The Cloud Application Modelling and Execution Language (CAMEL)

Alessandro Rossini · Kiriakos Kritikos · Nikolay Nikolov · Jörg Domaschka · Daniel Seybold · Frank Griesinger · Daniel Romero · Michal Orzechowski · Georgia Kapitsaki · Achilleas Achilleos

08 March 2017

Abstract Cloud computing provides ubiquitous networked access to a shared and virtualised pool of computing capabilities that can be provisioned with minimal management effort [27]. Cloud applications are deployed on cloud infrastructures and delivered as services. The PaaSage project aims to facilitate the modelling and execution of cloud applications by leveraging model-driven engineering (MDE) and by exploiting multiple cloud infrastructures. The Cloud Application Modelling and Execution Language (CAMEL) is the core modelling and execution language developed in the PaaSage project and enables the specification of multiple aspects of cross-cloud applications (*i.e.*, applications deployed across multiple private, public, or hybrid cloud infrastructures). By exploiting models at both design- and run-time, and by allowing both direct and programmatic manipulation of models, CAMEL enables the management of self-adaptive cross-cloud applications (*i.e.*, cross-cloud applications that autonomously adapt to changes in the environment, requirements, and usage). In this paper, we describe the design and implementation of CAMEL, with emphasis on the integration of heterogeneous domain-specific languages (DSLs) that cover different aspects of self-adaptive cross-cloud applications. Moreover, we provide a real-world running example to illustrate how to specify models in a concrete textual syntax and how to dynamically adapt these models during the application life cycle. Finally, we provide an evaluation of CAMEL's usability and usefulness, based on the technology acceptance model (TAM).

Keywords cloud computing, domain-specific language, model-driven engineering, models@run-time

1 Introduction

MDE is a branch of software engineering that aims to improve the productivity, quality, and cost-effectiveness of software development by shifting the paradigm from code-centric to model-centric. MDE promotes the use of models and model transformations as the primary

SINTEF, Oslo, Norway, E-mail: alessandro.rossini@sintef.no · FORTH, Heraklion, Greece · SINTEF, Oslo, Norway · University of Ulm, Germany · University of Ulm, Germany · University of Ulm, Germany · Inria, Lille, France · AGH, Krakow, Poland · University of Cyprus · University of Cyprus

assets in software development, where they are used to specify, simulate, generate, and manage software systems. This approach is particularly relevant when it comes to the modelling and execution of cross-cloud applications (*i.e.*, applications deployed across multiple private, public, or hybrid cloud infrastructures). This solution allows to exploit the peculiarities of each cloud service and hence to optimise the performance, availability, and cost of the applications.

Models can be specified using general-purpose languages like the Unified Modeling Language (UML) [30]. However, to fully unfold the potential of MDE, models are frequently specified using domain-specific languages (DSLs), which are tailored to a specific domain of concern. The PaaSage[1] project exploits the latter approach and provides an integrated platform to support the modelling and execution of cross-cloud applications. To achieve this goal, PaaSage developed the Cloud Application Modelling and Execution Language (CAMEL). This DSL allows to specify multiple aspects of cross-cloud applications, such as provisioning, deployment, service level, monitoring, scalability, providers, organisations, users, roles, security, and execution.

CAMEL supports the models@run-time [6] approach, which provides an abstract representation of the underlying running system, whereby a modification to the model is enacted on-demand in the system, and a change in the system is automatically reflected in the model.

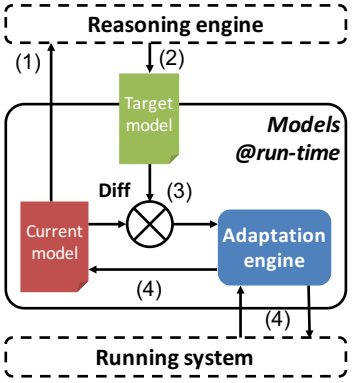


Figure 1: Models@run-time architecture

Fig. 1 depicts the architecture of models@run-time. A reasoning engine reads the current model (step 1) and produces a target model (step 2). Then, the run-time environment computes the difference between the current model and the target one (step 3). Finally, the adaptation engine enacts the adaptation by modifying only the parts of the cross-cloud application necessary to account for the difference and the target model becomes the current model (step 4).

CAMEL was designed and implemented to allow the design-time specification of models by users as well as their run-time manipulation by reasoners. By exploiting models at both design- and run-time, and by allowing both direct and programmatic manipulation of models, CAMEL enables the management of self-adaptive cross-cloud applications (*i.e.*,

[1] `http://www.paasage.eu`

cross-cloud applications that autonomously adapt to changes in the environment, requirements, and usage). This represents the main motivation for our research and contribution of our work, since, to the best of our knowledge, no other integrated language in the literature supports the management of self-adaptive cross-cloud applications (see Section 15).

Structure of the document: The remainder of the document is organised as follows. Section 2 describes the role of CAMEL models in a self-adaptation workflow. Section 3 presents some technologies used to design and implement CAMEL. Section 4 describes the integration of heterogeneous domain-specific languages in CAMEL. Sections 5–13 present the various packages of the CAMEL metamodel along with corresponding sample models in concrete syntax. Section 14 presents an evaluation of CAMEL based on the technology acceptance model (TAM). Finally, Section 15 compares the proposed approach with related work, while Section 16 draws conclusions and outlines plans for future work.

2 CAMEL and the Self-Adaptation Workflow

The components managing the life cycle of cross-cloud applications are integrated by leveraging CAMEL models. These models are progressively refined throughout the *modelling*, *deployment*, and *execution* phases of a self-adaptation workflow based on the models@runtime approach, as proposed in PaaSage [37].

Figure 2 shows the self-adaptation workflow. The *white trapezes* represent the activities performed by the user. The *white rectangles* represent the processes executed by the PaaSage platform. The *coloured shapes* represent the modelling artefacts, whereby the blue ones pertain to the modelling phase, the red ones to the deployment phase, and the green ones to the execution phase.

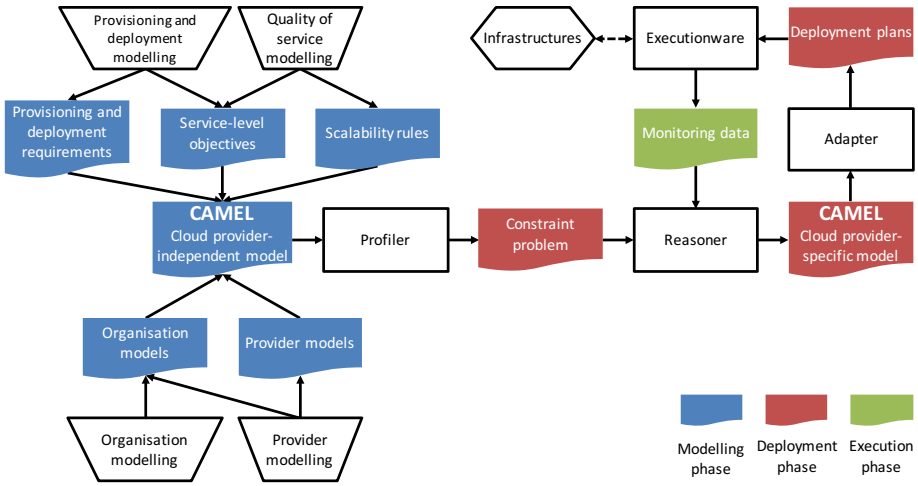


Figure 2: CAMEL models in the self-adaptation workflow

In the remainder of the document, we adopt the Scalarm[2] [25] use case as a real-world running example to illustrate how to specify multiple aspects of cloud applications in

[2] http://www.scalarm.com/

CAMEL, and how these specifications facilitate the deployment of cloud applications across multiple clouds and their self-adaptation to changes in the environment, requirements, and usage. Scalarm is a massively self-scalable platform for data farming. Data farming experiments utilise high-performance and high-throughput computing to generate large amounts of data via simulations. These data are analysed to obtain new insights into the studied phenomena. The architecture of Scalarm is based on the principles of service-oriented architecture (SOA) and consists of the following services:

- *Experiment Manager* provides a graphical user interface to coordinate the execution of data farming experiments.
- *Simulation Manager* provides a wrapper to execute the simulations on multiple computational infrastructures.

As data farming experiments are often executed on a large amount of computational infrastructures and across multiple data centres, the Scalarm use case is particularly suitable to illustrate the features of CAMEL.

Modelling phase. The users design a *cloud provider-independent model* (CPIM), which specifies the deployment of a cross-cloud application along with its requirements and objectives (*e.g.*, on virtual hardware, location, and service level) in a cloud provider-independent way.

Figure 3(a) shows the CPIM of Scalarm in graphical syntax. It consists of an Experiment Manager (represented by ExpMan) hosted on a GNU/Linux virtual machine (represented by Linux). Moreover, the Experiment Manager communicates with a Simulation Manager (represented by SimMan) hosted on a GNU/Linux virtual machine in a data centre in Norway. Finally, the Experiment Manager has a service-level objective specifying that the response time must be below 100 ms.

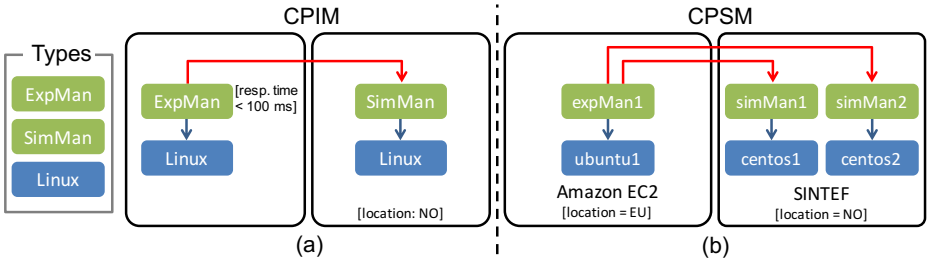


Figure 3: Sample CAMEL models: (a) CPIM; (b) CPSM

Deployment phase. The Profiler component consumes the CPIM, matches this model with the profile of cloud providers, and produces a *constraint problem*. The Reasoner component solves the constraint problem (if possible) and produces a *cloud provider-specific model* (CPSM), which specifies the deployment of a cross-cloud application along with its requirements and objectives in a cloud provider-specific way. The Adapter component consumes the CPSM and produces *deployment plans*, which specify the platform-specific details of the deployment.

For instance, the Profiler could match the CPIM of Scalarm with the profile of cloud providers, identify Amazon EC2, Google Compute Engine, and Azure as the three cloud providers satisfying the virtual hardware requirements and service-level objectives of the Experiment Manager (response time below 100 ms). Moreover, the Profiler could identify SINTEF[3] and EVRY[4] as the two cloud providers satisfying the virtual hardware and location requirements of the Simulation Manager (data centre in Norway), and produce a corresponding constraint problem. Then, the Reasoner could rank Amazon and SINTEF as the best cloud providers to satisfy these requirements and produce a corresponding CPSM.

Figure 3(b) shows the CPSM in graphical syntax. It consists of an Experiment Manager *instance*, which is hosted on a Ubuntu 14.04 instance at Amazon EC2 in the EU. Moreover, the Experiment Manager instance communicates with two Simulation Manager instances, which are hosted on two CentOS 7 virtual machine instances at SINTEF in Norway.

Execution phase. The Executionware consumes the deployment plans and enacts the deployment of the application components on suitable cloud infrastructures. The PaaSage platform records monitoring data about the application execution from the Executionware, which allows the Reasoner to continuously revise the solution to the constraint problem to better exploit the cloud infrastructures.

3 Technologies

In order to design and implement CAMEL, we adopted the Eclipse Modeling Framework (EMF)[5] along with Object Constraint Language (OCL) [31], Xtext[6], and Connected Data Objects (CDO).[7] In the following, we outline these technologies and describe how they facilitate the implementation of the PaaSage platform described in Section 2.

Eclipse Modeling Framework (EMF). In MDE, the abstract syntax of a DSL is typically defined by its metamodel, which describes the set of concepts, their attributes, and their relations, as well as the rules for combining these concepts to specify valid models conforming to this metamodel [30]. EMF is a modelling framework that facilitates defining DSLs. It provides the Ecore metamodel, which is an Ecore model that conforms to itself (*i.e.*, it is reflexive). The CAMEL metamodel is an Ecore model that conforms to the Ecore metamodel.

EMF allows to generate a Java class hierarchy representation of a metamodel. The Java representation provides a set of APIs that allows the programmatic manipulation of models.

Object Constraint Language (OCL). EMF enables to check the cardinality constraints on properties and to validate models against their metamodels. However, it lacks the expressiveness required to capture all of the semantics of the domain. OCL is a declarative language for specifying expressions on metamodels that are evaluated on models conforming to these metamodels. Eclipse OCL[8] is a tool-supported implementation of OCL that integrates with

[3] `https://www.sintef.no/en/`
[4] `https://www.cloudservices.no/`
[5] `https://www.eclipse.org/modeling/emf/`
[6] `https://eclipse.org/Xtext/`
[7] `https://www.eclipse.org/cdo/`
[8] `http://wiki.eclipse.org/OCL`

EMF. The CAMEL metamodel is annotated with OCL expressions to capture part of the semantics of cross-cloud applications and to guarantee the consistency, correctness, and integrity of CAMEL models at both design-time and run-time.

Xtext. In MDE, the concrete syntax describes the textual or graphical notation that renders the concepts, attributes, and relations in the abstract syntax. The concrete syntax may vary depending on the domain, *e.g.*, a DSL could provide a textual notation as well as a graphical notation along with the corresponding serialisation in XML Metadata Interchange (XMI) [32]. Xtext is a language development framework that is based on- and integrates with EMF. It facilitates the implementation of Eclipse-based IDEs providing features, such as syntax highlighting, code completion, code formatting, static analysis, and serialisation. The concrete syntax of CAMEL is a textual syntax defined as an Xtext grammar. The textual syntax was preferred over the graphical syntax by the early adopters of CAMEL (see Section 14).

Connected Data Objects (CDO). CDO is semi-automated persistence framework that works natively with Ecore models and their instances. It provides a model repository where clients can persist, share, and query their models. It provides features that satisfy the design-time and run-time requirements of the self-adaptation workflow (see Section 2), such as abstraction from database management systems (DBMSs), validation, transactional processing, optimistic versioning [40], automatic notification, auditing, and role-based security [22].

Thanks to the combination of EMF, Eclipse OCL, and Xtext, we realised the CAMEL Textual Editor, which allows users not only to specify CAMEL models but also to validate them. Moreover, thanks to these technologies and CDO, we realised the models@run-time approach, which allows multiple reasoners to progressively refine CAMEL models throughout the various phases of the self-adaptation workflow (see Section 2).

4 Integration of Heterogeneous DSLs

CAMEL integrates and extends DSLs developed before the PaaSage project, namely Cloud Modelling Language (CloudML) [14, 12, 13], Saloon [35, 34, 36], and the organisation part of CERIF [19]. In addition, CAMEL integrates new DSLs developed within the PaaSage project, such as the Scalability Rule Language (SRL) [21, 10].

At the beginning of the PaaSage project, CAMEL consisted of a family of loosely coupled and heterogeneous DSLs, each having their own abstract and concrete syntaxes. Moreover, the models specified using these DSLs were persisted in a relational database, the so-called Metadata Database (MDDB) [23].

This initial solution illustrates some of the challenges that are inherent to DSL integration and evolution [29]. First, the elements in the metamodels of the DSLs needed to be matched to the elements in the schema of the MDDB using a custom mapping. Such a mapping needed to be bi-directional, since the data persisted in the MDDB had to be transformed back to the (abstract or concrete) syntax of the corresponding DSL. Second, the size of the schema became cumbersome as it needed to cover a considerable number of concepts from the DSLs—the last version had 70 tables and multiple referential integrity constraints. Third, because of the complexity of the MDDB, the custom queries for reading and writing data also became complex. Finally, both the DSLs and the MDDB needed to evolve to cope with the changing requirements in the domain. This evolution affected the metamodels of the DSLs, the schema of the MDDB, the data persisted in the MDDB, and the back-end code managing this data. Therefore, we quickly realised that we needed a better solution than the

time-consuming and error-prone custom mapping approach, and opted for an integration approach, whereby the family of loosely coupled and heterogeneous DSLs was transformed into a single language.

The integration approach had to overcome certain language-specific issues. First, similar or equivalent concepts in the DSLs were duplicated. Second, concepts in the DSLs were defined at different levels of granularity. Third, the abstract and concrete syntaxes of the DSLs were defined using different formalisms and notations. Therefore, the integration solution had to: (a) join equivalent concepts and separate similar concepts into respective sub-concepts; (b) homogenise the remaining concepts so that they are defined at the same level of granularity; (c) enforce a uniform formalism and notation for the abstract and concrete syntaxes; and (d) enforce the consistency, correctness, and integrity of the models.

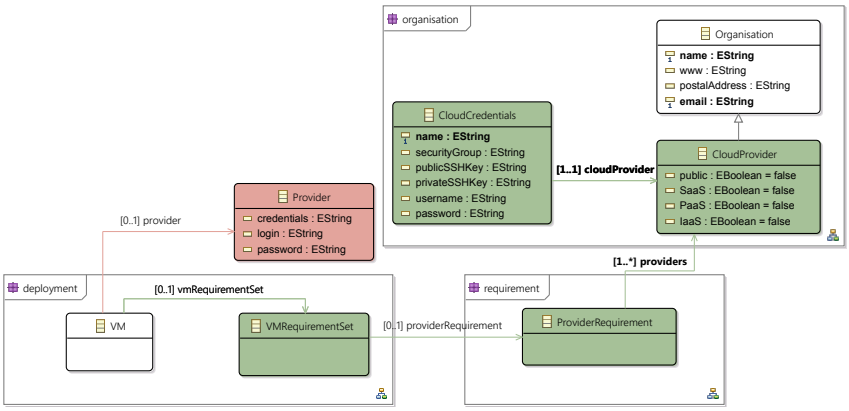


Figure 4: Integrating and extending CloudML and CERIF into CAMEL

In CloudML, providers are only described in terms of user credentials, whereby the attributes of providers do not reflect an actual cloud provider, but are instead used as a simple means to access cloud infrastructures. In order to be able to describe providers in a broader sense, we decided to integrate some relevant concepts from CERIF and CloudML into CAMEL. First, we ported the abstract syntaxes of the two DSLs to the same technical space of EMF. The abstract syntax of CloudML was already defined as an Ecore model, while the abstract syntax of CERIF was defined as an XML Schema. In order to transform the XML schema into an Ecore model, we used the importer provided by the XSD library of the Model Development Tools (MDT) project[9], which is integrated with the EMF stack. Second, we included the concepts from CloudML in a separate package called deployment, and a subset of the concepts from CERIF in a package called organisation. Third, we coupled the concepts in the deployment package with the ones in the organisation package.

Figure 4 shows an example of how we coupled the two concepts of VM and Provider in the deployment package with Organisation in the organisation package. Provider only allowed for a partial definition of providers, so we deleted this concept and added CloudProvider as an extension of Organisation in the organisation package. The association of cloud providers with virtual machines is realised through the ProviderRequirement concept in the requirement

[9] https://www.eclipse.org/modeling/mdt/

package. This solution allows for a comprehensive definition of providers as well as provider requirements, while still reusing most of the original concepts from CloudML and CERIF.

In order to guarantee the consistency, correctness, and integrity of CAMEL models, the CAMEL metamodel is annotated with various OCL expressions. Listing 1 shows an example of an OCL constraint that refers to concepts in the deployment and organisation packages and ensures that the deployment and requirement models only includes the requirements of providers for which the respective cloud credentials exist in the organisation model of the owning user.

Listing 1: OCL constraint checking that credentials exist for a requested provider

```
invariant credentials_exist_for_requested_provider:
  deploymentModels->forAll(p |
    ((p.globalVMRequirementSet <> null and p.globalVMRequirementSet.
         providerRequirement <> null) implies (cloudCredentials->exists(c | p.
         globalVMRequirementSet.providerRequirement.providers->includes(c.
         cloudProvider))))
    and
    (p.vms->forAll(v | (v.vmRequirementSet <> null and v.vmRequirementSet.
         providerRequirement <> null) implies (cloudCredentials->exists (x | v.
         vmRequirementSet.providerRequirement.providers->includes(x.cloudProvider)
         ))))
  );
```

5 CAMEL Design and Syntax

CAMEL has been designed based on the following requirements, among others:

- *Cloud provider-independence (R_1)*: CAMEL should support a cloud provider-agnostic specification of multiple aspects of cross-cloud applications (*i.e.*, provisioning, deployment, service level, monitoring, scalability, providers, organisations, users, roles, security, and execution). This will prevent vendor lock-in.
- *Separation of concerns (R_2)*: CAMEL should support loosely-coupled packages (or modules) corresponding to multiple aspects of cross-cloud applications. This will facilitate the development of models.
- *Reusability (R_3)*: CAMEL should support reusable types for multiple aspects of cross-cloud applications. This will ease the evolution of models.
- *Abstraction (R_4)*: CAMEL should provide an up-to-date, abstract representation of the running system. This will enable the reasoning, simulation, and validation of the adaptation actions before their enactment.

CAMEL is inspired by component-based approaches, which facilitate separation of concerns (R_2) and reusability (R_3). In this respect, deployment models can be regarded as assemblies of components exposing ports, and bindings between these ports.

In addition, CAMEL implements the *type-instance* pattern [1], which also facilitates reusability (R_3) and abstraction (R_4). This pattern exploits two flavours of typing, namely *ontological* and *linguistic* [26]. Figure 5 illustrates these two flavours of typing. SL (short for Small GNU/Linux) represents a reusable type of virtual machine. It is linguistically typed by the class VM (short for virtual machine). SL1 represents an instance of the virtual machine SL. It is ontologically typed by SL and linguistically typed by VMInstance.

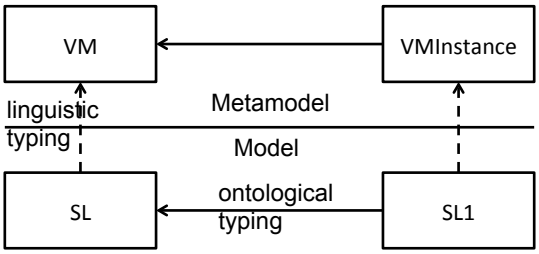


Figure 5: Linguistic and ontological typing

CAMEL and the CAMEL Textual Editor are available under Mozilla Public License 2.0[10] in the Git repository at OW2.[11] The CAMEL metamodel is an Ecore model organised into packages, whereby each package reflects an aspect (or domain).

As mentioned, in the remainder of the document, we adopt the Scalarm[12] [25] use case as a real-world running example to illustrate how to specify CAMEL models in textual syntax. The complete Scalarm CAMEL model in textual syntax can be downloaded from the Git repository at OW2.[13] This running example covers the most commonly used concepts in CAMEL. The interested reader may refer to [39] for a complete description of the abstract syntax of CAMEL.

Listing 2: Scalarm sample `application`

```
camel model ScalarmModel {

  application ScalarmApplication {
    version: 'v1.0'
    owner: AGHOrganisation.morzech
    deployment models [ScalarmModel.ScalarmDeployment]
  }

  organisation model AGHOrganisation {
    ...
    user morzech {
      ...
    }
  }

  deployment model ScalarmDeployment {
    ...
  }
}
```

Listing 2 shows an excerpt of the CAMEL model of Scalarm in textual syntax. An element of a CAMEL model is specified by the name of the element followed by an identifier in CamelCase and a block delimited by curly brackets. `camel model ScalarmModel {...}` specifies the CAMEL model of Scalarm, where `ScalarmModel` represents the identifier of this model. `application ScalarmApplication` specifies the Scalarm application itself.

A property is specified by the name of the property followed by a colon and a value. `version: 'v1.0'` specifies that the Scalarm application has version 1.0.

[10] https://www.mozilla.org/en-US/MPL/2.0/

[11] https://tuleap.ow2.org/plugins/git/paasage/camel

[12] http://www.scalarm.com/

[13] https://tuleap.ow2.org/plugins/git/paasage/camel?p=camel.git&a=blob&f=examples/Scalarm.camel

A reference to a single element is specified by the name of the reference followed by a colon and a fully qualified name conforming to the pattern: $id_1.id_2.\ldots.id_n$, where id_i refers to element at the i^{th} level of the containment path and id_n refers to the element at the leaf level, which is the element under discussion. `owner: AGHOrganisation.morzech` specifies that the application is owned by the user Michal Orzechowski by referring to the `user morzech` in the `organisation model AGHOrganisation` (see Section 9, Listing 11).

Finally, a reference to a list of elements is specified by the name of the reference followed by a comma separated list of fully qualified names delimited by square brackets.[14] `deployment models [ScalarmModel.ScalarmDeployment]` specifies that the Scalarm application has one deployment model by referring to the `deployment model ScalarmDeployment` (see Section 6, Listing 3).

6 Deployment

The deployment package of the CAMEL metamodel is based on CloudML[15] [14,12,13], which was developed in collaboration with the MODAClouds project.[16] CloudML consists of a tool-supported DSL for modelling and enacting the provisioning and deployment of cross-cloud applications, as well as for facilitating their dynamic adaptation, by leveraging MDE techniques and methods. In the following, we exemplify the main concepts in the deployment package.

Assume that we have to specify the Experiment Manager component of the Scalarm use case. Listing 3 shows this specification in textual syntax.

`internal component ExperimentManager` specifies a reusable type of the component Experiment Manager. `provided communication ExpManPort` specifies that the Experiment Manager provides its features through port 443. `required communication SimManPortReq` specifies that the Experiment Manager requires features from the Simulation Manager through port 80 (*cf.* Listing 5 for the corresponding specification of the communication binding). The property `mandatory` of `SimManPortReq` specifies that the Experiment Manager depends on the features of the Simulation Manager, and hence, that the Simulation Manager has to be deployed and started before the Experiment Manager. `required host CoreIntensiveUbuntuNorwayReq` specifies that the Experiment Manager requires a virtual machine that provides a large number of CPU cores, runs the operating system Ubuntu, and is located in Norway (*cf.* Listing 4 for the specification of the VM and VM requirements, and Listing 6 for the specification of the corresponding hosting binding).

`configuration ExperimentManagerConfiguration` specifies the commands to handle the life cycle of the Experiment Manager. `download`, `install`, and `start` specify the OS-specific shell scripts (in this case, Bash scripts) for downloading, installing, and starting the Experiment Manager, respectively. Note that, although not shown in this example, it is also possible to specify the `configure` and `stop` commands of a component.

The aforementioned commands are used by the Executionware during the execution phase to enact the deployment of the application components and to manage their life cycles (see Section 6.1).

Listing 3: Scalarm sample `internal component`

[14] Note that the colon is not used in this case.

[15] `http://cloudml.org`

[16] `http://www.modaclouds.eu`

```
deployment model ScalarmDeployment {

  internal component ExperimentManager {
    provided communication ExpManPort {port: 443}
    required communication SimManPortReq {port: 80 mandatory}
    required host CoreIntensiveUbuntuNorwayHostReq

    configuration ExperimentManagerConfiguration {
      download: 'wget http://www.scalarm.com/scalarm_exp_man.sh && chmod +x
     scalarm_exp_man.sh'
      install: './scalarm_exp_man.sh install'
      start: './scalarm_exp_man.sh start'
    }
  }
...
```

Then, assume that we have to specify the virtual machine on which the Experiment Manager can be deployed. Listing 4 shows this specification in textual syntax.

`vm CoreIntensiveUbuntuNorway` specifies a reusable type for a virtual machine. `requirement set` refers to the aforementioned requirement set `CoreIntensiveUbuntuNorwayRS`. `provided host CoreIntensiveUbuntuNorwayPort` specifies that the virtual machine provides a large number of CPU cores, runs the operating system Ubuntu, and is located in Norway (*cf.* Listing 6 for the specification of the corresponding hosting binding).

`requirement set CoreIntensiveUbuntuNorwayRS` specifies a reusable set of requirements for a virtual machine. `quantitative hardware`, `os`, and `location` refer to the requirements `CoreIntensive`, `Ubuntu`, and `NorwayReq`, respectively, in the `requirement model ScalarmRequirement` (*cf.* Listing 8), which in turn specify the hardware requirements encompassing a large number of CPU cores, the operating system requirement of Ubuntu, and the location requirement of Norway, respectively.

Listing 4: Scalarm sample `vm`

```
...
  vm CoreIntensiveUbuntuNorway {
    requirement set CoreIntensiveUbuntuNorwayRS
    provided host CoreIntensiveUbuntuNorwayPort
  }

  requirement set CoreIntensiveUbuntuNorwayRS {
    quantitative hardware: ScalarmRequirement.CoreIntensive
    os: ScalarmRequirement.Ubuntu
    location: ScalarmRequirement.NorwayReq
  }
...
```

Next, assume that we have to specify the communication binding between the Experiment Manager and the Simulation Manager. Listing 5 shows this specification in textual syntax.

`communication ExperimentManagerToSimulationManager` specifies a reusable type of communication binding between the Experiment Manager and the Simulation Manager. The `from .. to ..` block specifies that the communication binding is `from` the required communication port `SimManPortReq` of the component `ExperimentManager to` the provided communication port `SimManPort` of the component `SimulationManager`. `type: REMOTE` specifies that the Experiment Manager and the Simulation Manager must be deployed on separate virtual machine instances. Note that this property could have a value `ANY` (default) to specify that the components at each end of the communication can be deployed on any virtual machine instance(s), or `LOCAL` to specify that the components must be deployed on the same virtual machine instance.

Listing 5: Scalarm sample `communication`

```
...
  communication ExperimentManagerToSimulationManager {
    from ExperimentManager.SimManPortReq to SimulationManager.SimManPort
    type: REMOTE
  }
...
```

Finally, assume that we have to specify the hosting binding between the Experiment Manager and the virtual machine `CoreIntensiveUbuntuNorway`. Listing 6 shows this specification in textual syntax.

`hosting ExperimentManagerToCoreIntensiveUbuntuNorway` specifies a reusable type of hosting binding between the Experiment Manager and the virtual machine `CoreIntensiveUbuntuNorway`. The `from .. to ..` block specifies that the hosting binding is `from` the required hosting port `CoreIntensiveUbuntuNorwayPortReq` of the component `ExperimentManager` to the provided hosting port `CoreIntensiveUbuntuNorwayPortReq` of the virtual machine `CoreIntensiveUbuntuNorway`.

Listing 6: Scalarm sample `hosting`

```
...
  hosting ExperimentManagerToCoreIntensiveUbuntuNorway {
    from ExperimentManager.CoreIntensiveUbuntuNorwayPortReq to
     CoreIntensiveUbuntuNorway.CoreIntensiveUbuntuNorwayPort
  }
...
```

The types presented above can be instantiated in order to form a CPSM. In PaaSage, the instances within the deployment model are automatically manipulated during the deployment phase (see Section 2). In the general case, the instances could also be manipulated manually. Note that different CPSMs can adopt different instantiation patterns for communications and hosting bindings, while still conforming to the same CPIM. The interested reader may refer to [9] for an extensive discussion on the subject.

Listing 7 shows the specification of instances of the components, virtual machines, communications, and hosting bindings from the previous examples (*cf.* Listings 3, 4, 5, and 6) in textual syntax.

`vm instance CoreIntensiveUbuntuNorwayInst` specifies an instance of a virtual machine. `vm type` and `vm type value` refer to the virtual machine flavour `M1.LARGE` in the `provider model SINTEFProvider` (*cf.* Listing 12), which is compatible with the requirement set of the virtual machine template `CoreIntensiveUbuntuNorway` (*cf.* Listing 7).

`internal component instance ExperimentManagerInst` specifies an instance of the component `ExperimentManager`. The `connect .. to .. typed ..` and `host .. on .. typed ..` blocks specify instances of the communication `ExperimentManagerToSimulationManager` and the hosting `ExperimentManagerToCoreIntensiveUbuntuNorway`, respectively. `typed` refers to the identifier of the corresponding type.

Listing 7: Scalarm sample instances of `internal component`, `vm`, `communication`, and `hosting`

```
...
  vm instance CoreIntensiveUbuntuNorwayInst typed ScalarmModel.ScalarmDeployment.
     CoreIntensiveUbuntuNorway {
    vm type: ScalarmModel.SINTEFProvider.SINTEF.VM.VMType
    vm type value: ScalarmModel.SINTEFType.VMTypeEnum.M1.LARGE
    provided host instance CoreIntensiveUbuntuNorwayHostInst typed
     CoreIntensiveUbuntuNorway.CoreIntensiveUbuntuNorwayHost
  }

```

```
8   internal component instance SimulationManagerInst typed ScalarmModel.
       ScalarmDeployment.SimulationManager {
9     provided communication instance SimManPortInst typed SimulationManager.
       SimManPort
10    required host instance SimulationIntensiveUbuntuNorwayHostReqInst typed
       SimulationManager.SimulationIntensiveUbuntuNorwayHostReq
11  }
12
13  internal component instance ExperimentManagerInst typed ScalarmModel.
       ScalarmDeployment.ExperimentManager {
14    provided communication instance ExpManPortInst typed ExperimentManager.
       ExpManPort
15    required communication instance SimManPortReqInst typed ExperimentManager.Si,
       ManPortReq
16    required host instance CoreIntensiveUbuntuNorwayHostReqInst typed
       ExperimentManager.CoreIntensiveUbuntuNorwayHostReq
17  }
18
19  connect ExperimentManagerInst.SimManPortReqInst to SimulationManagerInst.
       SimManPortInst typed ScalarmModel.ScalarmDeployment.
       ExperimentManagerToSimulationManager
20
21  host ExperimentManagerInst.CoreIntensiveUbuntuNorwayHostReqInst on
       CoreIntensiveUbuntuNorwayInst.CoreIntensiveUbuntuNorwayHostInst typed
       ScalarmModel.ScalarmDeployment.ExperimentManagerToCoreIntensiveUbuntuNorway
22 ...
```

6.1 Interplay with Executionware

In order to execute a cloud application, the Executionware provisions VMs, deploys one or more component instances on these VMs, and starts these instances by relying on a cross-cloud orchestration framework. In PaaSage, this cross-cloud orchestration framework is Cloudiator [8]. Note the Executionware can solely rely on the information provided in a CAMEL model. Hence, the Executionware does not make any assumptions besides the information provided. In particular, the Executionware relies on life cycle handlers specified as script files, shell (*e.g.*, Bash or PowerShell) commands, or Java commands, in the `configuration` of a component in order to orchestrate the individual component instances. These life cycle handlers are invoked in the following order: *download*, *install*, *configure*, and *start*. Moreover, the Executionware relies on the communications specified in the deployment model in order to derive the dependencies between components and hence the correct order of deployment of component instances. This way, the provided communication instances are started before the required ones. The interested reader may refer to [39] for a detailed description of how the Executionware invokes these commands.

7 Requirements

The requirement package of the CAMEL metamodel is not based on existing DSLs and has been developed to enable the specification of requirements for cross-cloud applications. A requirement can be *hard*, such as a service level objective (SLO) (*e.g.*, response time < 100 ms), meaning that it is measurable and must be satisfied, or *soft*, such as a optimisation objective (*e.g.*, maximise performance), meaning that it is not measurable. A soft requirement has a *priority* from `0.0` to `1.0` that is used to rank these requirements when reasoning on the application and generating a new CPSM. In the following, we exemplify the main concepts in the requirement package.

Assume that we have to specify the requirements for the components of the Scalarm use case. Listing 8 shows this specification in textual syntax.

`quantitative hardware CoreIntensive` specifies that a virtual machine must have from 8 to 32 CPU cores and from 4 to 8 GB of RAM. `os Ubuntu` specifies that a virtual machine must run Ubuntu operating system 64-bit edition. `location requirement NorwayReq` specifies that a virtual machine must be deployed in Norway. `locations` refers to the location `NO` in the location model `ScalarmLocation`. The three requirements above are referred to by the `requirement set CoreIntensiveUbuntuNorwayRS` in the `deployment model ScalarmDeployment` (*cf.* Listing 4).

`slo CPUMetricSLO` specifies that the `metric condition CPUMetricCondition` is an SLO. `service level` refers to the `metric condition CPUMetricCondition` in the `metric model ScalarmModel` (*cf.* Listing 10).

`optimisation requirement MinimisePerformanceDegradationOfExperimentManager` specifies that the metric `MeanValueOfResponseTimeOfAllExprimentManagersMetric` of the component `ExperimentManager` should be minimised and that this minimisation has a priority `0.8`. `metric` refers to the `metric MeanValueOfResponseTimeOfAllExprimentManagersMetric` in the `metric model ScalarmModel` (*cf.* Listing 10), while `component` refers to the `internal component ExperimentManager` in the `deployment model ScalarmDeployment` (*cf.* Listing 4). `optimisation requirement MinimiseDataFarmingExperimentMakespan` specifies a similar optimisation requirement with priority `0.2`.

`group ScalarmRequirementGroup` specifies that the requirements `CPUMetricSLO`, `MinimisePerformanceDegradationOfExperimentManager`, and `MinimiseDataFarmingExperimentMakespan` are logically connected through the `AND` operator. Note that a requirement group also allows a requirement tree to be created. For example, a top-level requirement group could contain two or more requirement groups logically connected by an `OR` operator. Each of the latter requirement groups could in turn contain single requirements, such as SLOs, logically connected by an `AND` operator.

`horizontal scale requirement HorizontalScaleSimulationManager` specifies that the component `SimulationManager` can scale horizontally between 1 and 5 instances. `component` refers to the `internal component SimulationManager` in the `deployment model ScalarmDeployment` (*cf.* Listing 4).

Listing 8: Scalarm requirement model

```
requirement model ScalarmRequirement {

  quantitative hardware CoreIntensive {
    core: 8..32
    ram: 4096..8192
  }

  os Ubuntu {
    os: 'Ubuntu' 64os
  }

  location requirement NorwayReq {
    locations [ScalarmLocation.NO]
  }

  slo CPUMetricSLO {
    service level: ScalarmModel.ScalarmMetric.CPUMetricCondition
  }

  optimisation requirement MinimisePerformanceDegradationOfExperimentManager {
    function: MIN
    metric: ScalarmModel.ScalarmMetric.
     MeanValueOfResponseTimeOfAllExprimentManagersMetric
```

```
23      component: ScalarmModel.ScalarmDeployment.ExperimentManager
24      priority: 0.8
25    }
26
27    optimisation requirement MinimiseDataFarmingExperimentMakespan {
28      function: MIN
29      metric: ScalarmModel.ScalarmMetric.MakespanMetric
30      component: ScalarmModel.ScalarmDeployment.ExperimentManager
31      priority: 0.2
32    }
33
34    group ScalarmRequirementGroup {
35      operator: AND
36      requirements [ScalarmRequirement.CPUMetricSLO, ScalarmRequirement.
          MinimisePerformanceDegradationOfExperimentManager, ScalarmRequirement.
          MinimiseDataFarmingExperimentMakespan]
37    }
38
39    horizontal scale requirement HorizontalScaleSimulationManager {
40      component: ScalarmModel.ScalarmDeployment.SimulationManager
41      instances: 1 .. 5
42    }
43  }
```

8 Metrics and Scalability Rules

The scalability and metric packages of the CAMEL metamodel are based on the Scalability Rule Language (SRL) [21,10]. SRL enables the specification of rules that support complex adaptation scenarios of cross-cloud applications. In particular, SRL provides mechanisms for specifying cross-cloud behaviour patterns, metric aggregations, and the scaling actions to be enacted in order to change the provisioning and deployment of an application. SRL is inspired by the Esper Processing Language (EPL)[17] with respect to the specification of event patterns with formulas including logic operators and timing. SRL provides mechanisms for (a) specifying event patterns, (b) specifying scaling actions, and (c) associating these scaling actions with the corresponding event patterns. Moreover, in order to identify event patterns, the components of cross-cloud applications must be monitored. Therefore, SRL provides mechanisms for (d) expressing which components must be monitored by which metrics, and (e) associating event patterns with monitoring data. In the following, we exemplify the main concepts in the scalability and metric packages.

Assume that we have to specify scalability rules and metrics for the Scalarm use case. The SimulationManager scales out when the following conditions are met: (a) all instances have had an average CPU load beyond 50% for at least 5 *min*, and (b) concurrently at least one instance has had an average CPU load beyond 80% for at least 1 *min*. These conditions are represented by the following expression, where cpu_i and cpu_j represent the average CPU loads for instance i and j, respectively:

$$\forall i \mid cpu_i \geq 50 \wedge \exists j \mid cpu_j \geq 80$$

To implement this scenario, we specified a scalability and a metric model that represent, respectively: (a) the scalability rule along with the events used to trigger it, and (b) the metrics and conditions that, when evaluated, trigger the action of the scalability rule.

[17] http://esper.codehaus.org/

Listing 9 shows the scalability model in textual syntax. `non-functional event CPUAvgMetricNFEAll` specifies the violation of a metric condition. `metric condition` refers to the `metric condition CPUAvgMetricConditionAll` in the `metric model ScalarmMetric` (*cf.* Listing 10). `non-functional event CPUAvgMetricNFEAny` specifies a similar violation of a metric condition.

`binary event pattern CPUAvgMetricBEPAnd` specifies that the non-functional events above are logically connected through an AND operator.

`horizontal scaling action HorizontalScalingSimulationManager` specifies a scale-out action. `vm` and `internal component` refer to the `vm CPUIntensiveUbuntuNorway` and the `internal component SimulationManager`, respectively, in the `deployment model ScalarmDeployment` (*cf.* Listings 4 and 3).

`scalability rule CPUScalabilityRule` refers to the binary event pattern and the horizontal scaling action above, along with the `scale requirement HorizontalScaleSimulationManager` in the `requirement model ScalarmRequirement` (*cf.* Listing 8).

Listing 9: Scalarm scalability model

```
scalability model ScalarmScalability {

  non-functional event CPUAvgMetricNFEAll {
    metric condition: ScalarmModel.ScalarmMetric.CPUAvgMetricConditionAll
    violation
  }

  non-functional event CPUAvgMetricNFEAny {
    metric condition: ScalarmModel.ScalarmMetric.CPUAvgMetricConditionAny
    violation
  }

  binary event pattern CPUAvgMetricBEPAnd {
    left event: ScalarmModel.ScalarmScalability.CPUAvgMetricNFEAll
    right event: ScalarmModel.ScalarmScalability.CPUAvgMetricNFEAny
    operator: AND
  }

  horizontal scaling action HorizontalScalingSimulationManager {
    type: SCALE_OUT
    vm: ScalarmModel.ScalarmDeployment.CPUIntensiveUbuntuNorway
    internal component: ScalarmModel.ScalarmDeployment.SimulationManager
  }

  scalability rule CPUScalabilityRule {
    event: ScalarmModel.ScalarmScalability.CPUAvgMetricBEPAnd
    actions [ScalarmModel.ScalarmScalability.HorizontalScalingSimulationManager]
    scale requirements [ScalarmRequirement.HorizontalScaleSimulationManager]
  }
}
```

Listing 10 shows the metric model in textual syntax. `raw metric CPUMetric`, along with the elements referred by it, specify a raw (sensor) metric measuring CPU load. `composite metric CPUAverage`, along with the elements referred by it, specify an average CPU load metric. `composite metric context CPUAvgMetricContextAll` and `composite metric context CPUAvgMetricContextAny` specify that the average CPU load metric is instantiated in two contexts, one with a window of five minutes and one with a window of one minute, respectively. The aggregated composite metrics are instantiated as metric instances twice per virtual machine, and once per metric context.

Listing 10: Scalarm metric model

```
metric model ScalarmMetric {

```

```
3   window Win5Min {
4     window type: SLIDING
5     size type: TIME_ONLY
6     time size: 5
7     unit: ScalarmModel.ScalarmUnit.minutes
8   }
9
10  window Win1Min {
11    window type: SLIDING
12    size type: TIME_ONLY
13    time size: 1
14    unit: ScalarmModel.ScalarmUnit.minutes
15  }
16
17  schedule Schedule1Min {
18    type: FIXED_RATE
19    interval: 1
20    unit: ScalarmModel.ScalarmUnit.minutes
21  }
22
23  schedule Schedule1Sec {
24    type: FIXED_RATE
25    interval: 1
26    unit: ScalarmModel.ScalarmUnit.seconds
27  }
28
29  sensor CPUSensor {
30    configuration: 'cpu_usage;de.uniulm.omi.cloudiator.visor.sensors.
       CpuUsageSensor'
31    push
32  }
33
34  property CPUProperty {
35    type: MEASURABLE
36    sensors [ScalarmMetric.CPUSensor]
37  }
38
39  raw metric CPUMetric {
40    value direction: 0
41    layer: IaaS
42    property: ScalarmModel.ScalarmMetric.CPUProperty
43    unit: ScalarmModel.ScalarmUnit.CPUUnit
44    value type: ScalarmModel.ScalarmType.Range0_100
45  }
46
47  raw metric context CPURawMetricContext {
48    metric: ScalarmModel.ScalarmMetric.CPUMetric
49    sensor: ScalarmMetric.CPUSensor
50    component: ScalarmModel.ScalarmDeployment.SimulationManager
51    schedule: ScalarmModel.ScalarmMetric.Schedule1Sec
52    quantifier: ALL
53  }
54
55  raw metric context CPUMetricConditionContext {
56    metric: ScalarmModel.ScalarmMetric.CPUMetric
57    sensor: ScalarmMetric.CPUSensor
58    component: ScalarmModel.ScalarmDeployment.SimulationManager
59    quantifier: ANY
60  }
61
62  composite metric CPUAverage {
63    description: "Average of the CPU"
64    value direction: 1
65    layer: PaaS
66    property: ScalarmModel.ScalarmMetric.CPUProperty
67    unit: ScalarmModel.ScalarmUnit.CPUUnit
68
69    metric formula FormulaAverage {
70      function arity: UNARY
71      function pattern: MAP
```

```
72        MEAN( ScalarmModel.ScalarmMetric.CPUMetric )
73      }
74    }
75
76    composite metric context CPUAvgMetricContextAll {
77      metric: ScalarmModel.ScalarmMetric.CPUAverage
78      component: ScalarmModel.ScalarmDeployment.SimulationManager
79      window: ScalarmModel.ScalarmMetric.Win5Min
80      schedule: ScalarmModel.ScalarmMetric.Schedule1Min
81      composing metric contexts [ScalarmModel.ScalarmMetric.CPURawMetricContext]
82      quantifier: ALL
83    }
84
85    composite metric context CPUAvgMetricContextAny {
86      metric: ScalarmModel.ScalarmMetric.CPUAverage
87      component: ScalarmModel.ScalarmDeployment.SimulationManager
88      window: ScalarmModel.ScalarmMetric.Win1Min
89      schedule: ScalarmModel.ScalarmMetric.Schedule1Min
90      composing metric contexts [ScalarmModel.ScalarmMetric.CPURawMetricContext]
91      quantifier: ANY
92    }
93
94    metric condition CPUMetricCondition {
95      context: ScalarmModel.ScalarmMetric.CPUMetricConditionContext
96      threshold: 80.0
97      comparison operator: >
98    }
99
100   metric condition CPUAvgMetricConditionAll {
101     context: ScalarmModel.ScalarmMetric.CPUAvgMetricContextAll
102     threshold: 50.0
103     comparison operator: >
104   }
105
106   metric condition CPUAvgMetricConditionAny {
107     context: ScalarmModel.ScalarmMetric.CPUAvgMetricContextAny
108     threshold: 80.0
109     comparison operator: >
110   }
111 }
```

8.1 Interplay with Executionware

In order to enact the scalability rules, the Executionware provides a monitoring and adaptation engine for cross-cloud applications. In PaaSage, this monitoring and adaptation engine is Axe [11]. In particular, the Executionware configures the monitoring probes based on the specified metrics and evaluates the specified scalability rules. If a metric condition is violated, the Executionware enacts the specified scaling action (*e.g.*, scale-out), which may include the provisioning of vm instances, the deployment of component instances, and the wiring of these (see Section 6.1). Life-cycle handlers attached to the specified communications can wire existing component instances by reconfiguring them. The interested reader may refer to [11] for a detailed description of how the Executionware enacts scaling actions.

9 Organisations

The organisation package of the CAMEL metamodel is based on the organisation subset of CERIF [19]. CERIF is an EU standard[18] for research information. In particular, the organisa-

[18] http://cordis.europa.eu/cerif/

tion package of the CAMEL reuses the concepts from CERIF for specifying organisations, users, and roles. In the following, we exemplify the main concepts in the organisation package.

Assume that we have to specify the organisation model for the Scalarm use case. Listing 11 shows this specification in textual syntax.

`organisation AGH` specifies the organisation AGH (Akademia Górniczo-Hutnicza, *i.e.*, AGH University of Science and Technology), while `user MichalOrzechowski` specifies the user Michal Orzechowski belonging to the organisation AGH and owning the application Scalarm (*cf.* Listing 2).

`role devop` specifies the role *development and operations* (devop), while `role assignment MichalOrzechowskiDevop` specifies the assignment of the role devop to the user Michal Orzechowski, which is valid from 1 March 2016 to 28 February 2017.

Listing 11: Scalarm organisation model

```
organisation model AGHOrganisation {

  organisation AGH {
    www: 'http://www.agh.edu.pl/en/'
    email: 'info@agh.edu.pl'
  }

  user MichalOrzechowski {
    first name: Michal
    last name: Orzechowski
    email: 'morzech@agh.edu.pl'
    password: '************'
  }

  role devop

  role assignment MichalOrzechowskiDevop {
    start: 2016-03-01
    end: 2017-02-28
    assigned on: 2016-02-29
    user: AGHOrganisation.morzech
    role: ScalarmModel.AGHOrganisation.devop
  }
}
```

10 Providers

The provider package of the CAMEL metamodel is based on Saloon [34,35,36]. Saloon is a tool-supported DSL for specifying the features of cloud providers and matching them with requirements by leveraging feature models [4] and ontologies [18]. In the following, we exemplify the main concepts in the provider package.

Assume that we have to specify the provider model for the Scalarm use case. Listing 12 shows an excerpt of the provider model for a SINTEF private cloud specified using the CAMEL textual syntax.

`root feature SINTEF` is the root feature and specifies the attributes and sub-features characterising SINTEF's private cloud. `attribute DeliveryModel` specifies that SINTEF provides a private cloud. `attribute ServiceModel` specifies that SINTEF provides a IaaS. `attribute Availability` specifies that the guaranteed availability of SINTEF's private cloud is 95%. `attribute Driver` specifies that the provider uses an OpenStack Nova API. `attribute EndPoint` specifies the endpoint of the SINTEF's OpenStack Nova API.

`feature VM` is a sub-feature and specifies the attributes characterising the virtual machine flavours provided by SINTEF's private cloud, such as type (`attribute VMType`), operating system (`attribute VMOS`), size of RAM (`attribute VMMemory`), size of storage (`attribute VMStorage`), and number of CPU cores (`attribute VMCores`). Each attribute has a value type, and a unit type. For instance, `VMMemory` has `MemoryList`, a list of integer values (256, 512, 2048, etc.), as value type, and `MEGABYTES` as unit type. `feature cardinality` specifies that the feature has a cardinality between 1 and 8.

`constraints` specifies the constraints characterising SINTEF's private cloud. `implies M1LARGEMapping` is an intra-feature constraint and specifies the mapping between the assigned resources and the virtual machine flavours provided by SINTEF's private cloud. For instance, the first `attribute constraint` specifies that the size of RAM of the `M1.LARGE` virtual machine flavour is 8192 (megabytes).

Listing 12: SINTEF provider model (excerpt)

```
provider model SINTEFProvider {

  root feature SINTEF {

    attributes {

      attribute DeliveryModel {
        value: string value 'Private'
        value type: ScalarmModel.SINTEFType.StringValueType
      }

      attribute ServiceModel {
        value: string value 'IaaS'
        value type: ScalarmModel.SINTEFType.StringValueType
      }

      attribute Availability {
        unit type: PERCENTAGE
        value: string value '95'
        value type: ScalarmModel.SINTEFType.StringValueType
      }

      attribute Driver {
        value: string value 'openstack-nova'
        value type: ScalarmModel.SINTEFType.StringValueType
      }

      attribute EndPoint {
        value: string value 'https://minicloud.modelbased.net'
        value type: ScalarmModel.SINTEFType.StringValueType
      }
    }

    sub-features {

      feature VM {

        attributes {
          attribute VMType {value type: ScalarmModel.SINTEFType.VMTypeEnum}
          attribute VMOS {value type: ScalarmModel.SINTEFType.VMOSEnum}
          attribute VMMemory {unit type: MEGABYTES value type: ScalarmModel.
     SINTEFType.MemoryList}
          attribute VMStorage {unit type: GIGABYTES value type: ScalarmModel.
     SINTEFType.StorageList}
          attribute VMCores {value type: ScalarmModel.SINTEFType.CoresList}
        }

        feature cardinality {cardinality: 1 .. 8}
      }
...
```

```
49      }
50
51      feature cardinality {cardinality: 1 .. 1}
52    }
53
54    constraints {
55  ...
56      implies M1LARGEMapping {
57
58        from: ScalarmModel.SINTEFProvider.SINTEF.VM
59        to: ScalarmModel.SINTEFProvider.SINTEF.VM
60
61        attribute constraints {
62
63          attribute constraint {
64            from: ScalarmModel.SINTEFProvider.SINTEF.VM.VMType
65            to: ScalarmModel.SINTEFProvider.SINTEF.VM.VMMemory
66            from value: string value 'M1.LARGE'
67            to value: int value 8192
68          }
69
70          attribute constraint {
71            from: ScalarmModel.SINTEFProvider.SINTEF.VM.VMType
72            to: ScalarmModel.SINTEFProvider.SINTEF.VM.VMCores
73            from value: string value 'M1.LARGE'
74            to value: int value 4
75          }
76
77          attribute constraint {
78            from: ScalarmModel.SINTEFProvider.SINTEF.VM.VMType
79            to: ScalarmModel.SINTEFProvider.SINTEF.VM.VMStorage
80            from value: string value 'M1.LARGE'
81            to value: int value 80
82          }
83        }
84      }
85  ...
86    }
87  }
```

11 Security

The security package of the CAMEL metamodel is not based on existing DSLs and has been developed to enable the specification of security aspects of cross-cloud applications. It enables the specification of high-level and low-level security requirements and capabilities that can be exploited for filtering providers as well as adapting cross-cloud applications. In the following, we exemplify the main concepts in the security package.

Assume that we have to specify the security model for the Scalarm use case. Listing 11 shows this specification in textual syntax.

`domain IAM` specifies the domain of Identity & Access Management (IAM). `domain IAM_CLCPM` and `IAM_UAR` specify two sub-domains of IAM, namely Credential Life Cycle/Provision Management (CLCPM) and User Access Revocation (UAR), respectively.

`property IdentityAssurance` specifies an abstract property of identity assurance associated with the domain IAM. `security control IAM_02` specifies a security control associated with the sub-domain (CLCPM) and the property `IdentityAssurance`. Similarly, `security control IAM_11` specifies a security control associated with the sub-domain (UAR) and the

property `IdentityAssurance`. Note that these security controls are part of the set of security controls identified by the Cloud Security Alliance (CSA).[19]

`security capability SecCap` specifies a security capability associated with the security controls `IAM_02` and `IAM_11`. Finally, the `organisation model AmazonExt` refers to the security capability `SecCap`, which specifies that the Amazon provider supports this security capability.

Listing 13: Scalarm security model

```
security model ScalarmSecurity {

  domain IAM {
    name: "Identity & Access Management"
    sub-domains [ScalarmSecurity.IAM_CLCPM, ScalarmSecurity.IAM_CLCPM]
  }

  domain IAM_CLCPM {
    name: "Credential Life Cycle/Provision Management"
  }

  domain IAM_UAR {
    name: "User Access Revocation"
  }

  property IdentityAssurance {
    description: "The ability of a relying party to determine, with some level of
     certainty, that a claim to a particular identity made by some entity can be
     trusted to actually be the claimant's true, accurate and correct identity."
    type: ABSTRACT
    domain: ScalarmSecurity.IAM
  }

  security control IAM_02 {
    specification: "User access policies and procedures shall be established, and
     supporting business processes and technical measures implemented, for
     ensuring appropriate identity, entitlement, and access management for all
     internal corporate and customer (tenant) users with access to data and
     organisationally-owned or managed (physical and virtual) application
     interfaces and infrastructure network and systems components."
    domain: ScalarmSecurity.IAM
    sub-domain: ScalarmSecurity.IAM_CLCPM
    security properties [ScalarmModel.ScalarmSecurity.IdentityAssurance]
  }

  security control IAM_11 {
    specification: "Timely de-provisioning (revocation or modification) of user
     access to data and organisationally-owned or managed (physical and virtual)
     applications, infrastructure systems, and network components, shall be
     implemented as per established policies and procedures and based on user's
     change in status (e.g., termination of employment or other business
     relationship, job change or transfer). Upon request, provider shall inform
     customer (tenant) of these changes, especially if customer (tenant) data is
     used as part the service and/or customer (tenant) has some shared
     responsibility over implementation of control."
    domain: ScalarmSecurity.IAM
    sub-domain: ScalarmSecurity.IAM_UAR
    security properties [ScalarmModel.ScalarmSecurity.IdentityAssurance]
  }

  security capability SecCap {
    controls [ScalarmSecurity.IAM_02, ScalarmSecurity.IAM_11]
  }
}

requirement model ScalarmExtendedReqModel {

  security requirement AllIAMsSupported {
```

[19] https://cloudsecurityalliance.org/

```
    controls [ScalarmSecurity.IAM_02, ScalarmSecurity.IAM_11]
  }
}

organisation model AmazonExt {

  provider Amazon {
    www: 'https://aws.amazon.com/'
    email: 'contact@amazon.com'
    PaaS
    IaaS
    security capability [ScalarmModel.ScalarmSecurity.SecCap]
  }
}
```

12 Execution

The execution package of the CAMEL metamodel is not based on existing DSLs and has been developed to enable the recording of historical data about the execution of cross-cloud applications. Historical data, such as metric measurements and SLO assessments, can be used for auditing purposes as well as for optimising the CAMEL model to better exploit the available cloud infrastructures [24]. In PaaSage, the execution model is automatically manipulated by the PaaSage platform during the execution phase (see Section 2), and so it should be in the general case too. In the following, we exemplify the main concepts in the execution package.

Assume that we have to record the execution of the Scalarm use case. Listing 14 shows this specification in textual syntax.

`vm binding ScalarmVMBinding` specifies that the virtual machine instance `CoreIntensiveUbuntuNorwayInst` (*cf.* Listing 7) is bound to the execution context `EC1`.

`raw metric instance CPUMetricInstance` specifies that the metric instance `CPUMetricInstance` is an instance of the metric `CPUMetric` and is bound to the virtual machine instance `CoreIntensiveUbuntuNorwayInst` and the execution context `EC1` (*cf.* Listing 7).

`execution context EC1` specifies the current execution context. It refers to the application being executed, the deployment model of the application, the requirement group that led to this deployment model, and an indication of the total cost of application execution along with a reference to the corresponding monetary unit.

`vm measurement VM1` specifies the virtual machine measurement for the CPU metric instance. It refers to the execution context, the metric instance, the virtual machine instance (*cf.* Listing 7), the measured value (`95.0`), and the timestamp of the measurement.

Similar to the vm measurement, the `assessment A1` specifies the assessment for the CPU metric SLO. It comprises the appropriate reference, the indication that the SLO has been violated, and the timestamp of the assessment.

Listing 14: Scalarm execution model

```
metric model ScalarmMetric {

  vm binding ScalarmVMBinding {
    execution context: ScalarmExecution.EC1
    vm instance: ScalarmModel.ScalarmDeployment.CoreIntensiveUbuntuNorwayInst
  }

  raw metric instance CPUMetricInstance {
    metric: ScalarmModel.ScalarmMetric.CPUMetric
    sensor: ScalarmMetric.CPUSensor
```

```
    binding: ScalarmModel.ScalarmMetric.ScalarmVMBinding
  }
}

execution model ScalarmExecution {

  execution context EC1 {
    application: ScalarmModel.ScalarmApplication
    deployment model: ScalarmModel.ScalarmDeployment
    requirement group: ScalarmRequirement.ScalarmRequirementGroup
    total cost: 100.0
    cost unit: ScalarmModel.ScalarmUnits.Euro
  }

  vm measurement VM1 {
    execution context: ScalarmExecution.EC1
    metric instance: ScalarmMetric.RawCPUMetricInstance
    vm instance: ScalarmModel.ScalarmDeployment.CoreIntensiveUbuntuNorwayInst
    value: 95.0
    time: 2016-10-31 T 22:50
  }

  assessment A1 {
    execution context: ScalarmExecution.EC1
    measurement: ScalarmExecution.VM1
    slo: ScalarmRequirement.CPUMetricSLO
    violated
    time: 2016-10-31 T 22:50
  }
}
```

13 Locations, Units, and Types

The location package of the CAMEL metamodel is not based on existing DSLs and has been developed to enable the specification of locations. A location can be a *geographical region* (*e.g.*, Europe) or a *cloud location* (*e.g.*, Amazon EC2 eu-west-1). A geographical region can refer to a parent region, which allows for the creation of hierarchies of geographical regions (*e.g.*, continent, sub-continent, and country). Similar to the geographical region, a cloud location can refer to a parent location.

The unit package of the CAMEL metamodel is not based on existing DSLs and has been developed to enable the specification of units that are adopted by the following packages: (a) metric, where they are used to define the unit of measurement for a metric, (b) execution, where they are used to define the monetary unit for the cost of a particular application execution, and (c) the provider, where they are used to define the unit for a particular feature attribute.

The type package of the CAMEL metamodel is also based on Saloon [34,35,36]. It provides the concepts to specify value types and values used across CAMEL models (*e.g.*, integer, string, or enumeration).

The location, unit, and type models of the Scalarm use case have been omitted for brevity. The interested reader may refer to [39] for a complete description of these models.

14 Evaluation

In order to evaluate its usability and usefulness, CAMEL was exposed to different practitioners. The evaluation was performed in the context of the PaaSage use cases, namely

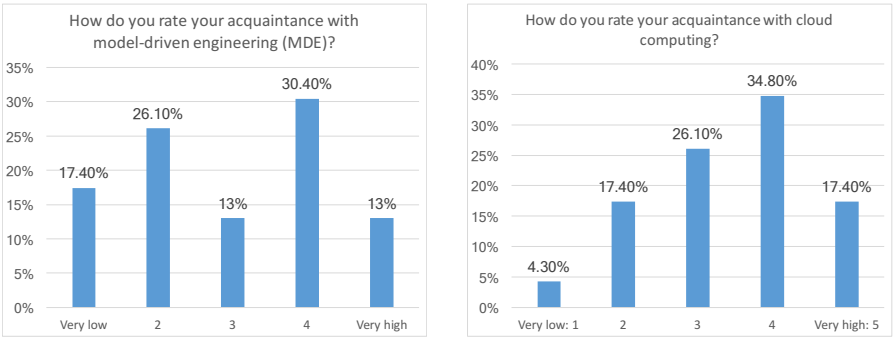
automotive simulation, flight scheduling, enterprise resource planning (ERP), public services, and eScience (the latter represented by Scalarm). The presentation of the use cases is beyond the scope of this paper. The interested reader may refer to the PaaSage website[20] for more information about them. The evaluation was based on the following steps:

1. The use case providers were familiarised with different versions of CAMEL, reported bugs, requested features, and provided general feedback to the CAMEL developers between October 2014 and September 2015.
2. The use case providers modelled their use cases with the final version of CAMEL (*i.e.*, v2015.09) between October 2015 and March 2016.
3. The use case providers assessed multiple CAMEL capabilities through an online questionnaire[21] between March and April 2016.

The evaluation procedure was based on the technology acceptance model (TAM) [7,2], where the following dimensions were considered:

– *Perceived Ease of Use (PEU)*: the degree to which a user believes that CAMEL reduces the effort in the modelling tasks.
– *Perceived Usefulness (PU)*: the degree to which a user believes that using CAMEL enhances the performance of the modelling tasks.

The participants in the questionnaire consisted of 23 individuals. Most participants were male (73.9% compared to 26.1% female). Moreover, most participants were employed as senior software engineers (52.2%), followed by junior software engineers (17.4%), and senior system administrators (17.4%). This is also reflected by the years of experience in IT, since most participants had been employed for more than 5 years (60.8%). Finally, most participants were well acquainted with MDE and cloud (see Figure 6).



(a) (b)

Figure 6: Participants' acquaintance with (a) MDE and (b) cloud computing

In general, most participants were satisfied with the completeness of CAMEL with respect to the requirements of their use cases. In particular, most participants were very satisfied with the completeness of CAMEL for specifying the deployment, organisation, and

[20] http://www.paasage.eu/use-cases
[21] https://goo.gl/forms/Fwr3Lc33SGqTJj832

requirement models (87%, 78%, and 65%, respectively). However, some participants indicated that they did not need to specify security and type models in their use cases (48% and 30%, respectively).

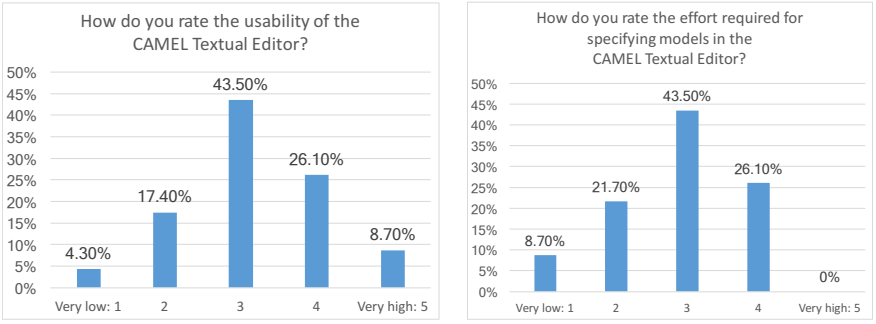
In the following, we present the evaluation of the usability and usefulness of the following CAMEL capabilities:

- CAMEL Textual Editor
- CAMEL Documentation
- CAMEL Deployment model
- CAMEL Requirement model
- CAMEL Metric model
- CAMEL Scalability model
- CAMEL Organisation model

Note that the CAMEL Textual Editor is evaluated on both the PEU and the PU, whereas the remaining capabilities are evaluated on the PU only. Also note that the provider model was not evaluated since the participants mostly reused the provider models built-in in the PaaSage platform, so the sample for the evaluation would have been too small.

14.1 Textual Editor

Many participants rated the usability of the CAMEL Textual Editor and the effort required for specifying models as neither high nor low (see Figures 7(a) and (b)).



(a) (b)

Figure 7: Participants' rating of (a) the usability of the CAMEL Textual Editor (b) the effort required for specifying models

Nevertheless, most participants rated the usefulness of the CAMEL Textual Editor's features (*e.g.*, syntax highlighting, input completion, error reporting) as very high or high (see Figure 8).

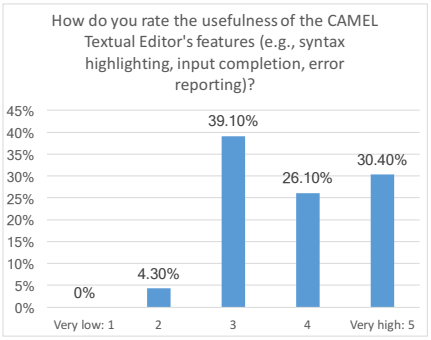
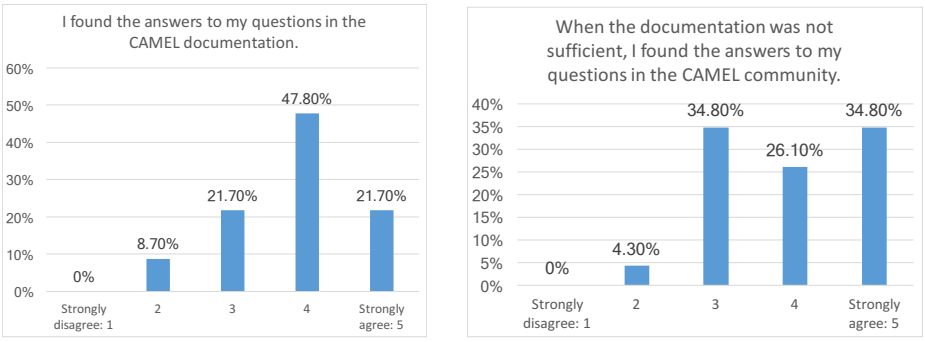


Figure 8: Participants' rating of the usefulness of the CAMEL Textual Editor's features

14.2 Documentation

Most participants indicated that they consulted the CAMEL documentation in order to specify models (73.9% indicated that the documentation was necessary or very necessary, whereas none indicated that it was not necessary).

Nevertheless, most participants found the answers to their questions in the CAMEL documentation (see Figure 9(a)). In case they did not, they found the answers in the CAMEL community (see Figure 9(b)), *e.g.*, by consulting the CAMEL developers or other CAMEL users.



(a) (b)

Figure 9: Participants' rating of (a) the CAMEL documentation and (b) the CAMEL community

14.3 Deployment model

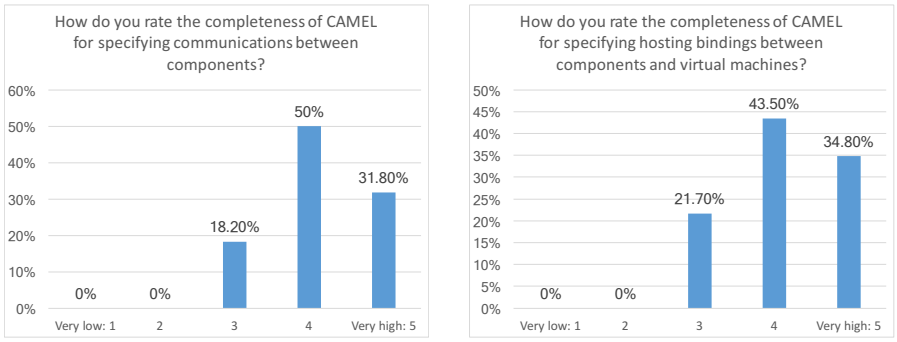
Most participants rated the completeness of CAMEL for specifying internal components as high (78.3% rated it high or very high, whereas the remaining 21.7% rated it neither high nor low) as well as life cycle management scripts (82.6% rated it high or very high, whereas

the remaining 17.4% rated it neither high nor low). Some participants suggest to support specifying instantiation workflow that is not based on the dependencies between the internal components.

Similar results apply to the completeness of CAMEL for specifying virtual machines as high (78.4% rated it high or very high, whereas the remaining 21.6% rated it neither high nor low).

Most participants were satisfied with the completeness of CAMEL for specifying communication between internal components (see Figure 10(a)). Some participants suggest to support distinguishing between TCP and UDP communications. Note that one participant did not provide any answers, which may indicate that she did not need to specify communications.

Similar results apply to the completeness of CAMEL for specifying hosting bindings between components and virtual machines (see Figure 10(b)).



(a) (b)

Figure 10: Participants' rating of the completeness of CAMEL for specifying (a) communications and (b) hosting bindings

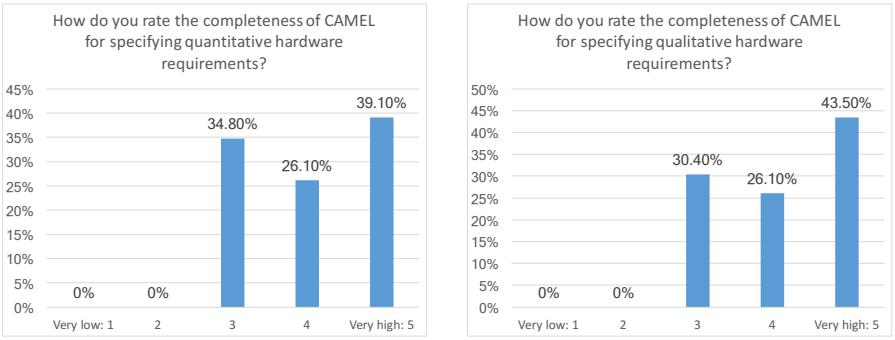
14.4 Requirement model

Most participants were satisfied with the completeness of CAMEL for specifying quantitative and qualitative hardware requirements (see Figures 11(a) and (b). Some participants suggested to extend CAMEL in order to support low-energy requirements, CPU pinning requirements, NUMA tuning requirements, and containers.

Most participants rated the completeness of CAMEL for specifying internal OS requirements as high (65.2% rated it high or very high, 30.4% neither high nor low, and 4.3%—*i.e.*, one participant—low). Three participants suggested to extend CAMEL in order to support the selection of OS versions, *e.g.*, in the form of a drop-down list.

Similar results apply to the completeness of CAMEL for specifying location requirements (47.8% rated it very high, 30.4% high, 17.4% neither high nor low, and 4.3% low).

Most participants rated the completeness of CAMEL for specifying optimisation requirements as high (78.3% rated it high or very high, whereas 8.7% rated it low) and SLO requirements (63.6% rated it high or very high).

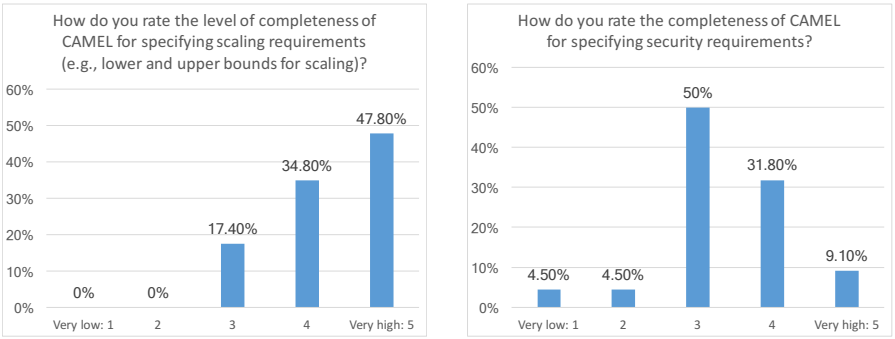


(a) (b)

Figure 11: Participants' rating of the completeness of CAMEL for specifying (a) quantitative and (b) qualitative hardware requirements

Most participants were satisfied with the completeness of CAMEL for specifying scalability requirements (see Figure 12(a)).

However, many participants rated the completeness of CAMEL for specifying security requirements as neither high nor low (see Figure 12(b)).



(a) (b)

Figure 12: Participants' rating of the completeness of CAMEL for specifying (a) scaling and (b) security requirements

Finally, most participants rated the completeness of CAMEL for specifying provider requirements as high (34.8% rated it very high, 26.1% high, 30.4% neither high nor low, and 8.7% low). Some participants suggested to extend CAMEL in order to support the selection of providers, *e.g.*, in the form of a drop-down list.

14.5 Metric model

Most participants were satisfied with the completeness of CAMEL for specifying metric contexts and metric conditions (see Figure 13).

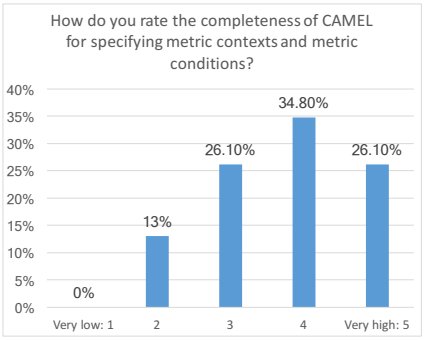


Figure 13: Participants' rating of the completeness of CAMEL for specifying metric contexts and metric conditions

14.6 Scalability model

Most participants were satisfied with the completeness of CAMEL for specifying events and (composite) event patterns (see Figure 14).

Even more encouraging results apply to the completeness of CAMEL for specifying scaling actions (see Figure 15(a)) and scalability rules (see Figure 15(b)).

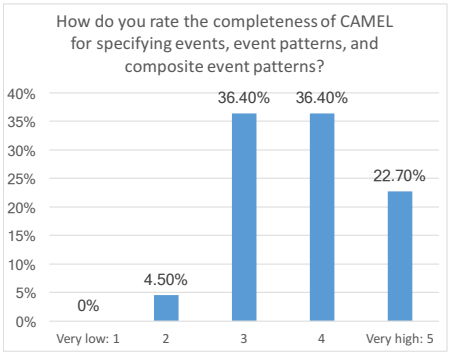
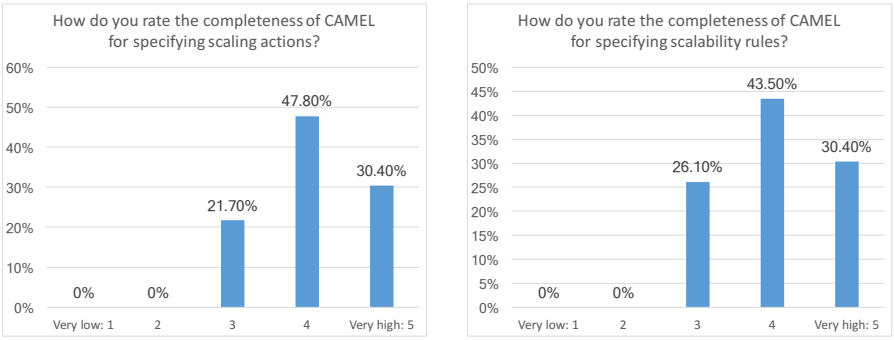


Figure 14: Participants' rating of the completeness of CAMEL for specifying events and (composite) event patterns

14.7 Organisation model

Since four of the participants did not have to specify organisation models (*e.g.*, they reused existing organisation models), only 19 out of 23 participants provided answers to questions on these models.

Most of the 19 participants rated the completeness of CAMEL for specifying users as high (76.2% rated it high or very high, and 4.8% low) as well as role assignments (57.1%



(a) (b)

Figure 15: Participants' rating of the completeness of CAMEL for specifying (a) scaling actions and (b) scalability rules

rated it high or very high) and permissions (57.9% rated it high or very high, and 42.1% neither high nor low).

Most participants also indicated that the basic roles of admin, DevOps and business were sufficient to group most of the users in their organisation (72.8% rated them high or very high).

Concerning access credentials (to their IaaS providers), most participants indicated that they do not feel comfortable with providing them in organisation models (see blue bars in Figure 16).

Nevertheless, most participants indicated that they would be comfortable to do so if they were guaranteed that the platform handles access credentials securely (see orange bars in Figure 16).

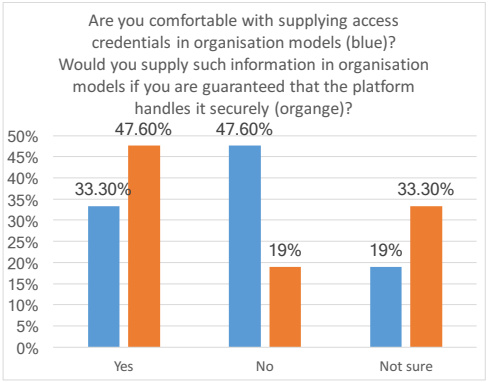


Figure 16: Participants' view on access credentials

Some participants suggested to extend CAMEL in order to support granting or denying permissions on particular types of models.

14.8 Analysis

Based on the answers to the questionnaire, we conclude that CAMEL and the CAMEL Textual Editor satisfy the usability and usefulness requirements from the domain to a high degree. Figure 17 shows a summary of the received feedback in terms of five dimensions related to usability and usefulness, namely: effectiveness, understandability, satisfaction, learnability, and integrability. As shown, satisfaction of CAMEL and the CAMEL Textual Editor is rated highest, while learnability is rated lowest—even though the rating itself is not low.

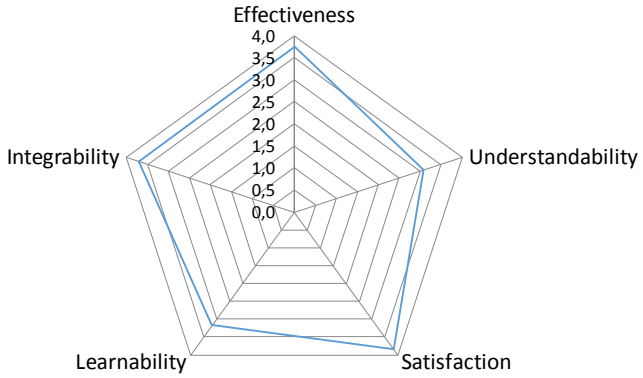


Figure 17: Usability aspects of CAMEL and the CAMEL Textual Editor on five non-functional dimensions.

The interested reader may refer to [20] for a complete description of this breakdown.

Nevertheless, we received feedback from the participants that requires particular attention. First, many participants rated the usability of the CAMEL Textual Editor and the effort required for specifying models as neither high nor low. This datum could be explained by the fact that specifying models in the CAMEL Textual Editor requires acquiring some background knowledge, *e.g.*, by reading the CAMEL documentation.

Second, many participants rated the completeness of CAMEL for specifying security requirements as neither high nor low. This datum could be explained by the fact that security requirements are not completely supported by the PaaSage platform.

Finally, most participants indicated that they do not feel comfortable with providing access credentials in organisation models. This datum was expected, given the sensitive nature of access credentials.

14.9 Threats to validity

In terms of *external validity*—*i.e.*, the extent to which the conclusions based on a study can be generalised—the selected use cases cover all the identified aspects of self-adaptive cross-cloud applications. However, extending the evaluation of CAMEL to other scenarios, environments, or even demographics may alter the findings.

Internal validity—i.e., the extent to which the conclusions based on a study are warranted—is not affected, since the data are unambiguous.

In terms of *conclusions validity—i.e.*, the degree to which conclusions about the relationship among variables based on a study are correct—some questions in the questionnaire were optional, since different use cases had different requirements and hence the corresponding CAMEL models did not necessarily cover all aspects. However, some of the participants may have selected the medium score (*i.e.*, 3) for aspects they have not used, which may have altered the conclusions.

Finally, *construct validity—i.e.*, the degree to which a test measures what it claims—is not affected, since all questions in the questionnaire were carefully prepared to cover all capabilities of CAMEL and the CAMEL Textual Editor.

15 Related Work

In the following, we compare CAMEL with related work. We distinguish between tools (*e.g.*, DevOps and cloud orchestration tools) to automate the deployment of cloud applications and languages to model cloud aspects. For the latter category, we define six comparison criteria and evaluate the languages according to these criteria. This comparison will validate our claim that CAMEL advances the state-of-the-art in modelling and execution of cloud applications.

15.1 Tools

In the cloud community, libraries such as jclouds[22] or DeltaCloud[23] provide generic APIs abstracting over the heterogeneous APIs of IaaS providers. These libraries reduce the cost and effort of deploying cloud applications and can be used by the platforms supporting CAMEL. For instance, the PaaSage platform uses jclouds.

DevOps tools such as Puppet[24] or Chef[25] rely on scripting languages to specify the deployment of cloud applications. These tools increase the automation in deploying cloud applications. However, the deployment scripts cannot be treated as deployment models, which introduces a mismatch between the deployment topology of cloud applications and the technique used to represent them.

Cloud orchestration tools such as Cloudify[26] or Apache Brooklyn[27] rely on the TOSCA [33] (see below) to specify the topologies of cloud applications along with the processes for their orchestration. These tools facilitate the provisioning, deployment, and monitoring of cloud applications across multiple cloud infrastructures [3]. However, TOSCA does not provide an instance model and hence does not support models@run-time, which makes these tools unsuitable for reasoning on the models and hence enabling self-adaptive cross-cloud applications.

[22] `http://www.jclouds.org`
[23] `http://deltacloud.apache.org/`
[24] `https://puppetlabs.com/`
[25] `http://www.opscode.com/chef/`
[26] `http://getcloudify.org/`
[27] `https://brooklyn.apache.org/`

15.2 Languages

In the research community, within the Reservoir[28] EU project, Galán et al. [16] proposed a service specification language for cloud computing platforms, which extends the DMTF's Open Virtualization Format (OVF) standard to address the specific requirements of these environments.

Within the 4CaaSt[29] EU project, Nguyen et al. [28] proposed a language to specify Blueprint Templates—a uniform abstract description for cloud service offerings that may cross different cloud computing layers, *i.e.*, infrastructure and platform.

The MODAClouds project[30] provides a family of DSLs collectively called MODACloudML. MODACloudML relies on the following three layers of abstraction: *(i)* the *cloud-enabled computation independent model* (CCIM) to describe an application and its data, *(ii)* the CPIM (as in PaaSage) to describe concerns of cloud applications in a cloud-agnostic way, and *(iii)* the CPSM (as in PaaSage) to describe concerns of cloud applications in a cloud-specific way, so that they can be provisioned and deployed on specific clouds.

The ARTIST project[31] provides the Cloud Application Modelling Language (CAML). CAML consists of an *internal* DSL [15] realised as a UML library along with UML profiles [5] rather than an *external* DSL such as CAMEL. The main rationale behind the latter stems from the goal of the ARTIST project to support the migration of existing applications to the cloud, whereby UML models are reverse-engineered and tailored to a selected cloud environment.

The ARCADIA project[32] provides a methodology and a framework to support the development of highly-distributed applications (HDAs) that are reconfigurable by design. The ARCADIA Framework [38] relies on unikernel technology in order to bundle microservices, and leverages on an extensible *context model* throughout the entire life cycle of HDAs. Similar to CAMEL, the ARCADIA Context Model [17] has multiple facets, such as the component model, the service graph model, the service deployment model, and the service run-time model.

In the standards community, the Topology and Orchestration Specification for Cloud Applications (TOSCA) [33] is a specification developed by the OASIS consortium, which provides a language for specifying the components comprising the topology of cloud applications along with the processes for their orchestration.

15.3 Comparison

In the following, we define six comparison criteria and evaluate the aforementioned languages according to these criteria. These criteria were selected to evaluate the usefulness, usability, and self-adaptation support of the lanugages. In particular, the abstract syntax and aspect coverage, delivery model support, and models@run-time support reflect the usefulness of the language; the concrete syntax and integration level reflect the usability; and models@run-time support also reflects the self-adaptation support.

[28] `http://www.reservoir-fp7.eu/`
[29] `http://www.4caast.eu`
[30] `http://www.modaclouds.eu/`
[31] `http://www.artist-project.eu/`
[32] `http://www.arcadia-framework.eu`

Abstract syntax. The abstract syntax of a language describes the set of concepts, their attributes, and their relations, as well as the rules for combining these concepts to specify valid statements that conform to this abstract syntax. The abstract syntax can be defined using formalisms that provide different capabilities. For instance, XML Schema are more suitable for tree-based structures, while MOF-based formalisms are more suitable for graph-based structures and offer better tool support and better integration with constraint languages such as OCL. This criterion is used to identify which formalisms are used by a language. The values for this criterion are "XML Schema" and "MOF".[33]

Concrete syntax. The concrete syntax of a language describes the textual or graphical notation that renders the concepts of the abstract syntax, their attributes, and their relations. The concrete syntax can be defined using notations that provide a trade-off between the intuitiveness and the effectiveness of the syntax. For instance, a textual syntax may be less intuitive but more effective than a corresponding graphical syntax. This criterion is used to identify which notations are supported by a language. The values for this criterion can be "XML", "txt" (textual), and "gra" (graphical).

Aspect coverage. A language may cover multiple aspects within the same domain or across multiple domains. For instance, in CAMEL we specify the life cycle of cross-cloud applications using 11 aspects, namely deployment, requirement, location, metric, scalability, provider, organisation, security, execution, unit, and type. This criterion reflects how many of these aspects are covered by a language. The values for this criterion can be "low" if it covers at most three aspects, "medium" if it covers at most six aspects, and "high" otherwise.

Integration level. A language that covers multiple aspects may provide different levels of integration across these aspects, especially when these aspects include similar or equivalent concepts. The integration solution has to: (a) join equivalent concepts and separate similar concepts into respective sub-concepts; (b) homogenise the remaining concepts so that they are defined at the same level of granularity; (c) enforce a uniform formalism and notation for the abstract and concrete syntaxes; and (d) enforce the consistency, correctness, and integrity of the models. Each of these steps is a precondition to the following step and requires an increasing amount of effort. This criterion reflects how many of these steps have been adopted to integrate the sub-languages. The values for this criterion can be "low" if the sub-languages were integrated following only step (a), "medium" if they were integrated following steps (a) and (b), "high" if they were integrated following all steps, and "N/A" if they were not integrated. In the last case, each sub-language covers one aspect and is independent from the other sub-languages. This independence leads to the following disadvantages: (a) it raises the complexity of the language, since each sub-language has its own abstract and concrete syntax; (b) it steepens the learning curve and increases the modelling effort for the same reason; (d) it leads to the duplication of modelling for similar or equivalent concepts; (e) it leads to the manual validation of cross-aspect dependencies.

Delivery model support. A cross-cloud application may exploit any of the cloud delivery models, namely IaaS and PaaS. A language for specifying the life cycle of cross-cloud applications should support concepts for each of these cloud delivery models. This criterion reflects how many of these delivery models are supported by a language. The values for this criterion can be "IaaS" and "PaaS".

[33] Abstract syntaxes defined in Ecore fall into this category.

Models@run-time support. Models@run-time [6] provides an abstract representation of the underlying running system, whereby a modification to the model is enacted on-demand in the system, and a change in the system is automatically reflected in the model. Models@run-time can be implemented using the type-instance pattern [1], which facilitates reusability and abstraction. For instance, we implemented the type-instance pattern in two aspects, namely deployment and metric. In the case of deployment, this allows to automatically adapt the component- and virtual machine instances in the deployment model based on scalability rules (*e.g.*, scale out a Scalarm service along with the underlying virtual machine). In the case of monitoring, this allows to automatically populate the metric model with metric instances (*e.g.*, CPU load measurements of the virtual machine hosting the Scalarm service). This criterion reflects how many of these aspects implement the type-instance pattern. The values for this criterion can be "deployment" and "metric".

15.4 Analysis

Table 1 shows the comparison of the languages based on the criteria above. CAMEL scores best in all criteria apart from the last one.

Language	Abstract Syntax	Concrete Syntax	Aspect Coverage	Integration Level	Delivery Model Support	Models@run-time Support
Reservoir OVF Extension	XML Schema	XML	low	N/A	IaaS	N/A
4CaaSt Blueprint Template	XML Schema	XML	low	N/A	IaaS, PaaS	N/A
ModaCloudML	MOF	XML, gra, txt	medium	low	IaaS, PaaS	deployment
CAML	MOF	gra	medium	medium	IaaS	N/A
ARCADIA Context Model	XML Schema	XML	high	medium	IaaS	deployment
TOSCA	XML Schema	XML, txt	medium	medium	IaaS, PaaS	N/A
CAMEL	MOF	XML, gra, txt	high	high	IaaS	deployment, metric

Table 1: The cloud language comparison table

Compared to CAMEL, the Reservoir OVF Extension and the 4CaaSt Blueprint Templates do not cover the multiple aspects necessary for modelling and especially executing cross-cloud applications.

MODACloudML and CAMEL achieve similar goals, but with different approaches: MODACloudML is a family of loosely coupled DSLs, while CAMEL is a standalone language. CAMEL has the advantage of providing a uniform abstract and concrete syntax, which resulted from a process of coupling and homogenising multiple DSLs [29]. However, ModaCloudML supports both IaaS and PaaS while CAMEL only supports the former. CAMEL will be extended to include support PaaS in future work.

CAML achieves a subset of the goals of CAMEL. CAML does not support aspects of execution, while CAMEL provides full support for models@run-time.

The ARCADIA Context Model is less expressive than CAMEL with respect to specifying complex conditions on composite metrics. However, the ARCADIA Context Model is more expressive than CAMEL with respect to specifying adaptation actions, and provides concepts for specifying unikernel and microservices aspects. CAMEL could possibly be extended to support these aspects.

TOSCA supports the specification of types and templates, but not instances, in deployment models. CAMEL, in contrast, supports the specification of types, templates, and instances. Note that CAMEL provides built-in types such as vm, internal component, communication, and hosting, while TOSCA offers similar types in a library of reusable types and allows

to define arbitrary types. In its current form, TOSCA can only be used at design-time, while CAMEL can be used at both design-time and run-time.

As part of the joint standardisation effort of MODAClouds, PaaSage, and ARCADIA, SINTEF presented the models@run-time approach to the TOSCA technical committee (TC) and proposed to form an ad hoc group to investigate how TOSCA could be extended to support this approach. The TC welcomed this proposal and approved the formation of the Instance Model Ad Hoc group in October 2015. The group is currently co-led by Alessandro Rossini from SINTEF and Derek Palma from Vnomic. The work performed in this group will guarantee that the contribution of CAMEL will partly be integrated into the standard.

16 Conclusion and Future Work

CAMEL allows to specify multiple aspects of cross-cloud applications, such as provisioning, deployment, service level, monitoring, scalability, providers, organisations, users, roles, security, and execution. It supports models@run-time, which enables reasoning on CAMEL models and hence managing self-adaptive cross-cloud applications.

In this paper, we described the design and implementation of CAMEL, with emphasis on the integration of heterogeneous DSLs that cover different aspects of self-adaptive cross-cloud applications. Moreover, we provided a real-world running example to illustrate how to specify models in a concrete textual syntax and how to dynamically adapt these models during the application life cycle. Finally, we provided an evaluation of CAMEL's usability and usefulness, based on TAM.

In the future, we will continue to develop CAMEL iteratively. In particular, we will adapt and extend the capabilities of CAMEL to the changing requirements. In this respect, the developers will provide feedback on whether the concepts in CAMEL are adequate to design and implement their components. Similarly, the users will provide feedback on whether the concepts in CAMEL are satisfactory for modelling the use cases. In addition to the PaaSage project, CAMEL has been adopted by the CACTOS[34], CloudSocket[35], and MUSA[36] projects. This will guarantee the further development and validation of CAMEL in a wide variety of cloud computing scenarios.

In addition, CAMEL models that conform to an old version of CAMEL often have to be migrated to conform to its current version. In the future, we would like to integrate a solution for the challenge of maintaining multiple versions and automatically migrating CAMEL models [29] based on CDO and Edapt.

Finally, we will contribute to the Instance Model Ad Hoc group of TOSCA so that the contribution of CAMEL will partly be integrated into the standard.

Acknowledgements. The research leading to these results was supported by the European Commission's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 317715 (PaaSage) and the European Commission's Framework Programme Horizon 2020 (ICT-07-2014) under grant agreement numbers 644690 (CloudSocket) and 645372 (ARCADIA). The authors would like to thank the use case providers and component developers in the above projects for the constructive feedback on CAMEL. Michal Orzechowski is also grateful to AGH University of Science and Technology for their support under grant number 15.11.230.212.

[34] `http://www.cactosfp7.eu/`

[35] `https://www.cloudsocket.eu`

[36] `http://www.musa-project.eu/`

References

1. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Transactions on Modeling and Computer Simulation **12**(4), 290–321 (2002). DOI 10.1145/643120.643123
2. Bagozzi, R.P., Davis, F.D., Warshaw, P.R.: Development and Test of a Theory of Technological Learning and Usage. Human Relations **45**, 659–686 (1992). DOI 10.1177/001872679204500702
3. Baur, D., Seybold, D., Griesinger, F., Tsitsipas, A., Hauser, C.B., Domaschka, J.: Cloud Orchestration Features: Are Tools Fit for Purpose? In: I. Raicu, O.F. Rana, R. Buyya (eds.) UCC 2015: 8th IEEE/ACM International Conference on Utility and Cloud Computing, pp. 95–101. IEEE Computer Society (2015). DOI 10.1109/UCC.2015.25
4. Benavides, D., Segura, S., Cortés, A.R.: Automated analysis of feature models 20 years later: A literature review. Inf. Syst. **35**(6), 615–636 (2010). DOI 10.1016/j.is.2010.01.001
5. Bergmayr, A., Troya, J., Neubauer, P., Wimmer, M., Kappel, G.: UML-based Cloud Application Modeling with Libraries, Profiles, and Templates. In: R.F. Paige, J. Cabot, M. Brambilla, L.M. Rose, J.H. Hill (eds.) CloudMDE 2014: 2nd International Workshop on Model-Driven Engineering on and for the Cloud, *CEUR Workshop Proceedings*, vol. 1242, pp. 56–65. CEUR (2014). URL `http://ceur-ws.org/Vol-1242/paper7.pdf`
6. Blair, G., Bencomo, N., France, R.: Models@run.time. IEEE Computer **42**(10), 22–27 (2009). DOI 10.1109/MC.2009.326
7. Davis, F.D.: Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. MIS Quarterly **13**(3), 319–340 (1989). DOI 10.2307/249008. URL `http://www.jstor.org/stable/249008`
8. Domaschka, J., Baur, D., Seybold, D., Griesinger, F.: Cloudiator: A Cross-Cloud, Multi-Tenant Deployment and Runtime Engine. In: SummerSOC 2015: 9th Workshop and Summer School On Service-Oriented Computing 2015 (2015)
9. Domaschka, J., Griesinger, F., Baur, D., Rossini, A.: Beyond Mere Application Structure: Thoughts on the Future of Cloud Orchestration Tools. Procedia Computer Science **68**, 151–162 (2015). DOI 10.1016/j.procs.2015.09.231. Cloud Forward 2015: 1st International Conference on Cloud Forward: From Distributed to Complete Computing
10. Domaschka, J., Kritikos, K., Rossini, A.: Towards a Generic Language for Scalability Rules. In: G. Ortiz, C. Tran (eds.) Advances in Service-Oriented and Cloud Computing—Workshops of ESOCC 2014, *Communications in Computer and Information Science*, vol. 508, pp. 206–220. Springer (2015). DOI 10.1007/978-3-319-14886-1_19
11. Domaschka, J., Seybold, D., Griesinger, F., Baur, D.: Axe: A Novel Approach for Generic, Flexible, and Comprehensive Monitoring and Adaptation of Cross-Cloud Applications. In: A. Celesti, P. Leitner (eds.) Advances in Service-Oriented and Cloud Computing—Workshops of ESOCC 2015, *Communications in Computer and Information Science*, vol. 567, pp. 184–196. Springer (2016). DOI 10.1007/978-3-319-33313-7_14
12. Ferry, N., Chauvel, F., Rossini, A., Morin, B., Solberg, A.: Managing multi-cloud systems with CloudMF. In: A. Solberg, M.A. Babar, M. Dumas, C.E. Cuesta (eds.) NordiCloud 2013: 2nd Nordic Symposium on Cloud Computing and Internet Technologies, pp. 38–45. ACM (2013). DOI 10.1145/2513534.2513542
13. Ferry, N., Rossini, A., Chauvel, F., Morin, B., Solberg, A.: Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In: L. O'Conner (ed.) CLOUD 2013: 6th IEEE International Conference on Cloud Computing, pp. 887–894. IEEE Computer Society (2013). DOI 10.1109/CLOUD.2013.133
14. Ferry, N., Song, H., Rossini, A., Chauvel, F., Solberg, A.: CloudMF: Applying MDE to Tame the Complexity of Managing Multi-Cloud Applications. In: R. Bilof (ed.) UCC 2014: 7th IEEE/ACM International Conference on Utility and Cloud Computing, pp. 269–277. IEEE Computer Society (2014). DOI 10.1109/UCC.2014.36
15. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
16. Galán, F., Sampaio, A., Rodero-Merino, L., Loy, I., Gil, V., Vaquero, L.M.: Service specification in cloud environments based on extensions to open standards. In: J. Bosch, S. Clarke (eds.) COMSWARE 2009: 4th International Conference on COMmunication System softWAre and MiddlewaRE, pp. 19:1–19:12. ACM (2009). DOI 10.1145/1621890.1621915
17. Gouvas, P., Rossini, A., Chauvel, F., Zafeiropoulos, A., Fotopoulou, E., Vassilakis, C., Repetto, M., Tsagkaris, K., Koutsouris, N., Demesticha, K., Kovaci, S., Carella, G.: D2.2—Definition of the ARCADIA Context Model. Arcadia project deliverable (2015)
18. Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge Acquisition **5**(2), 199–220 (1993). DOI 10.1006/knac.1993.1008

19. Jeffery, K., Houssos, N., Jörg, B., Asserson, A.: Research information management: the CERIF approach. IJMSO **9**(1), 5–14 (2014). DOI 10.1504/IJMSO.2014.059142
20. Kapitsaki, G., Achilleos, A., Papoutsakis, M.: D1.7.2—Results of Evaluation of developers related to the use cases. PaaSage project deliverable (2016)
21. Kritikos, K., Domaschka, J., Rossini, A.: SRL: A Scalability Rule Language for Multi-Cloud Environments. In: J.E. Guerrero (ed.) CloudCom 2014: 6th IEEE International Conference on Cloud Computing Technology and Science, pp. 1–9. IEEE Computer Society (2014). DOI 10.1109/CloudCom.2014.170
22. Kritikos, K., Kirkham, T., Kryza, B., Massonet, P.: Security Enforcement for Multi-Cloud Platforms—The Case of PaaSage. Procedia Computer Science **68**, 103–115 (2015). DOI 10.1016/j.procs.2015.09.227. Cloud Forward 2015: 1st International Conference on Cloud Forward: From Distributed to Complete Computing
23. Kritikos, K., Korozi, M., Kryza, B., Kirkham, T., Leonidis, A., Magoutis, K., Massonet, P., Ntoa, S., Papaioannou, A., Papoulas, C., Sheridan, C., Zeginis, C.: D4.1.1—Prototype Metadata Database and Social Network. PaaSage project deliverable (2014)
24. Kritikos, K., Magoutis, K., Plexousakis, D.: Towards Knowledge-Based Assisted IaaS Selection. In: CloudCom 2016: 8th IEEE International Conference on Cloud Computing Technology and Science. IEEE Computer Society (2016)
25. Król, D., Kitowski, J.: Self-scalable services in service oriented software for cost-effective data farming. Future Generation Comp. Syst. **54**, 1–15 (2016). DOI 10.1016/j.future.2015.07.003
26. Kühne, T.: Matters of (meta-)modeling. Software and Systems Modeling **5**(4), 369–385 (2006). DOI 10.1007/s10270-006-0017-9
27. Mell, P., Grance, T.: The NIST Definition of Cloud Computing. Special Publication 800-145, National Institute of Standards and Technology (2011)
28. Nguyen, D.K., Lelli, F., Taher, Y., Parkin, M., Papazoglou, M.P., van den Heuvel, W.: Blueprint Template Support for Engineering Cloud-Based Services. In: W. Abramowicz, I.M. Llorente, M. Surridge, A. Zisman, J. Vayssière (eds.) ServiceWave 2011: 4th European Conference Towards a Service-Based Internet, *Lecture Notes in Computer Science*, vol. 6994, pp. 26–37. Springer (2011). DOI 10.1007/978-3-642-24755-2_3
29. Nikolov, N., Rossini, A., Kritikos, K.: Integration of DSLs and Migration of Models: A Case Study in the Cloud Computing Domain. Procedia Computer Science **68**, 53–66 (2015). DOI 10.1016/j.procs.2015.09.223. Cloud Forward 2015: 1st International Conference on Cloud Forward: From Distributed to Complete Computing
30. Object Management Group: Unified Modeling Language Specification (2011). `http://www.omg.org/spec/UML/2.4.1/`
31. Object Management Group: Object Constraint Language (2014). `http://www.omg.org/spec/OCL/2.4/`
32. Object Management Group: XML Metadata Interchange Specification (2014). `http://www.omg.org/spec/XMI/2.4.2/`
33. Palma, D., Spatzier, T.: Topology and Orchestration Specification for Cloud Applications (TOSCA). Tech. rep., Organization for the Advancement of Structured Information Standards (OASIS) (2013). URL `http://docs.oasis-open.org/tosca/TOSCA/v1.0/cos01/TOSCA-v1.0-cos01.pdf`
34. Quinton, C., Haderer, N., Rouvoy, R., Duchien, L.: Towards multi-cloud configurations using feature models and ontologies. In: MultiCloud 2013: International Workshop on Multi-cloud Applications and Federated Clouds, pp. 21–26. ACM (2013). DOI 10.1145/2462326.2462332
35. Quinton, C., Romero, D., Duchien, L.: Cardinality-based feature models with constraints: a pragmatic approach. In: T. Kishi, S. Jarzabek, S. Gnesi (eds.) SPLC 2013: 17th International Software Product Line Conference, pp. 162–166. ACM (2013). DOI 10.1145/2491627.2491638
36. Quinton, C., Rouvoy, R., Duchien, L.: Leveraging Feature Models to Configure Virtual Appliances. In: CloudCP 2012: 2nd International Workshop on Cloud Computing Platforms, pp. 21–26. ACM (2012). DOI 10.1145/2168697.2168699
37. Rossini, A.: Cloud Application Modelling and Execution Language (CAMEL) and the PaaSage Workflow. In: A. Celesti, P. Leitner (eds.) Advances in Service-Oriented and Cloud Computing—Workshops of ESOCC 2015, *Communications in Computer and Information Science*, vol. 567, pp. 437–439. Springer (2016). DOI 10.1007/978-3-319-33313-7
38. Rossini, A., Chauvel, F., Gouvas, P., Zafeiropoulos, A., Vassilakis, C., Fotopoulou, E.: D3.1a—Smart Controller Reference Implementation. Arcadia project deliverable (2016)
39. Rossini, A., Kritikos, K., Nikolov, N., Domaschka, J., Griesinger, F., Seybold, D., Romero, D.: D2.1.3—CAMEL Documentation. PaaSage project deliverable (2015)
40. Rossini, A., Rutle, A., Lamo, Y., Wolter, U.: A formalisation of the copy-modify-merge approach to version control in MDE. Journal of Logic and Algebraic Programming **79**(7), 636–658 (2010). DOI 10.1016/j.jlap.2009.10.003