broadcast signal actions, which is not provided in the UML specification, has also been introduced by us. See our tutorial on "ActiveCharts" [Sar06] for details on the usage of these actions.

Besides, the UML specification [UML05] neither defines a suitable language for guards, nor does it give any hints to whether and how token values and attributes and relationships of the context can be referenced. For practical purposes, we allow valid C# expressions as guards, with elements of the context prefixed by "$". A single "$" is used to reference the value of the current data token. Thus, we can create complex guard expressions.

We envision the following development cycle for "ActiveCharts" projects.

1. A **new project** is created. All generated and handwritten code, as well as configuration and project information is stored in the specified location.

2. **Class diagrams** are created which model the static structure of a system. For hints on modeling, see [Lar05, Eva03]. These diagrams are then imported by the IDE, and code is

generated. If UML "behavior" tags refer to associated activities, a hint is given to create and import the respective activity diagrams.

3. **Activity diagrams** are created for the behavioral classes and imported. Various syntax and semantic checks are performed [Ges05]. The activity diagrams are serialized in an XML file and stored at the project location. ActiveChartsIDE automatically generates **interfaces** from action invocations in activity diagrams. These methods are added to interfaces the corresponding context class must implement.

4. The developer is, therefore, forced to **implement all action methods** before successfully compiling the project with a separate C# compiler. Input and output pins are used to build the formal parameter list and return type.

5. The **context objects have to be created**, and their attributes and relationships be set. This is normally done manually in the main method of the application. When creating the objects, associated activity executions are instantiated automatically and, therefore, their behavior is started.

6. The project can now be **compiled** using an external compiler or development environment, such as Microsoft Visual Studio .NET. The ActiveCharts runtime library and the serialized activity diagrams have to be included.

7. When **starting the project stand alone**, i.e. without using the ActiveChartsIDE, the diagrams are executed seamlessly and determine the control flow of the application.

8. To make use of the debugging possibilities of the ActiveChartsIDE, the compiled executable must first be loaded into the IDE. **The executable can then be started in debugging mode** without any adjustments to the application code. The IDE controls the step execution and visualizes each step by highlighting edges and showing tokens in new Visio diagrams, which represent the activity executions. Debugging functions include step-in, step-out, step-over and auto-step. Breakpoints can be set on actions, and edges and values of data tokens can be inspected. For an overview of the complete functionality, see [Sar06, SGK+05, Ges05]. A screenshot of the animation of our "Heartbeat" application (see appendix B) is shown in Figure 6.3.

9. **Changes** can be applied to the diagrams and code independently, leading to re-import and/or re-compilation of the application. Because of the strict separation of generated and compiled code, no user code is affected when code is generated for classes.

6.3 Discussion

In this section, we discuss related work, provide hints to possible extensions, experiences and make some concluding statements.

6.3.1 Related Work

A multitude of modeling and simulation tools currently exists on the market. The possibility of code generation from the used diagrams — mainly state charts [Mat05, ILo05] — is best available technology and widely used. Nevertheless, we decided to use UML 2 activity diagrams to model the behavior of an application for several reasons: first of all, it is a natural proceeding to expand use cases with activity diagrams. Many software development processes for object oriented applications, e.g. the Rational Unified Process (RUP) [IBM05], propose this. The explicit actions used in these diagrams can be seen as interactions between the different roles with the system. We, therefore, want to provide an experimental environment to verify whether our approach is beneficial for software development.

Second, activity diagrams model data flows in an explicit way, which can be used to model the data used in an application, whereas state charts do not offer this potentiality. Since, on the other hand, activity diagrams cannot model states, it is desirable to integrate these formalisms into a unified approach to Model-Driven Development.

We decided to use Microsoft Visio 2003 as a drawing tool for our diagrams for flexibility reasons. The use of other tools for drawing UML 2 compliant models is very restricted [Ber05]. It is impossible to draw interruptible regions or wait time actions for example, which, in our opinion, are fundamental elements. Visio provides great freedom in drawing as well as extensive possibilities for extending its drawing-shapes, although it requires additional syntactic and semantic checks in the ActiveChartsIDE. Further information on other tools and projects is given in our publication on the ActiveChartsIDE [SGK+05]. Details, on how our approach compares to the Model-Driven Architecture (MDA) [KWB03,MDA03,McN04] initiative of the Object Management Group (OMG), are given in our publication [SRKS05].

6.3.2 Possible Extensions

Besides enhancing the usability of our tool by integrating modeling, import, debugging and compilation into one integrated development environment, we want to provide further help in debugging tasks. One idea is to define test cases by creating initial guard values and assignments of tokens to object nodes. Different application scenarios could therefore be executed. It is also desirable to modify guard values at runtime, to be able to try different paths of execution.

Currently, the initial object configuration must be implemented by handwritten code. Objects are created, and attributes and relationships are set by user code, mostly on application startup. UML object diagrams [UML05] could be used to provide these initial objects. This is especially true for embedded systems, where objects often represent hardware, whose configuration does not change at runtime.

6.3.3 Experiences

Several case studies (see appendix B) have been modeled and tested with our implemented tool. These examples also serve as comprehensive test cases for the ASM rules given in chapter 5. In addition to that, practical courses have been conducted with students, who contributed to the maturity of the execution semantics by creating and testing small and large projects. Moreover, insights on the appropriateness of our approach to Model-Driven Development have been gained. Although not examined in an experimental setting, it should be noted that the feedback on our approach and tool has been entirely positive.

6.3.4 Concluding Remarks

Our tool enables the developer to experiment with UML 2 activity diagrams in Model-Driven Development. It implements the semantics defined by our ASM rules in chapter 5, though some adjustments have been made to enhance its usability for effective software development. Further experiments can now be made to measure the implication on quality of software and its documentation, as well as on the efficiency of software development projects.

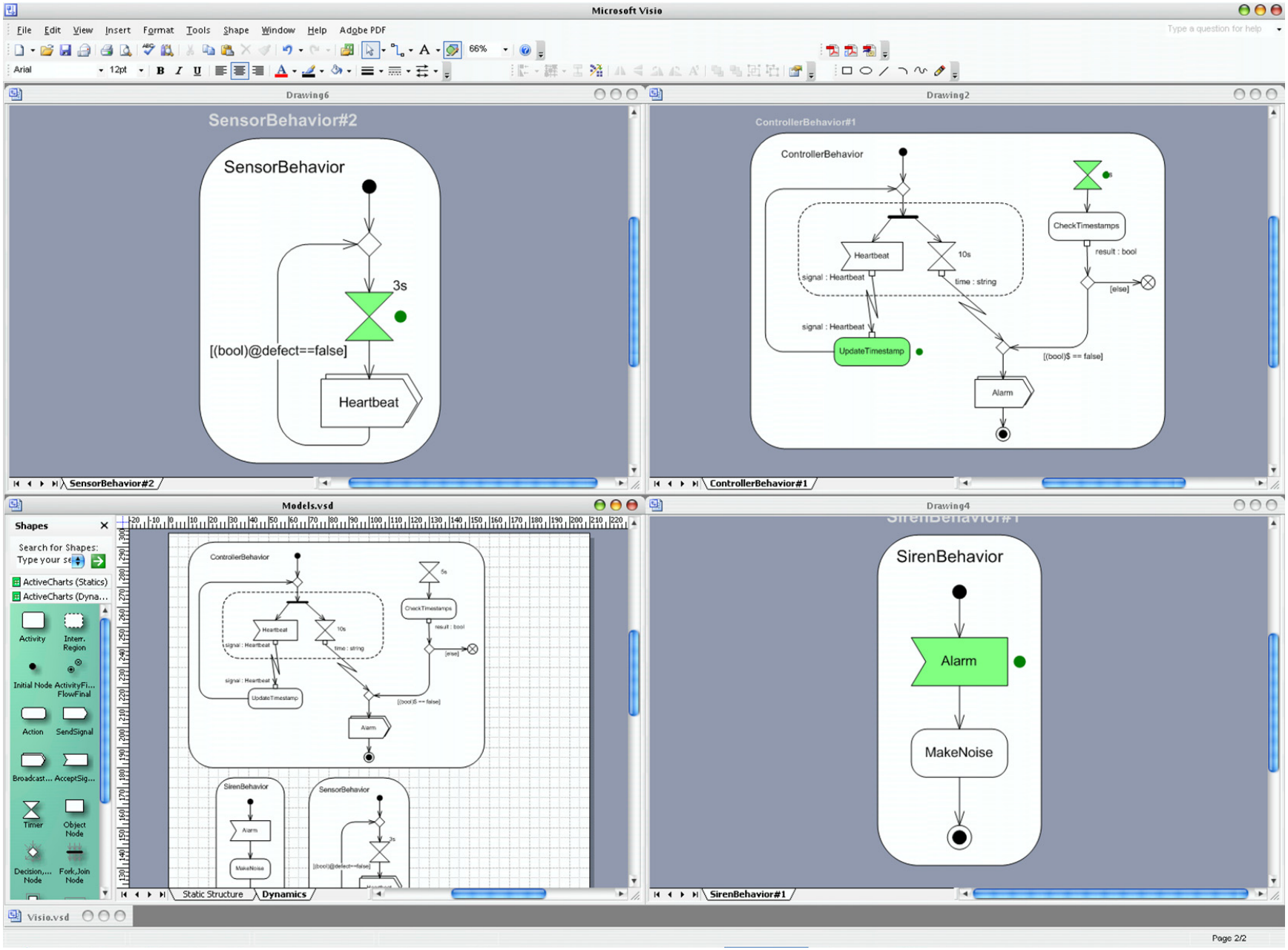

Figure 6.3: Execution of the “Heartbeat” example

Chapter 7

Summary

The aim of this thesis is to provide the basis for Model-Driven Development with UML 2 activity diagrams. To this end, we define a semantics and provide a framework and tool implementation for executing the diagrams.

7.1 Contributions

The most significant contributions of this thesis are the following:

Complete ASM Semantics. A complete ASM semantics has been defined for the selected subset of UML 2 activity diagrams (see chapter 5). Requirements of the rules are traced back to information present or absent in the UML specification. The UML 2 meta model for activity diagrams is translated into ASM domains and functions, and serves as a starting point for the interpreter. Asynchronous Multi-Agent ASMs are used to model multiple concurrent action and activity executions, which communicate by using events. We discuss the life-cycle of activities and actions. An execution semantics for CallBehaviorAction, CallOperationAction, AcceptEventAction, SendSignalAction, and BroadcastSignalAction is presented. An efficient flow computation algorithm for the propagation and selection of token offers, which will also be published in [SG06], is proposed.

The ASM formalization provides a good starting point for implementations, because of the operational nature of the ASM rules. It is also *modular*, in that new types of actions can be easily added. As long as no fundamental adjustments are made in the UML specification, such as changing the different states of actions as specified in [UML05, p. 301] ("created", "enabled", etc.), places for *modifications* can be located straight forwardly. This is especially true for the flow computation algorithm which is independent from other parts of the execution rules. Different flow and selection algorithms can be substituted for the existing rules.

Discussion of UML semantics. A subset of UML 2 activity diagrams has been selected, and problems of the specification are discussed (see chapter 4). This includes

- the positioning of control tokens
- the overtaking of control tokens by data tokens
- multiple problems with interruptible activity regions and nested interruptible activity regions
- issues with accept event actions

Variation Points. Although we tried to stick as closely as possible to the specification (in our – subjective – view), modifications had to be made to make sense of the activity diagram semantics. Wherever possible, freedom is left to adjust the semantics by introducing variation points (see chapter 4). To integrate the configuration of semantics with the graphical models, UML tags are

used, which are a standard extension mechanism of the UML. Variation points have been defined for

- handling flows out of and into interruptible activity regions
- activation and re-activation of accept event actions in nested interruptible activity regions
- determination of the context object to use for call behavior actions
- signal targets and buffering

Our approach for configuring the semantics with UML tags, and the application of tags for signals has already been published in [Sar05].

Tool Support. To verify the working of the ASM rules and to provide an environment for experimenting with our approach, a prototype has been implemented (see chapter 6), called "ActiveChartsIDE". The IDE is capable of importing class and activity diagrams and transforming them into an XML representation, generating code out of class diagrams and animating and debugging the execution of activity diagrams. Microsoft Visio 2003 is used for modeling because it turned out to be the most flexible and extensible tool for us. The ActiveChartsIDE also integrates well in our proposed development process. No other tool provides a similar coverage of the UML 2 notation for activity diagrams. The interpreter implementation can also execute independently of our IDE, without any modifications of the developed software. When used for debugging, the IDE "connects" to the implemented system and controls its execution. Further discussion of our tool implementation can be found in [SGK+05]. A tutorial for our tool is given in [Sar06].

Proposal to Model-Driven Development with UML Activity Diagrams. We propose an approach to Model-Driven Development which integrates UML 2 activity and class diagrams (see chapter 1). The control flow of the application (i.e., the behavior) is modeled with UML 2 activity diagrams during analysis and design phases. These diagrams are seamlessly reused for the implementation by interpreting them at runtime. Together with generated code of the static structure out of UML 2 class diagrams, the explicit modeling of the application control flow should simplify the creation of applications and lead to a continuous development process from the analysis/design phase to implementation. Actions are implemented by user code and executed by the diagram interpreter. The degree of functionality described by models versus functionality described by code can be freely chosen by the developer, which should improve acceptance of modeling tasks. This is even more useful, as some tasks are almost impossible or at least too extensive to be modeled (e.g., GUI-extensions, complex computations, database access, etc.). Our approach to Model-Driven Development is presented in [SRKS05].

Ideas towards system evolution with activity diagrams, which are not further elaborated in this thesis, are presented in [SKRS05].

7.2 Outlook

Several suggestions for further research are given in Sections 5.7.9, 5.9.3 and 6.3.2. Areas of interest include the generation of test cases, model-checking and incorporating more UML elements. Experiments can now be carried out to measure the possible improvements of our approach on software development projects. We assume that explicitly modeled control flow leads to shorter development cycles and enhances design documentation. In addition, the impact of our approach on system evolution can be examined.

It is desirable to provide an integrated formalization of both the UML "fundamentals", such as the "Common Behaviors" package, and the actual diagrams, which build on these basics. One step in this direction is to incorporate the event types as defined by the specification and to assign additional event queues for UML-specific events to context objects rather than executions.

Also, a tighter integration between the ASM specification and the tool implementation should be established. Several efforts of our group currently target these fields of research.

Appendix A

Mathematical Conventions

Operator	Description
Misc	
$x.g$	function application, equivalent to $g(x)$
$t_1 \preceq t_2$	*true* if type t_1 is a subtype of or equal to type t_2
Set operators	
$\cup$, $\cap$, $\in$, $\subset$, $\subseteq$, $\setminus$, $\times$, $\emptyset$	meaning as usual
$\bigcup X$	union of all sets contained in the set of sets X
$\mathcal{P}(S)$	powerset of set S
$\{\ s \in S \mid P(s)\ \}$ or $\{\ F(s) \mid P(s)\ \}$	set comprehension, P is a predicate, F is a function which is evaluated on element s
$\|S\|$	number of elements in set S; has lower precedence than "x.g"
asList(S)	returns a list containing the elements of set S in a random order
List operators	
A^*	list type
$L_1 \uplus L_2$	concatenation of two lists
$\uplus X$	concatenation of all lists contained in the list of lists X
$L \oplus e$	add element e to list L, equivalent to $L \uplus [e]$
$L_1 \oslash L_2$	list difference, remove from L_1 all elements contained in L_2
$[\ l \in L \mid P(l)\ ]$ or $[\ F(l) \mid P(l)\ ]$	list comprehension, P is a predicate, F is a function which is evaluated on list element l
$l \in L$	element function
$[\ ]$	empty list
$\|L\|$	number of elements in list L; has lower precedence than "x.g"
elementAt(L, i)	gets the i-th element from list L
last(L)	gets the last element from list L, equivalent to elementAt(L,$\|L\|$)
indexOf(L, e)	returns the index of the first occurrence of element e in list L, indexing starts at 1

Appendix B

Case Studies

This section presents some case studies, which have been implemented and tested with our tool presented in chapter 6. Each case study is introduced by a short description. The class and activity diagrams, which implement the requirements, are then presented. Action implementations are only given for the alarm device, other actions are implemented similarly.

B.1 Alarm Device

Figure B.1 shows the static structure of a simple alarm device. Multiple sirens and sensors are associated with a controller class, whereof only one instance exists at any time. We want to design an alarm system which can detect broken sensors. Other functionality is omitted for brevity. We, therefore, define the following requirements:

- Each sensor sends "Heartbeat" signals to the controller every three seconds.
- The controller manages timestamps for each sensor and updates them upon receipt of "Heartbeat" signals.
- The controller checks all timestamps every 15 seconds. If at least one sensor is "late", an "Alarm" signal is sent to all sirens. The same is true, if no "Heartbeat" signal has been received from any sensor within 10 seconds.
- Each siren waits for "Alarm" signals, and is activated upon receipt.

Figure B.2 shows the activity diagrams for the alarm device. Each sensor can be disabled for testing purposes by setting its "defect" attribute to "true". If defect, a guard ensures that the sensor does not send any more "Heartbeat" signals to the controller.

The diagrams show three call operation actions ("MakeNoise", "UpdateTimestamp", and "CheckTimestamps") which have to be implemented with custom code. The code is shown in Figure B.3. Each call behavior action maps to a method of the corresponding context class of the behavior. "MakeNoise" simply shows a dialog-box to indicate that an alarm has been issued. "UpdateTimestamp" updates the timestamp of the sensor where a signal was received for. "CheckTimestamp" is called every 15 seconds to see if any sensor is "late".

Finally, Figure B.4 shows the code to instantiate the objects and associations of an alarm device with two sensors and one siren. Note that the diagrams and code shown are the only artifacts needed to obtain a running alarm device with the above stated requirements.

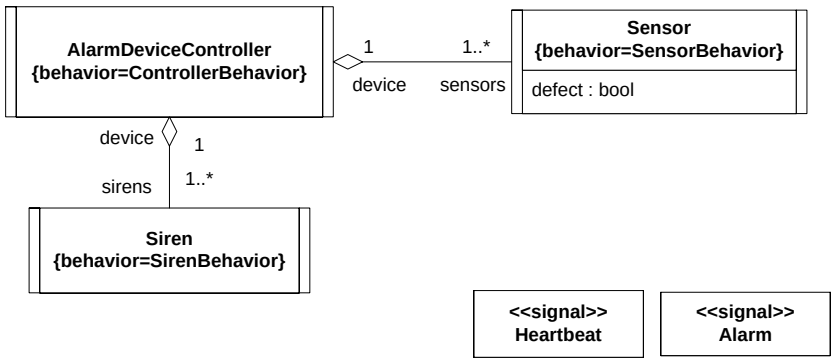

Figure B.1: Alarm device static structure

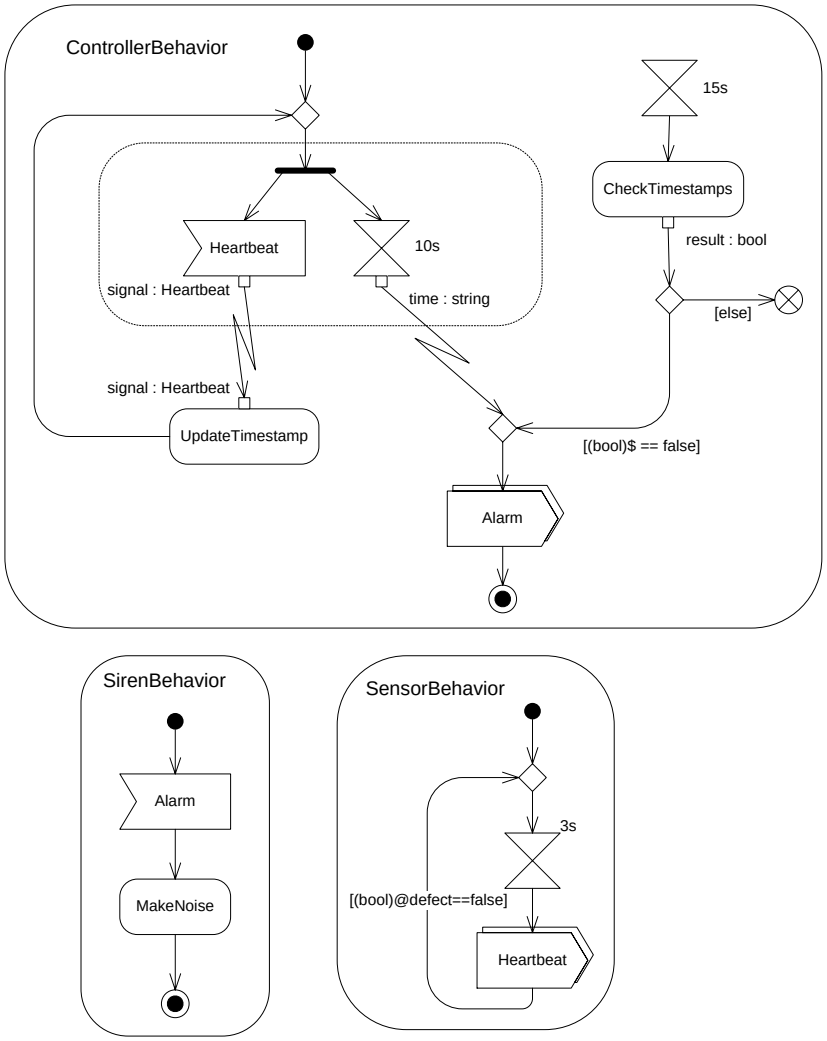

Figure B.2: Alarm device behaviors

```
public partial class AlarmDeviceController
{
    private Dictionary<Sensor, DateTime> LastBeat = new Dictionary<Sensor, DateTime>();

    public void UpdateTimestamp(Heartbeat signal)
    {
        DateTime now = ActiveChartsDateTime.Now;
        if (LastBeat.ContainsKey((Sensor)signal.source.context))
            LastBeat[(Sensor)signal.source.context] = now;
        else
            LastBeat.Add((Sensor)signal.source.context, now);
    }

    public bool CheckTimestamps()
    {
        DateTime now = ActiveChartsDateTime.Now;
        foreach (DateTime t in LastBeat.Values)
        {
            if ((now - t) > new TimeSpan(0, 0, 5))
                return false;
        }
        return true;
    }
}

public partial class Siren
{
    public void MakeNoise()
    {
        MessageBox.Show("Alarm", "Making Noise");
    }
}
```

Figure B.3: Alarm device action implementations

```
public void CreateObjects()
{
    AlarmDeviceController alarmDeviceController = new AlarmDeviceController();
    alarmDeviceController.sirens.Add(new Siren());
    alarmDeviceController.sensors.Add(new Sensor());
    alarmDeviceController.sensors.Add(new Sensor());
}
```

Figure B.4: Alarm device object setup

B.2 Molding Press

The second example is a molding press, which consists of a piston and two buttons. The piston is used to mold a work piece. The requirements are as follows:

- The piston starts moving downwards if its two buttons are pressed within 1 second. If the time span is greater, both buttons have to be released before starting again.
- When the piston moves downwards and any button is released until the "Point of no return" is reached (which is located at 3/4 of the total distance), the piston stops and moves upward again in its starting position. This is for safety reasons.
- When the "Point of no Return" has been reached, the piston continues moving downward even if any button is released after that.
- When the piston has reached the bottom position it stops and moves upward to its starting position.

The class- and activity diagrams for the molding press are given in figures B.5 and B.6, respectively. The behavior for the "PressController" class is defined by the "PressControllerBehavior" activity. For moving the piston down and up, two separate activities, "MoveDown" and "MoveUp" are introduced, which are invoked by using call behavior actions. Several UML tags must be attached to the diagrams to make them work properly:

- Buffering of signals must be disabled for the whole activity. Otherwise they would lead to incorrect behavior of the press (see variation point in Section 4.2).
- Since there are accept event actions without incoming edges in a nested interruptible activity region, "regionActivationPolicy" has to be set to "OnParentFlow" (see variation point in Section 4.4.4.4).
- "MoveDown" and "MoveUp" implement the actual piston movement by modifying the "position" attribute of the piston class (the code is not shown here). Since this attribute is naturally defined in the "Piston" class, the two activities must have this class as their context. To this end, the UML "context" tag is used to invoke the call behavior actions with another context class (see variation point in Section 4.4.1).

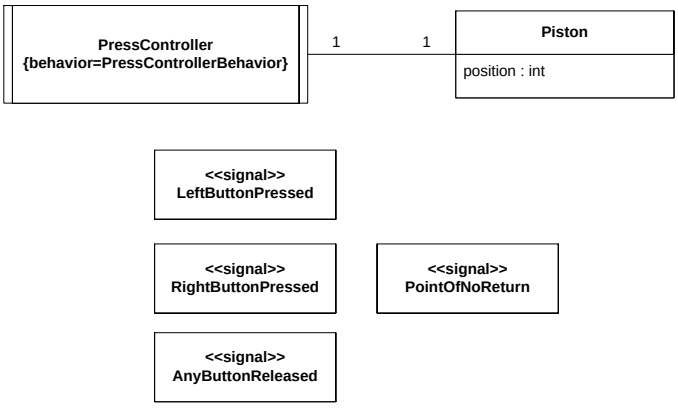

Figure B.5: Molding press static structure

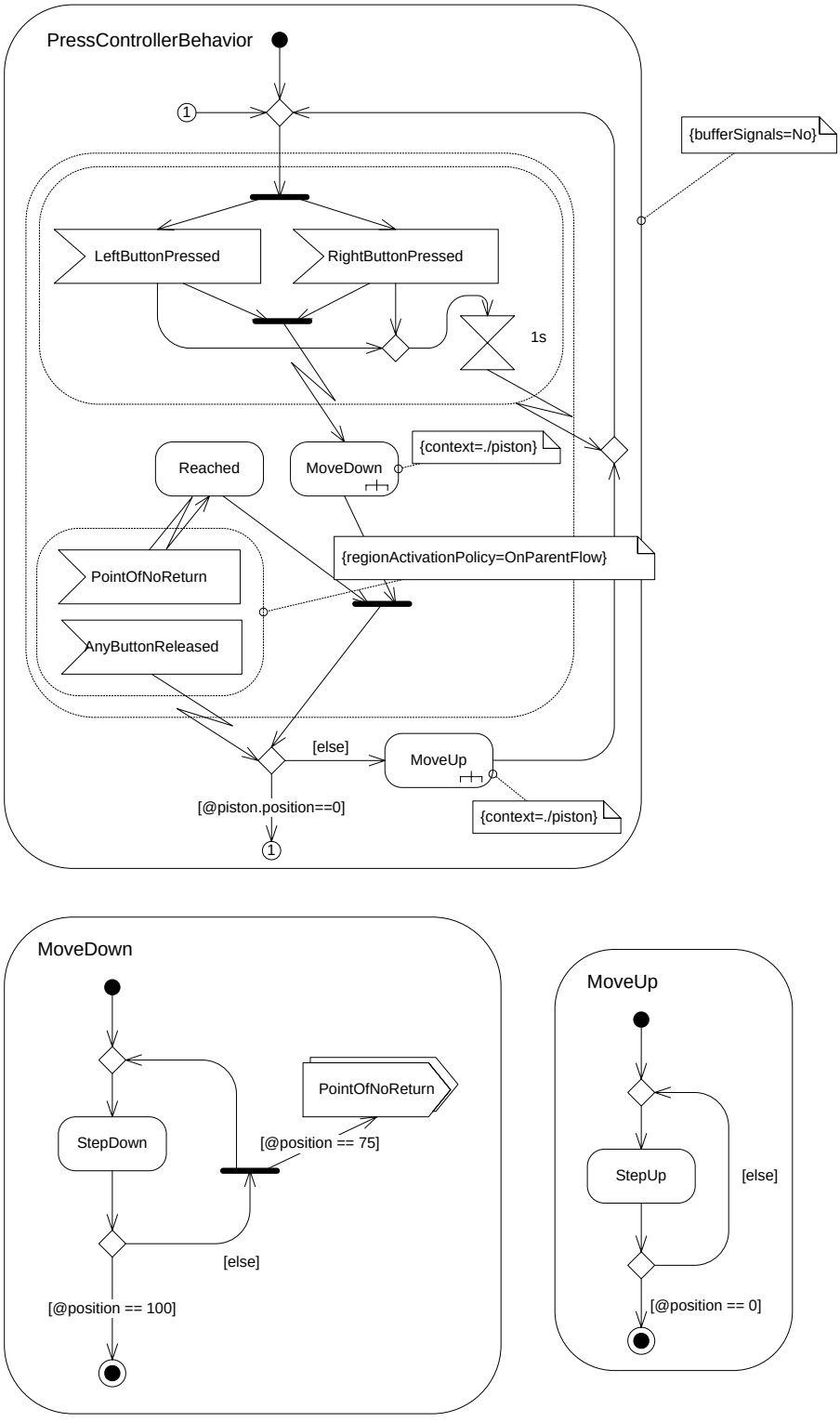

Figure B.6: Molding press behaviors

B.3 Microwave

The microwave was developed by students during a practical course. The static structure in Figure B.7 shows a microwave and its technical components. Use cases that have been defined include:

- Cook
- Defrost
- Timer functionality and clock
- Programming
- Ensuring safety properties, such as deactivating the microwave when opening the door, etc.

Figure B.8 shows a small part of the behavior of the microwave, where the "cooking"-behavior is modeled. When started, the microwave is activated ("EnableEverything"). This includes the magnetron, turntable, and lamp. The microwave can be stopped at any time by pressing "Stop" on the control panel, which is modeled with the "Stop" accept event action. Cooking has finished after the timer has elapsed. During cooking, the door can be opened, which pauses the microwave. If the door is not closed within 10 seconds, the microwave is stopped. Otherwise, cooking resumes.

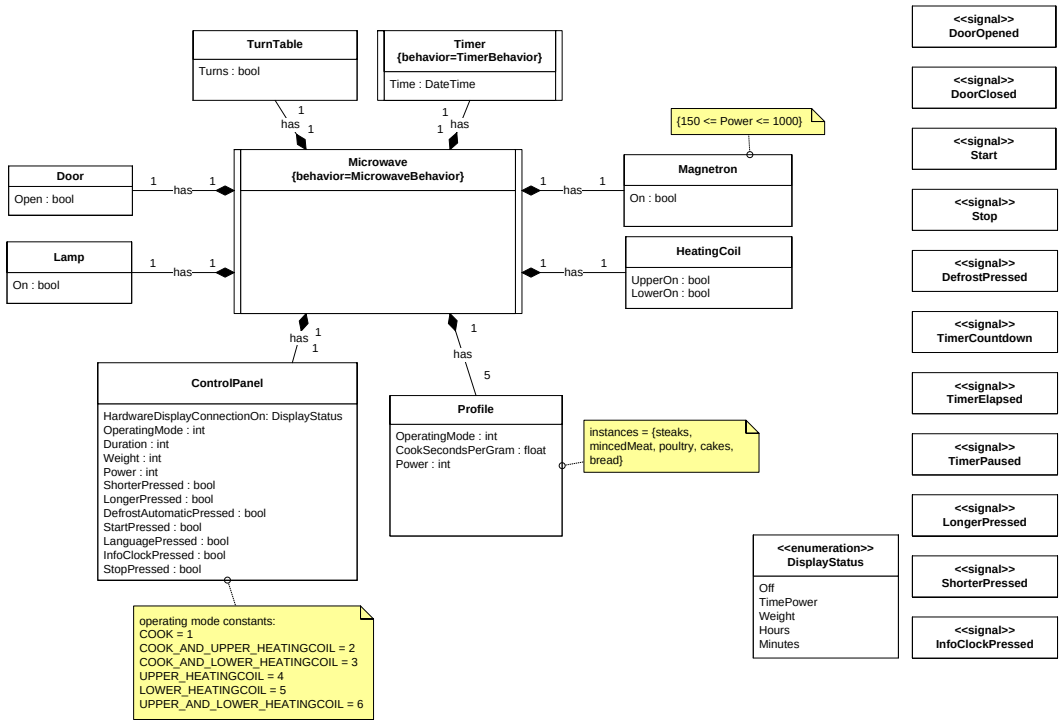

Figure B.7: Microwave static structure

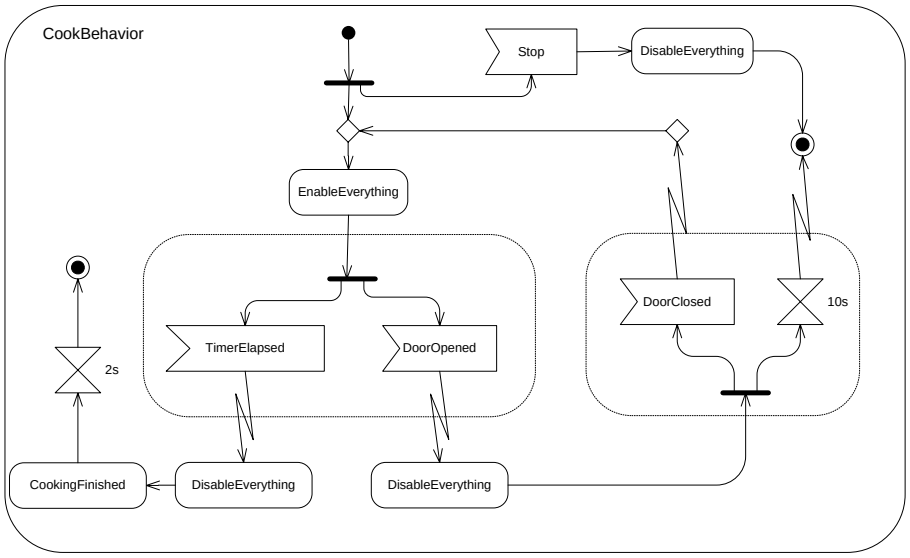

Figure B.8: Extract from microwave behavior

Zusammenfassung

Moderne Softwareentwicklungsprozesse propagieren die Verwendung grafischer Modelle in den Analyse- und Design-Phasen eines Projekts. Durch Abstraktion, Strukturierung und Kommunikation soll auf diese Weise ein besseres Verständnis des zu entwickelnden Systems erlangt werden. Des Weiteren wird die Produktdokumentation aufgewertet, da grafische Modelle einen besseren Überblick ermöglichen. Trotzdem werden diese Modelle in späteren Phasen der Entwicklung oft nicht mehr gepflegt, was zu einer Divergenz zwischen Implementierung und Dokumentation führt.

Die Modellgetriebene Softwareentwicklung versucht, dies durch direkte Verwendung der Modelle für die Implementierung zu verhindern. Bisherige Ansätze fokussieren jedoch zu sehr auf statische Strukturen, oder, falls das Verhalten einer Applikation überhaupt modelliert wird, auf die Verwendung von Zustandsautomaten. Anforderungen sind jedoch oft "aktionsorientiert", da Interaktionen zwischen Benutzer und System in der Regel durch Anwendungsfälle (Use Cases) beschrieben werden. Diese Use Cases werden oftmals zusätzlich in Form von UML Aktivitätsdiagrammen dargestellt.

Diese Arbeit befasst sich mit den Voraussetzungen für eine Verwendung von UML 2 Aktivitätsdiagrammen für die Modellgetriebene Softwareentwicklung. Diese Voraussetzungen liegen in der Definition einer formalen Semantik für UML 2 Aktivitätsdiagramme und der Bereitstellung eines Tools für deren Ausführung, Animation und Debugging.

Kapitel 1 bis 3 beschreiben den Kontext und die Grundlagen der Arbeit. In Kapitel 1 schlagen wir einen neuartigen Ansatz zur Modellgetriebenen Softwareentwicklung vor, der Klassen- und Aktivitätsdiagramme kombiniert. Unser Ziel ist es, die Softwareentwicklung zu verbessern, indem die grafischen Modelle aus den Analyse/Design-Phasen für die Implementierung verwendet werden. Aus Klassendiagrammen wird Code für Klassen und deren Attribute und Assoziationen erzeugt, der die statischen Struktur des Systems umsetzt. Das Verhalten von aktiven Klassen – und damit der Kontrollfluss – wird durch Aktivitätsdiagramme beschrieben, die zur Laufzeit interpretiert werden. Dadurch muss die in den Modellen enthaltene Logik nicht nochmals "von Hand" implementiert werden, was zu kürzeren Entwicklungszeiten führen soll. Da es nicht sinnvoll ist die gesamte Logik einer Anwendung grafisch zu modellieren, kann handgeschriebener Code mit den Aktivitätsdiagrammen integriert werden. Der erzeugte Klassencode wird dazu durch Methoden ergänzt, die an bestimmten Punkten im Diagramm-Ablauf aufgerufen werden. Das Verhältnis von Modellierung und Codierung kann vom Entwickler frei gewählt werden, was zu einer größeren Akzeptanz von Modellierungsansätzen führen soll.

Kapitel 2 befasst sich mit den Grundlagen zu UML 2 Aktivitätsdiagrammen. Insbesondere werden eine informelle Semantik für den Tokenfluss, sowie der für uns relevante Auszug aus dem UML 2 Metamodell beschrieben. In dieser Teilmenge enthalten sind alle für UML 2 Aktivitätsdiagramme relevanten Elemente wie z.B. Kontroll- und Datenflüsse, Aktivitätsaufrufe, Methodenaufrufe, Aktivitätsparameter, die Kommunikation mit Signalen sowie Unterbrechungsbereiche. Kapitel 3 gibt schließlich eine kurze Einführung in Abstract State Machines (ASMs), die wir für die Definition einer formalen Semantik für UML 2 Aktivitätsdiagramme verwenden.

Die folgenden Kapitel umfassen den Hauptteil der Arbeit. In Kapitel 4 diskutieren wir Probleme der UML 2 Spezifikation in Bezug auf Aktivitätsdiagramme. Da die offizielle UML 2 Spezifikation in rein textueller Form vorliegt, kommt es zu vielen Unklarheiten und Missverständnissen bei deren Auslegung. Wir diskutieren zunächst Probleme der Spezifikation, wie beispielsweise der Tatsache, dass Datentokens Kontrolltokens "überholen" können. Andere Probleme betreffen Signalempfänger, Signale, Unterbrechungsbereiche, die Positionierung von Kontrolltokens sowie die Pufferung von

Tokens an Verzweigungsknoten. Für UML 2 Elemente, bei denen mehrere Interpretationen oder Optionen sinnvoll sind, definieren wir "Semantische Variationspunkte", die dem Entwickler Freiheiten in der Modellierung lassen. Für die Konfiguration verwenden wir "UML tags", einen Standardmechanismus zur Erweiterung der UML. Konfigurierbar sind durch unsere Erweiterungen unter Anderem:

- die Behandlung von ein- und ausgehenden Flüssen in Unterbrechungsbereichen
- die Aktivierung und Re-Aktivierung von Signalempfängern in geschachtelten Unterbrechungsbereichen
- das zu verwendende Kontextobjekt bei einem Aktivitätsaufruf
- die Ziele von Signalen
- die Pufferung von Signalen

Ausgehend von der von uns geführten Diskussion über Probleme der UML 2 Spezifikation definieren wir in Kapitel 5 eine vollständige formale Semantik für die von uns untersuchte Teilmenge der UML 2 Aktivitätsdiagramme mittels Ansynchronous Multi-Agent ASMs. Das Verhalten der Diagramme wird durch einen Interpreter implementiert. Parallel ablaufende Aktivitäten und Aktionen verwenden jeweils eigene ASM Agenten, die mittels Ereignissen kommunizieren. Die wesentlichen Teile werden im Folgenden beschrieben.

- Zunächst wird gezeigt, wie das UML 2 Metamodell mit statischen ASM Domänen und Funktionen umgesetzt wird. Darauf bauen die folgenden Interpreterfunktionen auf.
- Es wird die Erzeugung und Terminierung von Aktivitätsinstanzen beschrieben. Jede Instanz wird in einer eigenen ASM ausgeführt, die eine Eventschleife implementiert. Events zur Kommunikation zwischen ASMs werden definiert.
- Die Semantik von Aktionen wird behandelt. Aktionen sind Elemente, die den Zustand des Systems verändern und nicht weiter verfeinert werden können. Wir unterstützen die UML Elemente CallBehaviorAction, CallOperationAction, AcceptEventAction, SendSignalAction, sowie BroadcastSignalAction.

  Wir führen verschiedenen Ausführungszustände von Aktionen ein und geben ASM Regeln für deren Implementierung an.
- Ein weiterer zentraler Teil der Formalisierung ist die Flussberechung für die Tokens in Aktivitätsdiagrammen. Dieser besteht aus einer Propagierungsphase, bei der Tokenangebote von Quellknoten zu Zielknoten verteilt werden, und einer Selektionsphase, bei der geeignete Tokenangebote an Zielknoten ausgewählt werden. Diese Auswahl führt schließlich zur Ausführung von Transitionen und damit zur Aktivierung von Aktionen. Wir unterstützen insbesondere auch *lower*- und *upper*-Angaben bei Aktionsparametern, was eine Konsumierung mehrere Tokens in einem einzelnen Aktivierungsschritt einer Aktion ermöglicht.

Bei der Formalisierung lehnen wir uns so eng wie möglich an den UML Standard an und verweisen jeweils auf die dort gemachten Aussagen. Da der Aufbau der ASM-Regeln modular ist, sind Ergänzungen um neue Arten von Aktionen oder Änderungen am Flussalgorithmus ohne großen Aufwand möglich.

Die Formalisierung ist die Voraussetzung für die Implementierung eines prototypischen Werkzeugs, der "ActiveChartsIDE", welches in Kapitel 6 vorgestellt wird und mit dessen Hilfe unser Ansatz der Modellgetriebenen Softwareentwicklung überprüft werden kann. Es wird zunächst die Architektur vorgestellt und dann auf die Arbeitsweise eines Entwicklers mit der ActiveChartsIDE eingegangen. Die Klassen- und Aktivitätsdiagramme werden mit Microsoft Visio 2003 erstellt, von der ActiveChartsIDE importiert und in eine XML-Darstellung überführt. Der aus dem Klassendiagramm erzeugte Code wird durch eigene Methoden vom Entwickler in einer Entwicklungsumgebung seiner Wahl ergänzt. Eine Laufzeitbibliothek führt die Diagramme aus und ruft die Methoden an den entsprechenden Stellen auf. Die ActiveChartsIDE kann bei der Ausführung für das Debugging und

die Animation der Diagramme verwendet werden. Ohne unser Werkzeug wird die implementierte Anwendung “standalone” und ohne jegliche Änderungen und Neugenerierung von Code ausgeführt.

Die abschließenden Kapitel enthalten eine Übersicht über die Beiträge dieser Arbeit, Mathematische Konventionen sowie umfangreiche Fallbeispiele, die mit unserem Tool implementiert und getestet wurden.

Teile dieser Arbeit wurden bereits in [SRKS05, SGK+05, Sar05, Sar06, SG06] veröffentlicht.

Bibliography

[BBG05] Sami Beydeda, Matthias Book, and Volker Gruhn. *Model-Driven Software Development*. Springer, July 2005.

[BCR00a] E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In T Rus, editor, *Algebraic Methodology and Software Technology*, volume 1816 of *Lecture Notes in Computer Science*, pages 293–308. Springer-Verlag, 2000.

[BCR00b] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. Modeling the Dynamics of UML State Machines. In *ASM '00: Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, pages 223–241, London, UK, 2000. Springer-Verlag.

[Ber05] Daniel Bernauer. Testfälle für Klassen- und Aktivitätsdiagramme der UML2 zur Überprüfung der Sprachkonformität und -vollständigkeit von UML2 Werkzeugen. Master's thesis, Universität Ulm, September 2005.

[BFGS05] Egon Börger, Nicu G. Fruja, Vincenzo Gervasi, and Robert F. Stärk. A high-level modular definition of the semantics of C#. volume 336, Issue 2-3 of *Electronic Notes in Theoretical Computer Science*, pages 235–284. Elsevier, May 2005.

[BG03] J.P. Barros and L. Gomes. Actions as Activities and Activities as Petri nets. In J. Jürjens, B. Rumpe, R. France, and E.B. Fernandez, editors, *Critical Systems Development with UML: Proceedings of the UML'03 workshop*, pages 129–135. TUM-I0317, September 2003.

[BG04] C. Bock and M. Gruninger. Inputs and Outputs in the Process Specification Language. NISTIR 7152, National Institute of Standards and Technology, 2004.

[BG05] C. Bock and M. Gruninger. PSL: A semantic domain for flow models. In *Software and Systems Modeling*, volume 4, pages 209–231, May 2005.

[BGN+03] Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Model-Based Testing with AsmL.NET. In *1st European Conference on Model-Driven Software Engineering*, December 2003.

[Bia04] B. Biafore. *Visio 2003 Bible*. Wiley Publishing, Inc., 2004.

[BM97] R. Bruni and U. Montanari. Zero-Safe Nets, or Transition Synchronization Made Simple. In *Electronic Notes in Theoretical Computer Science: Proceedings of EXPRESS'97, 4th workshop on Expressiveness in Concurrency*, volume 7, pages 1–19. Elsevier Science, 1997.

[Boc04] C. Bock. UML 2 Activity and Action Models Part 4: Object Nodes. *Journal of Object Technology*, 3(1):27–41, 2004. http://www.jot.fm/issues/issue_2004_01/column3.

[BS01] Mike Barnett and Wolfram Schulte. The ABCs of Specification: AsmL, Behavior, and Components. *Informatica*, 25(4):517–526, Nov. 2001.

[BS03] E. Börger and R. Stärk. *Abstract State Machines*. Springer-Verlag, 2003.

[CGN+05] Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. Technical Report MSR-TR-2005-59, Microsoft Research, May 2005.

[CW00] Giuseppe Del Castillo and Kirsten Winter. Model Checking Support for the ASM High-Level Language. In S. Graf and M. Schwartzbach, editors, *International Conference TACAS*, number 6 in LNCS 1785, pages 331–346. Springer-Verlag, 2000.

[dMGMP02] María del Mar Gallardo, Pedro Merino, and Ernesto Pimentelis. Debugging UML Designs with Model Checking. *Journal of Object Technology*, 1(2):101–117, July-August 2002.

[EGGP00] R. Eschbach, U. Glässer, R. Gotzhein, and A. Prinz. On the Formal Semantics of SDL-2000: A Compilation Approach based on an Abstract SDL Machine. In *International Workshop on Abstract State Machines (ASM 2000)*, LNCS. Springer-Verlag, 2000.

[Esh02] Rik Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, University of Twente, 2002.

[Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[EW01] Rik Eshuis and Roel Wieringa. A Real-Time Execution Semantics for UML Activity Diagrams. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering. 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6. 2001 Proceedings*, volume 2029, pages 76–90. Springer, 2001.

[EW02] Rik Eshuis and Roel Wieringa. Verification support for workflow design with UML activity graphs. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 166–176, New York, NY, USA, 2002. ACM Press.

[EW04] Rik Eshuis and Roel Wieringa. Tool Support for Verifying UML Activity Diagrams. In *IEEE Transactions on Software Engineering*, volume 30, pages 437–447. IEEE, 2004.

[Fre06] Patrick Frey. Development and validation of an executable ASM specification of UML 2 Activity Diagrams. Master's thesis, University of Ulm, 2006. (to appear).

[Ges05] Dominik Gessenharter. Visualisierung der Simulation von graphischen Prototypen als Möglichkeit des interaktiven Debuggings von UML 2.0 Aktivitätsdiagrammen. Master's thesis, Universität Ulm, 2005.

[GR00] Angelo Gargantini and Elvinia Riccobene. Encoding Abstract State Machines in PVS. In Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors, *Abstract State Machines*, volume 1912 of *Lecture Notes in Computer Science*, pages 303–322. Springer, 2000.

[GRS04] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic Essence of AsmL. Technical Report MSR-TR-2004-27, Microsoft Research, 2004.

[GT01] Y. Gurevich and N. Tillmann. Partial Updates: Exploration. *Journal of Universal Computer Science*, 7(11):917–951, 2001.

[Gur94] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–37. Oxford University Press, 1994.

[Hau05] J.H. Hausmann. *Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages*. PhD thesis, Universität Paderborn, 2005.

[Hea06] A. Hartman and et al. Model Execution Project, 2006. http://www.haifa.ibm.com/projects/software/ple/mex/.

[IBM05] IBM: Rational Unified Process, 2005. http://www-306.ibm.com/software/awdtools/rup/.

[ILo05] ILogix, 2005. http://www.ilogix.com/.

[ITU00a] International Telecommunication Union ITU. Z.100 Annex F1 (11/00) SDL formal definition - General overview, 2000. http://www.sdl-forum.org.

[ITU00b] International Telecommunication Union ITU. Z.100 Annex F2 (11/00) SDL formal definition - Well-formedness and Transformation rules, 2000. http://www.sdl-forum.org.

[ITU00c] International Telecommunication Union ITU. Z.100 Annex F3 (11/00) SDL formal definition - Dynamic Semantics, 2000. http://www.sdl-forum.org.

[ITU02] International Telecommunication Union ITU. Z.100 (08/02) Specification and description language (SDL), 2002. http://www.sdl-forum.org.

[JEJ02] Yan Jin, Robert Esser, and Jorn W. Janneck. Describing the Syntax and Semantics of UML Statecharts in a Heterogeneous Modelling Environment. In *Diagrams*, pages 320–334, 2002.

[Jür02] J. Jürjens. A UML statecharts semantics with message-passing. In *ACM Symposium on Applied Computing (SAC)*, pages 1009–1013. ACM Press, 2002.

[Kay04] Michael Kay. *XPath 2.0 Programmer's Reference*. Wrox Press, 2004.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture — Practice and Promise*. Addison–Wesley, 2003.

[Lar05] Craig Larman. *Applying UML and Patterns*. Prentice Hall, 2005.

[Mat05] Mathworks, 2005. http://www.mathworks.com/.

[MB02] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison–Wesley, 2002.

[McN04] Ashley McNeile. MDA: The Vision with the Hole?, 2004. http://www.metamaxim.com/download/documents/MDAv1.pdf.

[MDA03] Object Management Group, MDA Guide 1.0.1, Document 03-06-01, June 2003, 2003.

[Mic05] Microsoft Developer Network. Create Elegant Code with Anonymous Methods, Iterators, and Partial Classes, 2005. http://msdn.microsoft.com/msdnmag/issues/04/05/c20/default.aspx.

[Mod05] Modelware Project, WP1 Modelling Techniques. Model Simulation Scheme – Definition, August 2005. http://www.modelware-ist.org/public_area/publications/reports/WP1_Modelling_Techniques/D1.3_Model_Simulation_Scheme-Definition.pdf.

[NEG05] C. Nagel, B. Evjen, and J. Glynn. *Professional C# 2005*. Wrox Press, 2005.

[Obe03] I. Ober. An ASM semantics of UML derived from the meta-model and incorporating actions. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines: Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 356–371. Springer-Verlag, 2003.

[Pre04] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, 2004.

[Sar05] Stefan Sarstedt. Overcoming The Limitations of Signal Handling when Simulating UML 2 Activity Charts. In J.M. Feliz-Teixeira and A.E. Carvalho Brito, editors, *Proceedings of the 2005 European Simulation and Modelling Conference (ESM'05)*, pages 61–65, October 2005.

[Sar06] Stefan Sarstedt. Model-Driven Development with ActiveCharts - Tutorial. Technical Report 2006-01, University of Ulm, March 2006.

[Sax03] Steve Saxon. XPath Querying Over Objects with ObjectXPathNavigator. 2003. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexxml/html/xml03172003.asp.

[SB04] Robert F. Stärk and Egon Börger. An ASM specification of C# threads and the .NET memory model. In *ASM 2004*, pages 38–60, 2004.

[SG05] Rajesh Sudarsan and Jeff Gray. Meta-Model Search: Using XPath to Search Domain-Specific Models. In *The 2005 International Conference on Software Engineering Research and Practice (SERP '05)*, 2005.

[SG06] Stefan Sarstedt and Walter Guttmann. An ASM Semantics of Token Flow in UML 2 Activity Diagrams. In *Perspectives of System Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006*. Springer, 2006. (to appear).

[SGK+05] S. Sarstedt, D. Gessenharter, J. Kohlmeyer, A. Raschke, and M. Schneiderhan. ActiveChartsIDE: An Integrated Software Development Environment Comprising a Component for Simulating UML 2 Activity Charts. In J.M. Feliz-Teixeira and A.E. Carvalho Brito, editors, *Proceedings of the 2005 European Simulation and Modelling Conference (ESM'05)*, pages 66–73, October 2005.

[SH05] H. Störrle and J.H. Hausmann. Towards a Formal Semantics of UML 2.0 Activities. In P. Liggesmeyer, K. Pohl, and M. Goedicke, editors, *Software Engineering 2005*, volume P-64 of *Lecture Notes in Informatics*, pages 117–128. Gesellschaft für Informatik, 2005.

[SKRS05] S. Sarstedt, J. Kohlmeyer, A. Raschke, and M. Schneiderhan. Targeting System Evolution by Explicit Modeling of Control Flows using UML 2 Activity Charts. In *Proceedings of the International Conference on Programming Languages and Compilers (PLC '05), Technical Session on Support for Unanticipated Software Evolution*, 2005.

[SRKS05] S. Sarstedt, A. Raschke, J. Kohlmeyer, and M. Schneiderhan. A New Approach to Combine Models and Code in Model Driven Development. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '05), International Workshop on Applications of UML/MDA to Software Systems*, 2005.

[Stö04a] H. Störrle. Semantics of Control-Flow in UML 2.0 Activities. In *Symposium On Visual Language And Human Centric Computing*, pages 235–242. IEEE, sept 2004.

[Stö04b] H. Störrle. Semantics of Exceptions in UML 2.0 Activities. Technical Report TR0402, Ludwig-Maximilians-Universität München, Institut für Informatik, apr 2004.

[Stö04c] H. Störrle. Semantics of Structured Nodes in UML 2.0 Activities. In K. et al. Koskimies, editor, *Nordic Workshop on UML*, volume 2, pages 19–32, aug 2004.

[Stö05] H. Störrle. Semantics and Verification of Data Flow in UML 2.0 Activities. In M. Minas, editor, *Workshop on Visual Languages and Formal Methods*, volume 127, Issue 4 of *Electronic Notes in Theoretical Computer Science*, pages 35–52. Elsevier, 2005.

[UML01] Object Management Group. *UML 1.4 Specification*, September 2001.

[UML03] Object Management Group. *UML 2.0 Superstructure Final Adopted Specification*, August 2003. http://www.omg.org/cgi-bin/doc?ptc/03-08-02.

[UML04] Object Management Group. *UML 2.0 Superstructure Specification, Convenience Document*, October 2004. http://www.omg.org/cgi-bin/doc?ptc/04-10-02.

[UML05] Object Management Group. *UML 2.0 Superstructure Specification*, August 2005. http://www.omg.org/cgi-bin/doc?formal/05-07-04.

[UML06] UML 2.0 Semantics Project. Homepage, 2006. http://www.cs.queensu.ca/~stl/internal/uml2/.

[VK05] V. Vitolins and A. Kalnins. Semantics of UML 2.0 Activity Diagram for Business Modeling by Means of Virtual Machine. In *Ninth International EDOC Enterprise Computing Conference*, pages 181–192. IEEE, 2005.

[WE04] M. H. Walter and N. J. Eaton. *Microsoft Office Visio 2003 Inside Out*. Microsoft Press, 2004.

[Win01] K. Winter. *Model Checking Abstract State Machines*. PhD thesis, Technical University of Berlin, 2001.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language. Getting your models ready for MDA*. Addison-Wesley Professional, 2nd edition, September 2003.

[WM97] M. Walicki and S. Meldal. Algebraic approaches to nondeterminism - an overview. *ACM Computing Surveys*, 29(1):30–81, 1997.

[WYY+04] Linzhang Wang, Jiesong Yuan, Xiaofeng Yu, Jun Hu, Xuandong Li, and Guoliang Zheng. Generating Test Cases from UML Activity Diagram based on Gray-Box Method. In *APSEC*, pages 284–291. IEEE Computer Society, 2004.