CAMEL Documentation v2015.9

Alessandro Rossini*, Kiriakos Kritikos†, Nikolay Nikolov,
Jörg Domaschka, Frank Griesinger, Daniel Seybold,
Daniel Romero, Michal Orzechowski

7th July 2016

*alessandro.rossini@sintef.no
†kritikos@ics.forth.gr

Executive Summary

Cloud computing provides a ubiquitous networked access to a shared and virtualised pool of computing capabilities that can be provisioned with minimal management effort [19]. Cloud-based applications are applications that are deployed on cloud infrastructures and delivered as services. PaaSage aims to facilitate the modelling and execution of cloud-based applications by leveraging upon model-driven engineering (MDE) techniques and methods, and by exploiting multiple cloud infrastructures.

MDE is a branch of software engineering that aims at improving the productivity, quality, and cost-effectiveness of software development by shifting the paradigm from code-centric to model-centric. Models enable the abstraction from the implementation details of heterogeneous cloud services, while model transformations facilitate the automatic generation of the source code that exploits these services. This approach, which is commonly summarised as "model once, generate anywhere", is particularly relevant when it comes to the modelling and execution of cross-cloud applications (*i.e.*, applications that can be deployed across multiple private, public, or hybrid cloud infrastructures). This solution allows exploiting the peculiarities of each cloud service and hence optimising performance, availability, and cost of the applications.

CAMEL integrates and extends existing domain-specific languages (DSLs), namely the Cloud Modelling Language (CloudML), Saloon, and the organisation part of CERIF. In addition, CAMEL integrates new DSLs developed within the project, such as the Scalability Rule Language (SRL).

CAMEL enables PaaSage users to specify multiple aspects of cross-cloud applications, such as provisioning and deployment, service-level objectives, metrics, scalability rules, providers, organisations, users, roles, security controls, execution contexts, execution histories, etc.

In this document, we describe the modelling concepts, their attributes and their relations, as well as the rules for combining these concepts to specify valid CAMEL models. Moreover, we exemplify how to specify models through a textual editor as well as how to programmatically manipulate and persist them through Java APIs.

Contents

1 Introduction

MDE is a branch of software engineering that aims at improving 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 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 exploiting the peculiarities of each cloud service and hence optimising performance, availability, and cost of the applications.

Models can be specified using general-purpose languages like the Unified Modeling Language (UML) [22]. 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. In order to cover the necessary aspects of the modelling and execution of cross-cloud applications, PaaSage adopts the Cloud Application Modelling and Execution Language (CAMEL). CAMEL integrates and extends existing DSLs, namely Cloud Modelling Language (CloudML) [9, 7, 8], Saloon [25, 24, 26], and the organisation part of CERIF [13]. In addition, CAMEL integrates new DSLs developed within the project, such as the Scalability Rule Language (SRL) [15, 5].

CAMEL enables PaaSage users to specify multiple aspects of cross-cloud applications, such as provisioning and deployment, service-level objectives, metrics, scalability rules, providers, organisations, users, roles, security controls, execution contexts, execution histories, etc.

CAMEL supports models@run-time [4], 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. By exploiting models at both design- and run-time, and by allowing both direct and programmatic manipulation of models, CAMEL enable self-adaptive cross-cloud applications (*i.e.*, cross-cloud applications that automatically adapt to changes in the environment, requirements, and usage).

Structure of the document

The remainder of the document is organised as follows. Section 2 describes the role of CAMEL models in the PaaSage workflow. Section 3 presents some technologies used to design and implement CAMEL. Sections 6-17 present the various packages of the CAMEL metamodel along with corresponding sample models in concrete syntax. Section 18 exemplifies the usage of Java APIs to

programmatically manipulate and persist models. Finally, Section 19 compares the proposed approach with related work, while Section 20 draws conclusions and outlines plans for future work.

2 CAMEL and the PaaSage workflow

In order to facilitate the integration across the components managing the life cycle of cross-cloud applications, PaaSage leverages upon CAMEL models cross-cutting the aforementioned aspects. These models are progressively refined throughout the *modelling*, *deployment*, and *execution* phases of the PaaSage workflow [27].

Figure 1 shows the PaaSage workflow. The white trapezes represent the activities performed by the PaaSage 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.

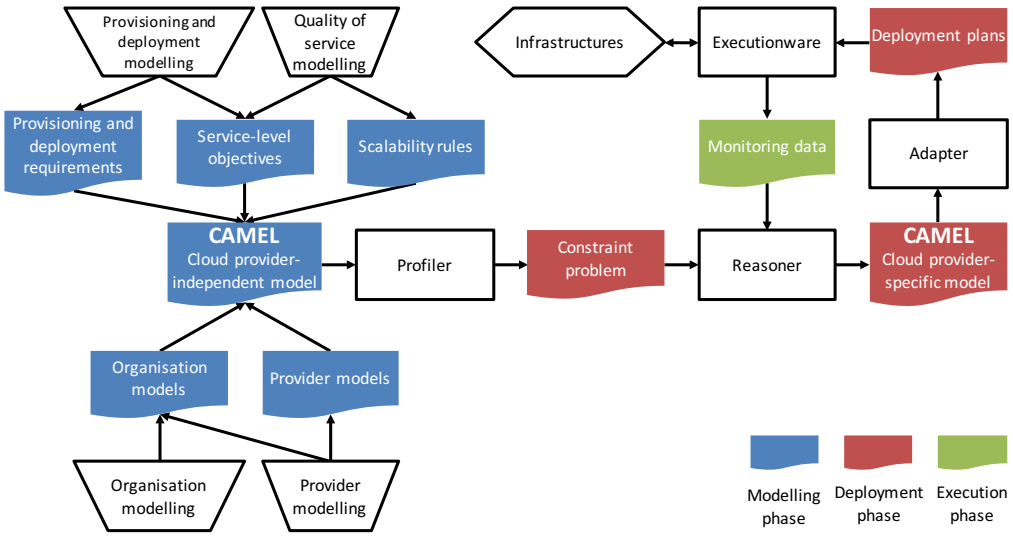

Figure 1: CAMEL models in the PaaSage workflow

In the remainder of the paper, we adopt the Scalarm[1][17] use case as a running example to exemplify how to specify CAMEL models. Scalarm is a scientific platform enabling users to execute *data farming experiments* on heterogeneous computing resources and analyse data farming experiment results.

[1] `http://www.scalarm.com/`

Modelling phase: The PaaSage 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. For instance, a PaaSage user could specify a CPIM of Scalarm.

Figure 2(a) shows the CPIM in graphical syntax. It consists of an Experiment Manager (represented by ExpMan)—a component responsible for coordinating the execution of data farming experiments—hosted by a GNU/Linux virtual machine (represented by Linux). Moreover, the Experiment Manager communicates with a Simulation Manager (represented by SimMan)—a component responsible for executing simulations that comprise the data farming experiments—hosted by a GNU/Linux virtual machine in a data centre in Norway. Finally, the Experiment Manager must have a response time below 100 ms.

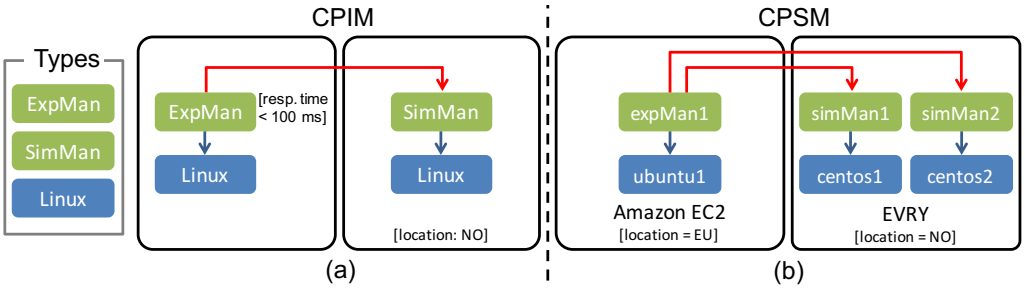

Figure 2: Sample CAMEL models: (a) CPIM; (b) CPSM

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. For instance, the Profiler could match the CPIM of Scalarm with the profile of cloud providers, identify EVRY and Telenor as the only two cloud providers offering GNU/Linux virtual machines in data centres in Norway, and produce a corresponding constraint problem. Then, the Reasoner could rank EVRY as the best cloud provider to satisfy the requirements in the CPIM, and produce a corresponding CPSM.

Figure 2(b) shows the CPSM in graphical syntax. It consists of an Experiment Manager *instance*, which is hosted by a Ubuntu 14.04 virtual machine instance at Amazon EC2 in the EU. Moreover, the Experiment Manager instance communicates with two Simulation Manager instances, which are hosted by two CentOS 7 virtual machines instances at EVRY in Norway.

The Adapter component consumes the CPSM and produces a *deployment plan*, which specifies platform-specific details of the deployment.

Execution phase: The Executionware consumes the deployment plan 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 Modelling Framework (EMF)[2] along with Object Constraint Language (OCL) [21], Xtext[3], and Connected Data Objects (CDO)[4]. In this section, we outline these technologies and describe how they facilitate the implementation of the PaaSage platform described in Section 2.

3.1 Eclipse Modeling Framework (EMF)

EMF is a modelling framework that facilitates defining DSLs. EMF provides the Ecore metamodel, which allows specifying Ecore models. The CAMEL metamodel is an Ecore model that conforms to the Ecore metamodel (see Figure 3). The Ecore metamodel, in turn, is an Ecore model that conforms to itself (*i.e.*, it is reflexive).

EMF allows generating Java class hierarchy representations of the metamodels based on those definitions. The Java representations provide a set of APIs that enables the programmatic manipulation of models. In addition, EMF provides code generation facilities that can be used to automatically generate a tree-based editor, as well as frameworks such as Graphical Modeling Framework (GMF)[5] or Graphical Editing Framework (GEF)[6] to manually create a custom graphical editor.

3.2 Object Constraint Language (OCL)

EMF enables checking of cardinality constraints on properties, creating classification trees, automatically generating code, and validating the produced models

[2] `https://www.eclipse.org/modeling/emf/`
[3] `https://eclipse.org/Xtext/`
[4] `https://www.eclipse.org/cdo/`
[5] `https://www.eclipse.org/modeling/gmp/`
[6] `https://www.eclipse.org/gef/`

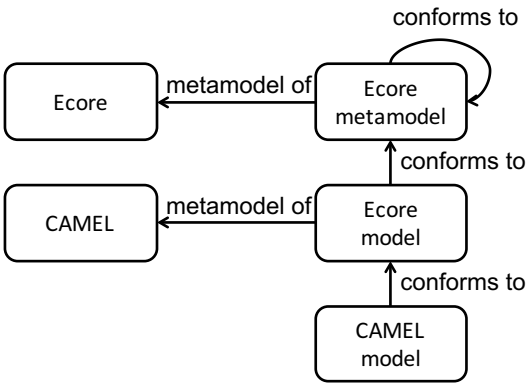

Figure 3: Ecore-based modelling stack in PaaSage

according to their metamodels. However, it lacks the expressiveness required for capturing (part of) the semantics of the domain, and hence cannot guarantee the consistency, correctness, and integrity of information in CAMEL models at both design-time and run-time.

In order to validate CAMEL models, we annotated the CAMEL metamodel with OCL constraints. OCL is a declarative language for specifying expressions such as constraints and queries on models and metamodels. In particular, Eclipse OCL[7] is a tool-supported implementation of the OCL declarative language, compatible with EMF. The OCL constraints are attached to the elements of the CAMEL metamodel and evaluated on the instances of these elements. By navigating the cross-references across models, these OCL constraints guarantee the consistency, correctness, and integrity of CAMEL models.

3.3 XText

EMF enables the automatic generation of a tree-based editor for specifying models. Nevertheless, EMF allows specifying both textual and graphical syntaxes for DSLs. In the case of PaaSage, the use case partners required a textual syntax for CAMEL.

Therefore, we adopted Xtext, which is a language development framework that is based on- and integrates with EMF. It facilitates the implementation of an Eclipse-based IDE providing features such as syntax highlighting, code completion, code formatting, static analysis, and serialisation.

Thanks to the combination of EMF, Eclipse OCL, and Xtext, we realised a CAMEL Textual Editor, which allows PaaSage users not only to specify CAMEL models but also to syntactically and semantically validate them.

[7] `http://wiki.eclipse.org/OCL`

3.4 Connected Data Objects (CDO)

As mentioned, CAMEL models are progressively refined throughout the various phases of the PaaSage workflow (*cf.* Section 2). Therefore, we adopted CDO to persist CAMEL models in the Metadata Database (MDDB) (*cf.* D4.1.2 [16]) and facilitate the integration across the components of the PaaSage platform. CDO is semi-automated persistence framework that works natively with Ecore models and their instances. It can be used as a model repository where clients persist and share their models. It provides features that satisfy the design-time and run-time requirements of the PaaSage platform, such as:

- Validation: CDO supports automatic checking of the conformance between the models persisted in the repository and their metamodel. This also ensures that the models persisted in the repository are valid at any time. Both EMF- and OCL-based validation is supported.
- Transaction: CDO supports transactional manipulations of the models persisted in the repository. This ensures that the models persisted in the repository are valid at any time, so that the components of the PaaSage workflow can rely on a consistent view of the data.
- Versioning: CDO supports *optimistic* versioning[30], where each client of the repository has a local (or working) copy of a model. These local copies are modified independently and in parallel and, as needed, local modifications can be committed to the repository. Non-overlapping changes are automatically merged. Otherwise, they are rejected, and the model is put in a *conflict* state that requires manual intervention.
- Automatic Notification: CDO automatically notifies clients about changes in the state of the models persisted in the repository. This allows PaaSage components to monitor certain models or parts of the models and respond to events that occur in the system.
- Auditing: CDO automatically records the history of revisions of each model since its creation, thus allowing to trace the model origin.
- Role-based Security: CDO provides role-based access control to the models persisted in the repository, thus supporting the controlled access to (parts of) models by different components and actors in the PaaSage workflow.

4 CAMEL Textual Editor

In this section, we provide the list of steps for installing and using the CAMEL Textual Editor. These steps have been tested with the latest version of Eclipse, which at the time of writing is Eclipse Neon v4.6.0. We distinguish between the installation for PaaSage users and for PaaSage developers.

4.1 Installation—Users

- Download and install "Eclipse IDE for Java and **DSL** Developers" from: `https://www.eclipse.org/downloads/`
- Start Eclipse
- Select Help > Install New Software...
- Select Work with: Neon—http://download.eclipse.org/releases/neon
- Select Modeling > CDO Model Repository SDK
- Select Modeling > OCL Classic SDK: Ecore/UML Parsers,Evaluator,Edit
- Select Modeling > OCL Examples and Editors
- Install the three packages
- Download `org.ow2.paasage.camel_2015.9.1.jar`, `org.ow2.paasage.camel.dsl_2015.9.1.jar`, and `org.ow2.paasage.camel.dsl.ui_2015.9.1.jar` from: `http://jenkins.paasage.cetic.be/job/CAMEL/`
- Copy the three jar files to the `eclipse/plugins` folder
- Restart Eclipse

4.2 Installation—Developers

- Clone the CAMEL Git repository from: `https://tuleap.ow2.org/plugins/git/paasage/camel`
- Download and install "Eclipse IDE for Java and **DSL** Developers" from: `https://www.eclipse.org/downloads/`
- Start Eclipse
- Select Help > Install New Software...

- Select Work with: Neon—http://download.eclipse.org/releases/neon
- Select Modeling > CDO Model Repository SDK
- Select Modeling > OCL Classic SDK: Ecore/UML Parsers,Evaluator,Edit
- Select Modeling > OCL Examples and Editors
- Install the three packages
- Restart Eclipse
- Select Import > Existing Projects into Workspace
- Select Browse...
- Select the folder where you cloned the CAMEL Git repository
- Select Finish
- Select `eu.paasage.camel.dsl/src/eu.paasage.camel.dsl/GenerateCamelDsl.mwe2`
- Select Run As > MWE2 Workflow...
- Select `eu.paasage.camel.dsl`
- Select Run > Run As > Eclipse Application...

4.3 Usage

- Add a (general) project
- Add a new file (or open an existing one) with `.camel` extension to the project
- Accept to add the Xtext nature to the project
- Restart Eclipse
- **Read the remainder of this paper**
- Edit the file

5 Naming Conventions

In this section, we provide naming conventions in order to ensure the correct specification of CAMEL models. All elements in a CAMEL model are identified by a name. The name must be a string with no quotes surrounding it. The string must be a concatenation of meaningful names in CamelCase syntax. For instance, if we have to specify an *average response time* metric, we could use the name `AverageResponseTime`.

A reference in a CAMEL model (in textual syntax) must be a string with no quotes surrounding it. The string must be a fully qualified name conforming to the following pattern: $id_1.id_2. \ldots .id_n$, where id_i, with $i \leq n$, refers to the name of an element at the i^{th} level of the containment path and id_n refers to the name at the leaf level, which is actually the name of the element at hand. For instance, if we have to refer to the `AverageResponseTime` metric, we must use the fully qualified name `MyModel.MyMetric.AverageResponseTime`.

6 CAMEL Metamodel

CAMEL is available under Mozilla Public License 2.0[8] in the Git repository at OW2[9]. As mentioned, the CAMEL metamodel is an Ecore model, and is organised into packages, whereby each package reflects an aspect (or domain).

Figure 4 shows the top-level camel package of the CAMEL metamodel. A CamelModel consists of an Application along with a collection of sub-models, namely DeploymentModels (*cf.* Section 7), RequirementModels (*cf.* Section 8), LocationModels (*cf.* Section 9), MetricModels (*cf.* Section 10), ScalabilityModels (*cf.* Section 11), ProviderModels (*cf.* Section 12), OrganisationModels (*cf.* Section 13), SecurityModels (*cf.* Section 14), ExecutionModels (*cf.* Section 15), TypeModels (*cf.* Section 16), and UnitModels (*cf.* Section 17).

An Application (see Figure 6) represents a cross-cloud application. It refers to an Owner, which represents the entity of an organisation owning the application. Moreover, it refers to one or more DeploymentModels, which represent the topology of the application along with the commands to handle its life cycle. The properties version and description represent the version (or revision) and description of the application, respectively.

As mentioned, in the remainder of the paper we adopt the Scalarm[10][17] use case as a running example to exemplify how to specify CAMEL models. The

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

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

[10] `http://www.scalarm.com/`

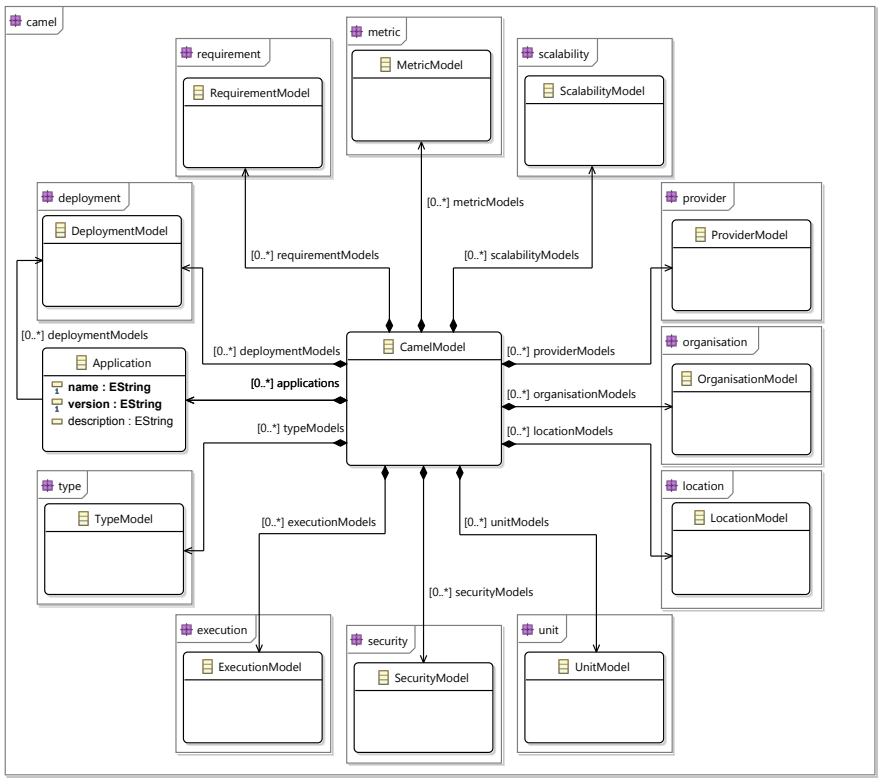

Figure 4: The class diagram of the CAMEL metamodel including packages

complete Scalarm CAMEL model in textual syntax can be downloaded from the Git repository at OW2[11].

Example

Assume that we have to specify the Scalarm application of the Scalarm use case. Listing 1 shows this specification in textual syntax.

Listing 1: Scalarm sample `application`

```
1 camel model ScalarmModel {
2
3 application ScalarmApplication {
4   version: 'v1.0'
5   owner: AGHOrganisation.morzech
6   deployment models [ScalarmModel.ScalarmDeployment]
7 }
8
9 organisation model AGHOrganisation {
```

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

```
10    ...
11    user morzech {
12      ...
13    }
14  }
15
16  deployment model ScalarmDeployment {
17    ...
18  }
19
20  }
```

`camel model ScalarmModel` specifies the CAMEL model for the Scalarm application. `application ScalarmApplication` specifies the Scalarm application itself. `version: 'v1.0'` specifies that Scalarm application has version 1.0. `owner: AGHOrganisation.morzech` specifies that Scalarm application is owned by the user `morzech` of the organisation AGH. `deployment models [ScalarmModel.ScalarmDeployment]` specifies that the Scalarm application has one `ScalarmDeployment` deployment model.

7 Deployment

The deployment package of the CAMEL metamodel is based on CloudML[12] [9, 7, 8], which was developed in collaboration with the MODAClouds project[13]. 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 upon MDE techniques and methods.

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

Cloud provider-independence (R_1): CloudML should support a cloud provider-agnostic specification of the provisioning and deployment. This will simplify the design of cross-cloud applications and prevent vendor lock-in.

Separation of concerns (R_2): CloudML should support a modular, loosely-coupled specification of the deployment. This will facilitate the maintenance as well as the dynamic adaptation of the deployment model.

Reusability (R_3): CloudML should support the specification of types that can be seamlessly reused to model the deployment. This will ease the evolution as well as the rapid development of different variants of the deployment model.

[12] `http://cloumdml.org`
[13] `http://www.modaclouds.eu`

Abstraction (R_4): CloudML should provide an up-to-date, abstract representation of the running system. This will facilitate the reasoning, simulation, and validation of the adaptation actions before their actual enactments.

CloudML is also inspired by component-based approaches [32], 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, CloudML 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* [18]. 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.

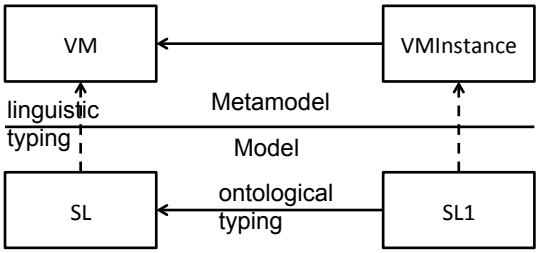

Figure 5: Linguistic and ontological typing

In the following, we describe and exemplify the main concepts in the deployment package.

Figure 6 shows the type part of the class diagram of the deployment package. A DeploymentModel (omitted for brevity) is a collection of DeploymentElements. A deployment element can be a Component, a Communication, or a Hosting. A deployment element can refer to Configurations, which represent sets of commands to handle the life cycle of the deployment element.

In the following, we discuss the three kinds of deployment elements.

7.1 Components, Communications, and Hostings

A Component (see Figure 6) represents a reusable type of application component. A component can be an InternalComponent or a VM (short for virtual machine).

A VMRequirementSet represents a set of requirements for virtual machines, *i.e.* hardware requirements, operating system requirements, location requirements, and image requirements (*cf.* Section 8). It can be referred to by a virtual

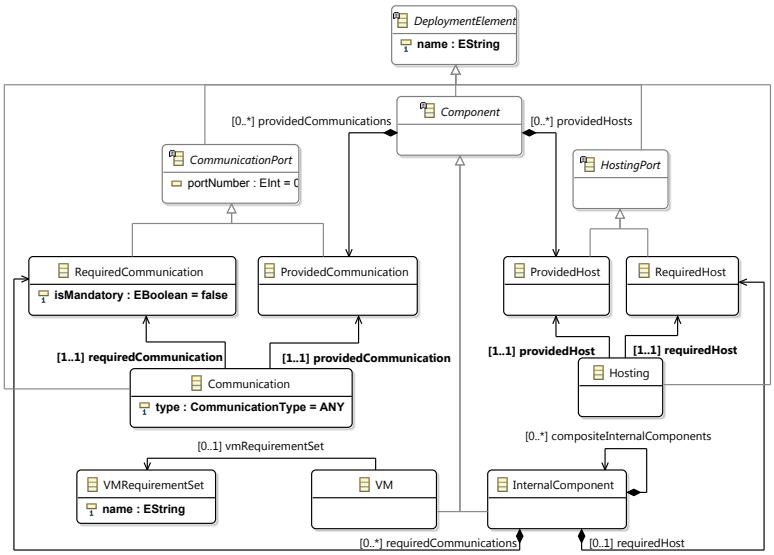

Figure 6: Part of the class diagram of the deployment package related to deployment types

machine or by a deployment model. In the former case, the set of requirements applies to all instances of the virtual machine, whereas in the latter case, the set of requirements applies to all instances of all virtual machines within a deployment model.

A CommunicationPort (see Figure 6) represents a communication port of an application component. A communication port can be a ProvidedCommunication, meaning that it provides a feature to another component, or a RequiredCommunication, meaning that it consumes a feature from another component. The property isMandatory of RequiredCommunication represents that the component depends on the feature provided by another component.

A HostingPort (see Figure 6) represents a containment port of a component. A hosting port can be a ProvidedHost, meaning that it provides an execution environment to another component (*e.g.*, a virtual machine provides an execution environment to a servlet container, and a servlet container provides an execution environment to a servlet), or a RequiredHost, meaning that a component consumes an execution environment from another component.

A Communication represents a reusable type of communication binding between a required and a provided communication port. The property type of Communication specifies that the components at each end of the communication can be deployed on any virtual machine instances (value ANY, default); the components must be deployed on the same virtual machine instance (value LOCAL); the components must be deployed on separate virtual machine instances (value REMOTE).

A Hosting represents a reusable type of containment binding between a required and a provided host port.

Example

Assume that we have to specify the Experiment Manager component of the Scalarm use case. Listing 2 shows this specification in textual syntax.

`internal component ExperimentManager` specifies a reusable type of the component Experiment Manager. `provided communication ExpManPort` represents that the Experiment Manager provides its features through port 443. `required communication StoManPortReq` and `InfSerPortReq` specify that the Experiment Manager requires features from the Information Service through port 11300 and from the Storage Manager through port 20001, respectively. The property `mandatory` of the latter specifies that the Execution Manager depends on the features of the Storage Manager (hence, the Storage Manager has to be deployed and started before the Execution Manager). `required host CoreIntensiveUbuntuGermanyReq` specifies that the Experiment Manager requires a virtual machine with a large number of CPU cores, running operating system Ubuntu, and located in Germany (*cf.* Listing 3).

`configuration ExperimentManagerConfiguration` specifies the commands to handle the life cycle of the Experiment Manager. `download`, `install`, and `start` specify the Unix shell 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 manage their life cycles. The interested reader may refer to [29] for a detailed description of how the Executionware invokes these commands.

Listing 2: Scalarm sample `internal component`

```
1  deployment model ScalarmDeployment {
2
3    internal component ExperimentManager {
4      provided communication ExpManPort {port: 443}
5      required communication StoManPortReq {port: 20001 mandatory}
6      required communication InfSerPortReq {port: 11300}
7      required host CoreIntensiveUbuntuGermanyHostReq
8
9      configuration ExperimentManagerConfiguration {
10       download: 'cd ~ && wget https://github.com/kliput/
       scalarm_service_scripts/archive/paasage.tar.gz && sudo apt-get
       update && sudo apt-get install -y groovy ant && tar -zxvf paasage.
       tar.gz && cd ~/scalarm_service_scripts-paasage'
11       install: 'cd ~/scalarm_service_scripts-paasage && ./
       experiment_manager_install.sh'
12       start: '~/scalarm_service_scripts-paasage/
       experiment_manager_start.sh'
```

```
13      }
14    }
15  ...
```

Then, assume that we have to specify the virtual machine on which the Experiment Manager can be deployed. Listing 3 shows this specification in textual syntax.

`requirement set CoreIntensiveUbuntuGermanyRS` specifies a reusable set of requirements for a virtual machine. `quantitative hardware`, `os`, and `location` refer to the requirements `CoreIntensive`, `Ubuntu`, and `GermanyReq`, respectively, from the `requirement model ScalarmRequirement` (*cf.* Listing 7), 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 Germany, respectively (*cf.* Section 8).

`vm CoreIntensiveUbuntuGermany` specifies a reusable type for a virtual machine. `requirement set` refers to the aforementioned requirement set `CoreIntensiveUbuntuGermanyRS`. `provided host CoreIntensiveUbuntuGermanyPort` represents that the virtual machine provides a large number of CPU cores, running Ubuntu, and located in Germany.

Listing 3: Scalarm sample `vm`

```
1  ...
2    requirement set CoreIntensiveUbuntuGermanyRS {
3      quantitative hardware: ScalarmRequirement.CoreIntensive
4      os: ScalarmRequirement.Ubuntu
5      location: ScalarmRequirement.GermanyReq
6    }
7
8    vm CoreIntensiveUbuntuGermany {
9      requirement set CoreIntensiveUbuntuGermanyRS
10     provided host CoreIntensiveUbuntuGermanyPort
11   }
12 ...
```

Next, Assume that we have to specify the communication binding between the Experiment Manager and the Storage Manager. Listing 4 shows this specification in textual syntax.

`communication ExperimentManagerToStorageManager` specifies a reusable type of communication binding between the Experiment Manager and the Storage Manager. `from .. to ..` block specifies that the communication binding is `from` the required communication port `StoManPortReq` of the component `ExperimentManager to` the provided communication port `StoManPort` of the component `StorageManager`. `type: REMOTE` specifies that the Experiment Manager and the Storage Manager must be deployed on separate virtual machine instances.

Listing 4: Scalarm sample `communication`

```
...
  communication ExperimentManagerToStorageManager {
    from ExperimentManager.StoManPortReq to StorageManager.StoManPort
    type: REMOTE
  }
...
```

Finally, assume that we have to specify the hosting binding between the Experiment Manager and the virtual machine `CoreIntensiveUbuntuGermany`. Listing 5 shows this specification in textual syntax.

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

Listing 5: Scalarm sample `hosting`

```
...
  hosting ExperimentManagerToCoreIntensiveUbuntuGermany {
    from ExperimentManager.CoreIntensiveUbuntuGermanyPortReq to
    CoreIntensiveUbuntuGermany.CoreIntensiveUbuntuGermanyPort
  }
...
```

7.2 Component, Communication, and Hosting instances

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 (*cf.* Section 2). In the general case, the instances could also be manipulated manually.

Example

Listing 6 shows the specification of instances of the components, virtual machines, communications, and hostings from the previous examples (*cf.* Listings 2, 3, 4, and 5) in textual syntax for illustrative purposes.

`vm instance CoreIntensiveUbuntuGermanyInst` 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 GWDGProvider` (*cf.* Listing 11), which is compatible with the requirement set of the virtual machine template `CoreIntensiveUbuntuGermany` (*cf.* Listing 8).

`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 `ExperimentManagerToStorageManager` and the hosting `ExperimentManagerToCoreIntensiveUbuntuGermany`, respectively. `typed` refers to the identifier of the corresponding type. `named` is optional and specifies the identifier of the instance.

Listing 6: Scalarm sample instances of `internal component`, `vm`, `communication`, and `hosting`

```
1  ...
2    vm instance CoreIntensiveUbuntuGermanyInst typed ScalarmModel.
       ScalarmDeployment.CoreIntensiveUbuntuGermany {
3      vm type: ScalarmModel.GWDGProvider.GWDG.VM.VMType
4      vm type value: ScalarmModel.GWDGType.VMTypeEnum.M1.LARGE
5      provided host instance CoreIntensiveUbuntuGermanyHostInst typed
       CoreIntensiveUbuntuGermany.CoreIntensiveUbuntuGermanyHost
6    }
7
8    internal component instance StorageManagerInst typed ScalarmModel.
       ScalarmDeployment.StorageManager {
9      provided communication instance StoManPortInst typed
       StorageManager.StoManPort
10     required communication instance InfSerPortReqInst typed
       StorageManager.InfSerPortReq
11     required host instance StorageIntensiveUbuntuGermanyHostReqInst
       typed StorageManager.StorageIntensiveUbuntuGermanyHostReq
12   }
13
14   internal component instance ExperimentManagerInst typed ScalarmModel
       .ScalarmDeployment.ExperimentManager {
15     provided communication instance ExpManPortInst typed
       ExperimentManager.ExpManPort
16     required communication instance StoManPortReqInst typed
       ExperimentManager.StoManPortReq
17     required communication instance InfSerPortReqInst typed
       ExperimentManager.InfSerPortReq
18     required host instance CoreIntensiveUbuntuGermanyHostReqInst typed
        ExperimentManager.CoreIntensiveUbuntuGermanyHostReq
19   }
20
21   connect ExperimentManagerInst.StoManPortReqInst to
       StorageManagerInst.StoManPortInst typed ScalarmModel.
       ScalarmDeployment.ExperimentManagerToStorageManager named
       ExperimentManagerToStorageManagerInst
22
23   host ExperimentManagerInst.CoreIntensiveUbuntuGermanyHostReqInst on
       CoreIntensiveUbuntuGermanyInst.CoreIntensiveUbuntuGermanyHostInst
       typed ScalarmModel.ScalarmDeployment.
       ExperimentManagerToCoreIntensiveUbuntuGermany named
       ExperimentManagerToCoreIntensiveUbuntuGermanyInst
24 ...
```

7.3 Interplay with Executionware

It is important to understand that, in order to execute an application, *i.e.*, deploying and executing at least one instance of each of its components, the Execu-

tionware can solely rely on the information provided in the deployment model within a CAMEL model. Hence, the Executionware does not make any assumptions besides the information provided. In particular, this means: only ports of communications specified in the CAMEL model are guaranteed to be opened; other ports may be blocked by either the cloud infrastructure or operating system.

In order to steer the individual instances of internal components, the Executionware relies on handlers that have been specified in the `configuration` of an internal component. The handlers are invoked in the following order:

1. download
2. install
3. configure
4. start

Life Cycle Scripts for Unix-based Applications

As stated above, the Executionware relies on the deployment model within a CAMEL model, and in particular on the `configuration` block of `internal components`, in order to manage the component instances. For GNU/Linux deployments, all of the handlers are executed as a single Unix shell script (*e.g.*, compatible with Bash) that has to be specified in the `configuration` block of `internal components` (*e.g.*, for downloading the executable code of the component). A return value different from `0` is interpreted as an error and causes the component instance to move to an `error` state. Data about ports and connection information as well as the local host is provided via environment variables.

Note that the different handlers are not necessarily executed in the very same instance of the Unix shell. This means that custom environment variables set in a handler (*e.g.*, in the `download` command) are not necessarily available in later handlers. If such information is required, the only approach is to write the necessary data to a file and `source` this file in later handlers. For GNU/Linux deployments, all component instances are run within an own Docker container[14] in order to enable a maximum of isolation between the instances. This has a consequence on user handling and networking: As for the users, this means that all commands are executed as `root`. Also, the handlers cannot assume that any other user beside `root` exists in the system. Hence, if further users are required, the handlers are responsible for creating them. As for networking, the effects on both IP addresses and port numbers are discussed in the following.

[14] `http://docker.io`

IP Addresses in the Execution Environment: First, all components have at least two IP addresses, namely the IP address of their Docker container and the IP address of the virtual machine this container is hosted on. Often, the IP of the virtual machine is a cloud-internal IP address that is not routed outside the cloud provider. Hence, it is very likely that there is a third IP address involved that represents the public IP address of the virtual machine. All the three IP addresses are passed to `configuration` and `start` handlers as environment variables using the following formats:

- `CONTAINER_IP`: the IP address of the container. It should be used for binding purposes.
- `CLOUD_IP`: the IP address of the virtual machine running the container. This IP is probably cloud provider-specific and cannot be reached from outside the cloud.
- `PUBLIC_IP`: the public IP address of the virtual machine running the container, if available.

Port Numbers in the Execution Environment: Moreover, the port numbers used within the container do not necessarily match the port numbers as used by the operating system hosting the Docker container. Indeed, the Executionware will not force the use of any fixed port numbers outside the container in order to allow maximum flexibility. Again, the port numbers are passed to the `configuration` and `start` handlers as environment variables. The name of the variable is based on the name of the `provided communication` from the deployment model. For instance, the provided port `ExpManPort` from Listing 2 is mapped to the following three environment variables:

- `CONTAINER_EXPMANPORT`: the port number as specified in the deployment model and as accessible from within the container. Should be used for binding.
- `CLOUD_EXPMANPORT`: the port number as accessible from within the cloud.
- `PUBLIC_EXPMANPORT`: the port number as accessible from the outside world (*i.e.*, by using the public IP).

Outgoing Connections in the Execution Environment: Similar to provided communications, there is a mapping for required communications. The main difference is that it uses sets of IP addresses in combinations with ports. For instance, the required port `StoManPortReq` from Listing 2 is mapped to the following three environment variables; all consisting of a sequence of `ipv4:port` separated by comma (,).

- PUBLIC_STOMANPORTREQ: provides access to the public IP addresses and public ports of all downstream component instances.
  <stoman1publicip>:<public_port>,<stoman2publicip>:<public_port>

- CLOUD_STOMANPORTREQ: provides access to the cloud-internal IP addresses and cloud-internal ports of all downstream component instances. Note that addresses of component instances not hosted in the same cloud as the local component instance are still in the list, but very likely traffic cannot be routed to them.
  <stoman1cloudip>:<cloud_port>,<stoman2cloudip>:<cloud_port>

- CONTAINER_STOMANPORTREQ: provides access to the container-internal IP addresses and container-internal ports of all downstream component instances. Note that addresses of component instances not hosted in the very same container as the local component instance are still in the list, but very likely cannot be routed to.
  <stoman1containerip>:<container_port>,<stoman2containerip>:
  <container_port>

Currently, it is up to the CAMEL user to decide which of the combinations is needed. Using the public IPs and ports enables a full routability of network traffic, but may introduce networking overhead. Future work may improve upon this status quo by providing shortest distance combinations of the addresses.

Updating Required Communications: Whenever the set of downstream instances changes (*e.g.*, a new Storage Manager instance is created), the `start` handler of the associated required communication is invoked. This should lead to an updated configuration and if necessary a re-started main process.

The Start Life Cycle Scripts: The life cycle script attached to `start` is a special script. It is supposed to not return from its call. As such, the Execution Environment will use `exec <install command from CAMEL>`. This means that the CAMEL user **shall not** use more than one command in the install handler, as *e.g.*, `cd directory && ./run.sh` will not work. The same holds for `cd directory ; ./run.sh`. Use `directory/run.sh` instead.

Other Environment Variables: Per default, Docker uses only very few environment variables in a default container and except for HOME (home of the current user—`root` by default), PWD (current working directory—`/` by default), and PATH. Users should not rely on any of these.

Life Cycle Scripts for Windows-based Applications

For Windows-based deployments the Executionware relies on the same deployment model as GNU/Linux deployments, mainly the `configuration` block of

`internal components`. All handlers are executed as a single Powershell script. The return value will be interpreted and a value different from `0` will move the instance to an error state. Information about ports, connection and the local host is provided via environment variables.

Like on GNU/Linux, the different handlers are not necessarily executed in the very same shell instance, in particular the same Powershell instance. If custom environment variables are set in a handler which will be used in a later handler they have to be set on the user-level. In Powershell this can be achieved with the command `[Environment]::SetEnvironmentVariable($NAME, $VALUE, "User")`. `$NAME` and `$VALUE` represent the respective parameters and the static value `"User"` specifies the user-level for the environment variable. For Windows deployments every components runs in its own folder. All commands are executed as `Administrator` and no other existing users can be assumed. If further users are required, the handlers are responsible for creating them. The networking is discussed in the following.

IP Addresses in the Execution Environment: Unlike Unix components, Windows components have at least one IP address. Depending on the cloud provider, it is possible that this IP is a cloud-internal IP and there is a second IP address that represents the public IP address of the virtual machine. Both IP addresses are passed to `configuration` and `start` handlers as environment variables:

- `CLOUD_IP`: the IP address of the virtual machine running the component. This IP is probably cloud provider-specific and cannot be reached from outside the cloud.
- `PUBLIC_IP`: the public IP address of the virtual machine running the component, if available.

Port Numbers in the Execution Environment: As Windows components just run in an unique folder and not in a container like Unix components there is a small difference. The port numbers of Windows components match the port numbers of the virtual machine's operating system. The port numbers are passed to the `configuration` and `start` handlers as environment variables. The name of the variable is based on the name of the provided communication from the deployment model. Considering the example from Listing 2 as a Windows component the resulting two environment variables are set:

- `CLOUD_EXPMANPORT`: the port number as accessible from within the cloud.
- `PUBLIC_EXPMANPORT`: the port number as accessible from the outside world (*i.e.*, by using the public IP).

Outgoing Connections in the Execution Environment: The mapping of required communications is similar to Unix components (*cf.* Section 7.3) except there is no need to map the communication to a container-internal IP. Again considered Listing 2 as a Windows component the required port is mapped to the following environment variables:

- `PUBLIC_STOMANPORTREQ`: provides access to the public IP addresses and public ports of all downstream component instances.
  `<stoman1publicip>:<public_port>,<stoman2publicip>:<public_port>`

- `CLOUD_STOMANPORTREQ`: provides access to the cloud-internal IP addresses and cloud-internal ports of all downstream component instances. Note that addresses of component instances not hosted in the same cloud as the local component instance are still in the list, but very likely cannot be routed to.
  `<stoman1cloudip>:<cloud_port>,<stoman2cloudip>:<cloud_port>`

Updating Required Communications: Whenever the set of downstream instances
changes (*e.g.*, a new Storage Manager instance is created), the `start` handler of the associated required communication is invoked. This should lead to an updated configuration and if necessary a re-started main process.

The Start Life Cycle Scripts: In contrast to Unix components, for Windows components the `start` command is supposed to return from its call. It is also possible to use more than one command like in all other handlers.

Other Environment Variables: The default Windows environment variables are set (*e.g.*, `HOMEPATH` or `ProgramData`), but the user should be aware that the environment variables can differ depending on the operating system version.

8 Requirements

The requirement package provides the concepts to specify requirements for cross-cloud applications. In the following, we describe and exemplify the main concepts in the requirement package.

8.1 Requirements and RequirementGroups

Figure 7 shows the part of the class diagram of the requirement package related to its main concepts.

A RequirementModel is a collection of Requirements. A requirement can be a HardRequirement, such as a service level objective (SLO) (*e.g.*, response time <

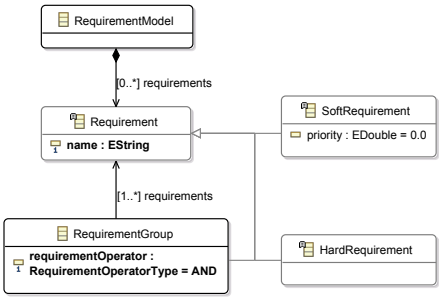

Figure 7: Part of the class diagram of the requirement package related to its main concepts

100ms), meaning that it is measurable and must be satisfied, or a SoftRequirement, such as a optimisation objective (*e.g.*, maximise performance), meaning that it is not measurable. The property priority of SoftRequirement represents the priority of the soft requirements. These priorities can be used to rank these soft requirements when reasoning on the application and generating a new CPSM for it. They must be specified in the scale from 0.0 to 1.0 and the respective soft requirements could refer to metrics that are generated based on normalisation functions also taking values from 0.0 to 1.0.

A RequirementGroup represents a logical group of requirements, which can be comprised of single requirements or other requirement groups. The property requirementOperator of RequirementGroup represents the logical operator that is used to connect these requirements and can be assigned values AND (logical conjunction) or OR (logical disjunction). A requirement group refers to an Application for which the requirements must be satisfied. Note that a requirement group should not contain conflicting requirements, such as scale requirements that are of the same type and that refer to the same component.

A requirement group allows creating a requirement tree, which represents a tree of logically connected requirements that must be satisfied. For instance, a top-level requirement group (*e.g.*, identified by the name Global) could contain two or more requirement groups logically connected by the OR operator. Each of the latter requirement groups (*e.g.*, identified by the name Alternative_i) could in turn contain single requirements, such as SLOs, logically connected by the AND operator.

In the following, we discuss the different kinds of requirements.

8.2 Hardware, OS & Image and Provider Requirements

Figure 8 shows the part of the class diagram of the requirement package related to hardware, OS, image, and provider requirements.

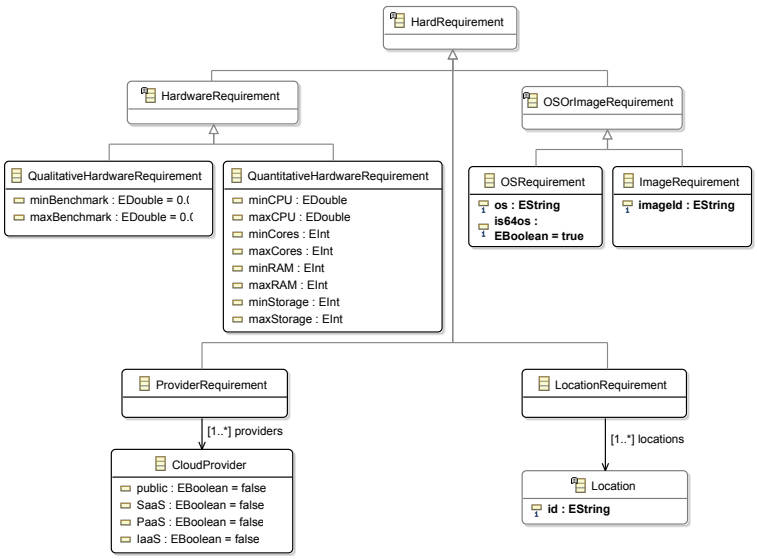

Figure 8: Part of the class diagram of the requirement package related to hardware, OS, image, and provider requirements

HardwareRequirement, OsOrImageRequirement, and ProviderRequirement are hard requirements. They can be referred to by a VMRequirementSet (*cf.* Section 7).

A HardwareRequirement can be specialised into a QualitativeHardwareRequirement, which represents a requirement on the performance of a virtual machine instance, or a QuantitativeHardwareRequirement, which represents a requirement on the virtual hardware of a virtual machine. The properties min- and maxBenchmark of QualitativeHardwareRequirement represent the range of benchmark results that a virtual machine instance must satisfy. The properties of QuantitativeHardwareRequirement represent the ranges of: CPU frequency, number of CPU cores, size of RAM, and size of storage that a virtual machine instance must satisfy. Note that for at least one of these properties, at least one of the minimum and maximum bounds must be specified.

A OsOrImageRequirement can be specialised into a OSRequirement, which represents a requirement on the operating system run by a virtual machine, or a ImageRequirement, which represents a requirement on the image deployed on a virtual machine. The property os of OSRequirement represents the required operating system (*e.g.*, "Ubuntu" or "Windows"), while the property is64os represents whether the operating system must be compiled for 64 bits architectures (*e.g.*, x86-64). The property imageId of ImageRequirement represents the identifier of the required image.

A ProviderRequirement represents the set of cloud providers that must be considered for an application deployment (*e.g.*, Amazon and Rackspace only).

A LocationRequirement refers to one or more Locations (*cf.* Section 9), which represent either geographical regions (*e.g.*, a continent, a country, or a region) or cloud locations (*i.e.*, a location specific to a cloud provider) that must be considered for an application deployment (*e.g.*, Germany only).

8.3 Service Level Objectives and Optimisation Requirements

Figure 9 shows the part of the class diagram of the requirement package related to SLOs and optimisation requirements.

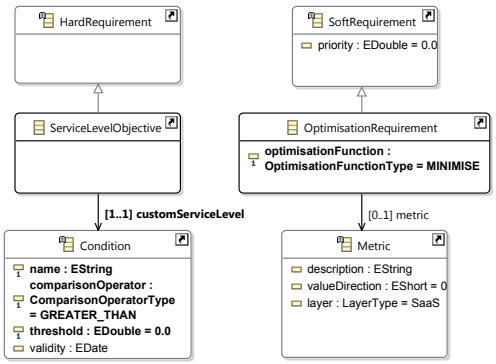

Figure 9: Part of the class diagram of the requirement package related to SLOs and optimisation requirements

A ServiceLevelObjective is a hard requirement. It refers to a Condition, such as MetricCondition (*cf.* Section 10), which represents the metric condition that must be satisfied (*i.e.*, the corresponding measurement values must not cross a particular threshold).

An OptimisationRequirement is a soft requirement. It refers to a Metric or a Property (*cf.* Section 10), which represents the metric or the property, respectively, that should be optimised. Moreover, it refers to an Application or InternalComponent, which represents the application or component, respectively, for which the metric should be optimised. The property optimisationFunction of OptimisationRequirement represents the optimisation function applied to the metric and can be assigned values MINIMISE or MAXIMISE.

8.4 Scale Requirements

Figure 10 shows the part of the class diagram of the requirement package related to scaling requirements.

A ScaleRequirement is a hard requirement. It can be referred to by a ScalabilityRule (*cf.* Section 11), which restrains which scaling actions are performed. A

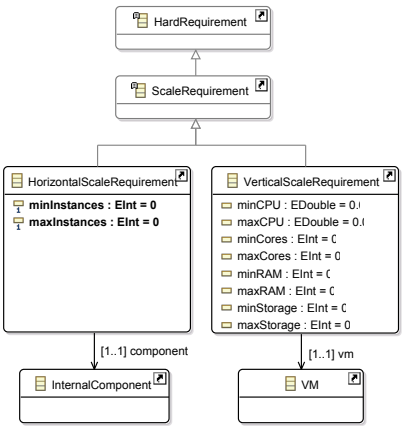

Figure 10: Part of the class diagram of the requirement package related to scaling requirements

ScaleRequirement can be a HorizontalScaleRequirement, which represents the minimum and maximum amount of instances allowed for a component, so that scale-out and scale-in actions will not exceed these bounds. Alternatively, it can be a VerticalScaleRequirement, which represents the minimum and maximum values allowed for virtual machine properties (*e.g.*, number of CPU cores), so that scale-up and scale-down actions will not exceed these bounds. Note that for horizontal scale requirements, the maximum number of instances should be either -1 (infinite) or greater or equal to the respective minimum amount. Minimum and maximum values must be specified for at least one virtual machine property.

8.5 Security Requirements

Figure 11 shows the part of the class diagram of the requirement package related to location and security requirements.

A SecurityRequirement is a hard requirement. It refers to one or more SecurityControls (*cf.* Section 14), which represent the security controls that must be enforced. Moreover, it can refer to an Application or InternalComponent, which represent the application or component on which the security controls must be enforced. If the security requirement refers to an application, then all cloud providers' offerings and services, which are used by the application, must support the corresponding security controls. In case the security requirement refers to a single component, such as a virtual machine, then only the cloud providers that support the corresponding security controls are considered for the particular component. If the security requirement does not refer to an application or a

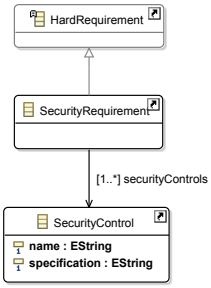

Figure 11: Part of the class diagram of the requirement package related to security requirements

component, then the security controls must be enforced on all applications and components within a deployment model.

Example

Assume that we have to specify the requirements for the components of the Scalarm use case. Listing 7 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 GermanyReq` specifies that a virtual machine must be deployed in Germany. `locations` refers to the location `DE` in the location model `ScalarmLocation` (*cf.* Listing 8). These three requirements above are referred to by the `requirement set CoreIntensiveUbuntuGermanyRS` in the `deployment model ScalarmDeployment` (*cf.* Listing 3).

`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 3).

`group ScalarmRequirementGroup` specifies that the requirements `CPUMetricSLO`, `MinimisePerformanceDegradationOfExperimentManager`, and `MinimiseDataFarmingExperimentMakespan` are logically conjuncted.

Finally, `horizontal scale requirement HorizontalScaleSimulationManager` specifies that the component `SimulationManager` must scale horizontally between

1 and 5 instances. `component` refers to the `internal component SimulationManager` in the `deployment model ScalarmDeployment` (*cf.* Listing 3).

Listing 7: Scalarm requirement model

```
requirement model ScalarmRequirement {

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

  os Ubuntu {os: 'Ubuntu' 64os}

  location requirement GermanyReq {
    locations [ScalarmLocation.DE]
  }

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

  optimisation requirement
    MinimisePerformanceDegradationOfExperimentManager {
    function: MIN
    metric: ScalarmModel.ScalarmMetric.
    MeanValueOfResponseTimeOfAllExprimentManagersMetric
    component: ScalarmModel.ScalarmDeployment.ExperimentManager
    priority: 0.8
  }

  optimisation requirement MinimiseDataFarmingExperimentMakespan {
    function: MIN
    metric: ScalarmModel.ScalarmMetric.MakespanMetric
    component: ScalarmModel.ScalarmDeployment.ExperimentManager
    priority: 0.2
  }

  group ScalarmRequirementGroup {
    operator: AND
    requirements [ScalarmRequirement.CPUMetricSLO, ScalarmRequirement.
    MinimisePerformanceDegradationOfExperimentManager,
    ScalarmRequirement.MinimiseDataFarmingExperimentMakespan]
  }

  horizontal scale requirement HorizontalScaleSimulationManager {
    component: ScalarmModel.ScalarmDeployment.SimulationManager
    instances: 1 .. 5
  }
}
```

9 Locations

The location package provides the concepts to specify locations. In the following, we describe and exemplify the main concepts in the location package.

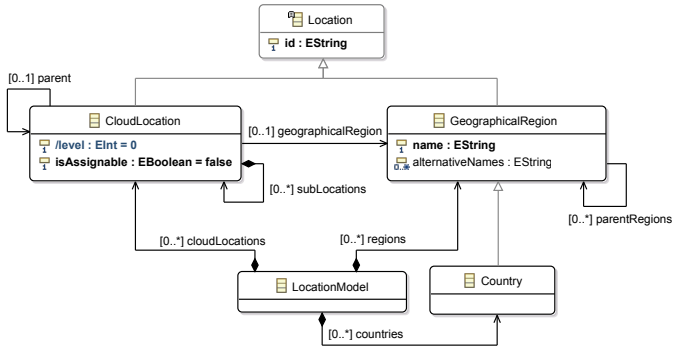

Figure 12: The class diagram of the location package

Figure 12 shows the class diagram of the location package.

A Location represents a physical or virtual location. It can be specialised into a GeographicalRegion, which represents a geographical region, or a CloudLocation, which represents virtual cloud location in a cloud (*e.g.*, Amazon EC2 eu-west-1). The property name of GeographicalRegion represents the name in English, while the property alternativeNames represents possible alternative names in other natural languages. A geographical region can refer to a parent region, which allows creating hierarchies of geographical regions (*e.g.*, continent, sub-continent, and country). A GeographicalRegion can be a Country, which represents a distinct entity in political geography. Similar to the geographical region, a cloud location can refer to a parent location, which allows creating hierarchies of cloud locations. Note that both the geographical region and the cloud location should not (recursively) refer to themselves.

Example

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

`region EU` specifies the region Europe. `country DE` specifies the country Germany. `parent regions` refers to the parent region Europe.

Listing 8: Scalarm location model

```
location model ScalarmLocation {

  region EU {
    name: 'Europe'
  }

  country DE {
    name: 'Germany'
    parent regions [ScalarmLocation.EU]
  }
```

```
11
12    country UK {
13      name: 'United Kingdom'
14      parent regions [ScalarmLocation.EU]
15    }
16  }
```

10 Metrics

The metric package of the CAMEL metamodel is based on the Scalability Rule Language (SRL) [15, 5]. 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. In the following, we describe and exemplify the main concepts in the metric package, namely metrics, properties, windows, and schedules in Section 10.1 and conditions and contexts in Section 10.2. The scalability aspects are then presented in Section 11.

10.1 Metrics, Properties, Windows, and Schedule

In order to identify event patterns in a scalability rule the components and virtual machines must be monitored.

Figure 13 shows the part of the class diagram of the metric package related to its main concepts. Figure 14 shows the part of the class diagram of the metric package related to enumeration types.

Generally, a *metric* is a standard of measurement. In the metric package, a Metric represents a generic metric and encapsulates the details for measuring properties (*e.g.*, average CPU load metric). A RawMetric represents a metric collected through direct measurements (*e.g.*, CPU load). A CompositeMetric, in turn, represents a metric computed from other metrics. A metric refers to the Unit of measurement (*e.g.*, the PERCENTAGE unit for a CPU load metric). In order to assist in checking the correctness of measurement values or their aggregations, a metric also refers to a ValueType, which represents the range of values the metric is allowed to take.

Each CompositeMetric refers to a MetricFormula, which defines the computation used to derive this metric. For that purpose a MetricFormula refers to one or more MetricFormulaParameters, which define the input for the formula. Further, it refers to a pre-defined function specifying the operation of the formula. There exist three types of parameters: constants, Metrics, or MetricFormulas. In case of constants, the parameter refers to a Value. Thus, metric formulas can involve not only constant values and other metrics but also calls to other metric formulas.

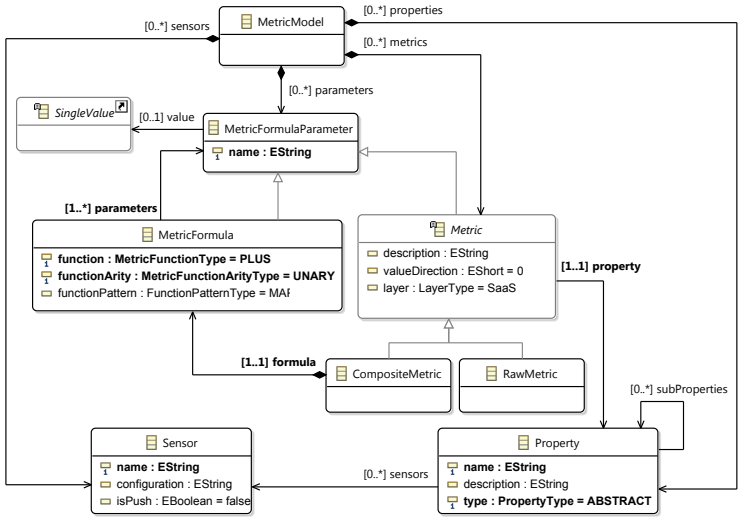

Figure 13: Part of the class diagram of the metric package related to its main concepts

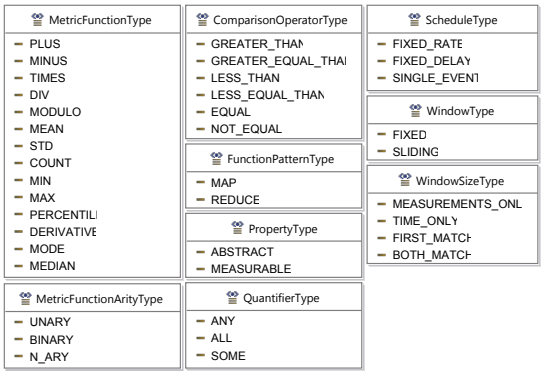

Figure 14: Part of the class diagram of the metric package related to enumeration types

The MetricFunctionType represents the pre-defined function types, which include mean (MEAN), standard deviation (STD), addition (ADD), subtraction (MINUS), division (DIV), and minimum (MIN). The function type restricts the number of parameters, their order, and their kind. For instance, MEAN refers to one parameter only, which should be of type metric or metric formula.

The FunctionPatternType classifies the strategy used to combine sets of metric instances.

Any Metric also refers to a measurable Property, which represents the measured non-functional property of a component or virtual machine. The property type represents the kind of property and can be assigned values MEASURABLE

(the property can be measured directly) and ABSTRACT (the property cannot be measured and is reified by its sub-properties).

For instance, in the security domain, the *incident management quality* is a property that is realised at least by the concrete and measurable *reporting capability* sub-property.

Figure 15 shows the part of the class diagram of the metric package related to metric instances.

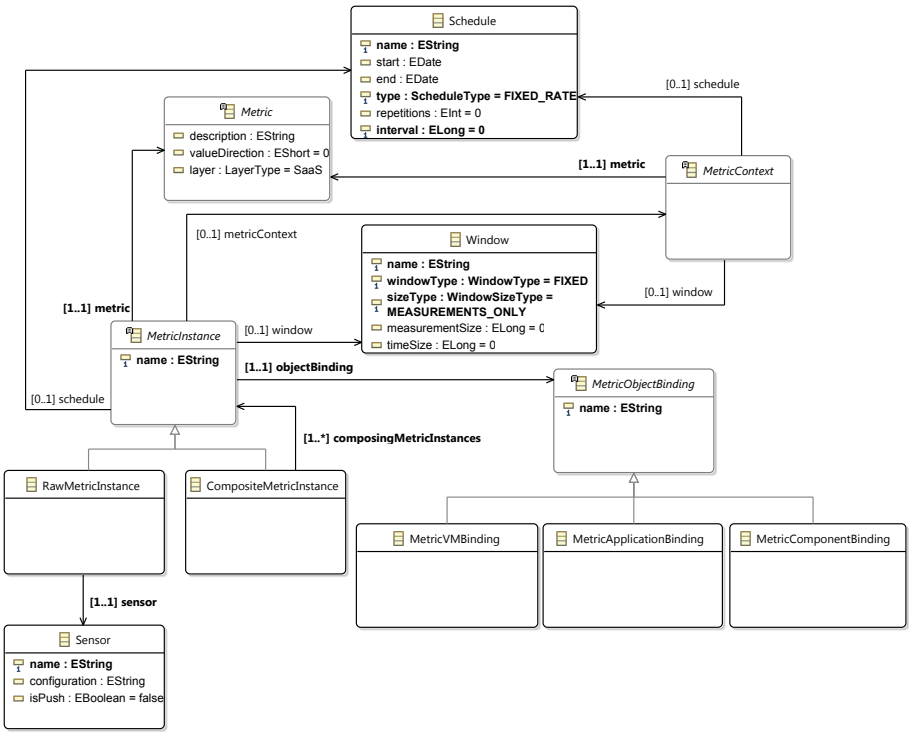

Figure 15: Part of the class diagram of the metric package related to metric instances

Following the type-instance pattern, a MetricInstance represents a concrete metric which has been created in order to measure a property for a particular instance of a virtual machine or component. Metric instances follow the same specialisation into raw and composite metric instances as metrics. The differentiator between metric instances and metrics is that a MetricInstance has measured values attached. For raw metric instances, this data corresponds to time series generated by a sensor. The property configuration defines that sensor (*e.g.*, the name of the probe to be installed on a monitoring system). The property isPush defines whether the measured data will be pushed by the sensor or have to be pulled by the run-time system.

The data associated to a composite metric instance CMI is computed from the data of other metric instances according to the computation specified by the corresponding metric CM. It is important to note that from the descriptions given so far, there exist multiple ways of creating composite metric instances from a metric. In order to narrow down and the creation, the FunctionPatternType is used as demonstrated in the following example:

Assume we have a composite metric CM computed from a source metric M_c using MEAN as the function type of the corresponding metric formula. This specification can be interpreted in two ways: (i) for each metric instance associated with M_c, compute the average and map it to a composite metric instance; (ii) compute the average over all metric instances associated with M_c and compute the overall average. Using MAP as a function pattern type for CM realises case (i), *mapping* each metric instance of M_c to a new metric instance; using REDUCE realises case (ii), *reducing* a set of metric instances to a single instance.

Each MetricInstance refers to a particular sub-class of MetricObjectBinding: MetricComponentBinding, MetricVMBinding, MetricApplicationBinding. Bindings are used by the run-time system to configure the monitoring system. A MetricComponentBinding associates a particular component and respective application with a metric and requires the deployment of one or more sensors reporting measurements for this component. Similarly, the MetricVMBinding associates a virtual machine with a metric. Finally, the MetricApplicationBinding associates the application as a whole with a metric. In this case, one or more sensors will have to be deployed to virtual machines on which one or more component instances of this application have been deployed to perform the respective measurements to be aggregated.

Windows and Schedules allow to further specify the way computations of composite metrics are performed.

Metric instances may refer to Windows, which represent how multiple measurements will be temporary stored and used to perform computations for this instance. The window size may be defined by a time frame, a fixed number of measurements, or both. In the last case, it may be sufficient to wait for either the first property to be fulfilled, or for both. The property sizeType represents the strategy to be used for this purpose. Finally, the property windowType, in turn, represents what happens when the window size is reached. SLIDING represents that the window is slid by dropping superfluous elements, while FIXED represents that the window is cleared.

Metric instances may also refer to a Schedule, which represents any aspect of the operations/measurements that must be executed on a regular, timely basis (*e.g.*, when an operation must run and when the scheduling must end). For a composite metric instance, a schedule defines when and how often the instance will be evaluated by applying the indirectly associated MetricFormula. For a raw metric instance, a schedule defines how often its value is measured by the re-

spective metric sensor. The property type represents whether successive runs happen at a fixed rate or with a fixed delay. Finally, the property intervalUnit represents the time unit used for the schedule interval.

10.2 Conditions and Contexts

Figure 16 shows the part of the class diagram of the metric package related to conditions.

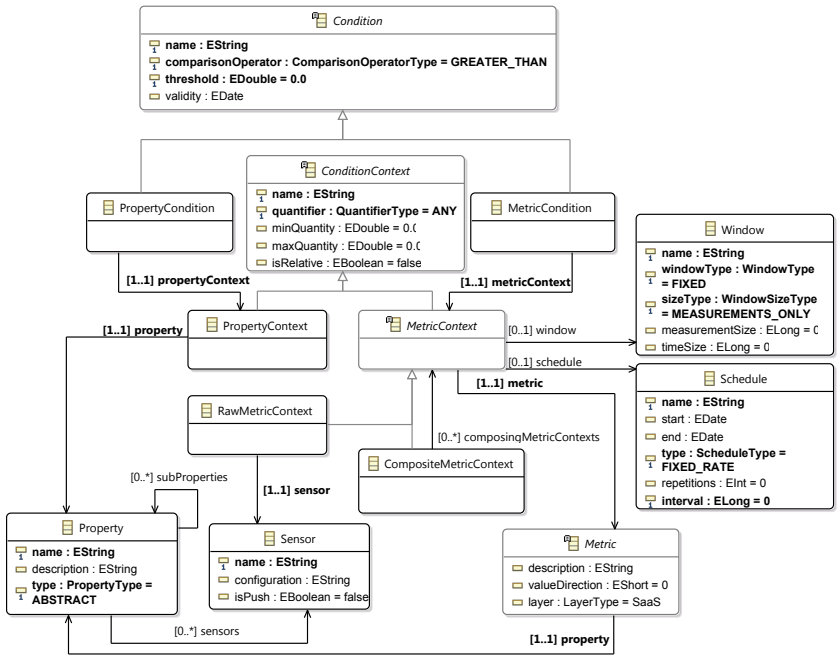

Figure 16: Part of the class diagram of the metric package related to conditions

A Condition represents an abstract condition with a threshold value. The property operator represents a comparison operator *i.e.*, greater than or less than (including and excluding equality) as well as (in)equality. The property validity defines for how long the condition will be valid. A condition enables expressing general requirements for cloud-based applications not tied to a particular deployment model. It also enables the expression of generic requirements that may hold for a CAMEL user irrespective of the applications. A condition can be specialised into a MetricCondition or a PropertyCondition.

A MetricCondition represents a constraint imposed on a metric. A constraint is violated when the comparison of the values of the instances of this metric with the threshold of the condition using the metric's operator is false. The violation of a metric condition will lead to the triggering of a simple, non-functional

event (*cf.* Section 11.3) and/or a violation of an SLO (which is reflected in the respective SLOAssessment—*cf.* Section 15). Note that constraints are not expressed on metric instances but on metrics. This enables the re-use of metric conditions in different execution contexts as well as guides the production of the instances of the metrics involved in these conditions to enable the assessment of the conditions under the respective particular execution contexts.

A PropertyCondition represents a condition on a property. Thus, it is possible to specify constraints, *e.g.*, on cost for the whole application or for some of its components only. These constraints must be interpreted appropriately in order to derive the required property values (*e.g.*, based on a particular internal metric used for producing the respective property value). As a property is not associated to a specific unit, a property condition refers to a monetary unit (*e.g.*, euros) and time interval unit (*e.g.*, seconds) to allow assessing the cost per time and not in absolute terms.

A condition, either pertaining to a metric or to a property, refers to a particular ConditionContext, which represents the context under which the condition should hold. The context represents whether the condition must be enforced on the whole application or a particular component/virtual machine. It also represents for how many instances of the application or component/virtual machine the condition must be checked. Two different types of quantification are distinguished: relative and absolute. The qualifier relative represents the minimum and maximum percentage of application or component/virtual machine instances on which the condition must hold. The qualifier absolute, in turn, represents the minimum and maximum number of instances of an application or component/virtual machine on which the condition must hold.

Four of the properties of ConditionContext allow specifying quantifications: (a) quantifier refers to a QuantifierType, which represents the set of instances to consider (all, any, some) in order to evaluate the condition—in case of *some*, the constructs (b)–(c) can be used to further specify the type and limits of quantification; (b) isRelative defines whether a relative or absolute quantification is concerned; (c) minQuantity represents the minimum relative or absolute value of instances; (d) maxQuantity represents the maximum relative or absolute value of instances.

Note that the CAMEL user must specify correct minimum and maximum instance values in case the quantifier type is SOME: for absolute quantification, the maximum value should always be greater than or equal to the minimum one unless it equals to -1 (infinite) and both values should be integer-based, while, for relative quantification, not only the maximum value should be greater or equal to the minimum one but also both should be in the range [0.0,1.0].

Depending on which element is associated, *i.e.*, metric or property, a ConditionContext is further specialised into a PropertyContext and MetricContext. A Proper-

tyContext represents the context of the property to be measured and evaluated. A MetricContext represents the metric to be used in the evaluation of the condition. For composite metrics, CompositeMetricContext refers to the contexts of the composing metrics of the current metric. For raw metrics, RawMetricContext refers to the sensor that produces the measurements of this metric.

The example of metrics is available along with the example of scalability rules in Section 11.3.

11 Scalability Rules

The scalability package of the CAMEL metamodel is also based on SRL [15, 5]. SRL is inspired by the Esper Processing Language (EPL)[15] 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 describe and exemplify the main concepts in the scalability package.

11.1 Scalability Rules

A ScalabilityRule refers to an Event and a set of Actions. The Event represents either a single event or an event pattern that triggers the execution of the actions. The Actions can either specify which components and virtual machines are changed by the scalability rule (*i.e.*, case of scaling actions) and how or just remark that a global deployment decision has to be made (*i.e.*, case of event creation actions, see next sub-section). A ScalabilityRule refers to a set of ScaleRequirements that restrict how scaling actions are performed. It also refers to Entities such as the user or the organisation associated with the scalability rule (*cf.* Section 13).

Note that the CAMEL user must not specify conflicting scale requirements (*i.e.*, scale requirements of the same type which refer to the same internal component/virtual machine) or scaling actions which conflict with the scale requirements posed. A scale action conflicts with a scale requirement in the following two cases: (a) it is a HorizontalScalingAction, the requirement is of the respective type HorizontalScaleRequirement, and the amount of instances to scale-in or out does not conform to the range limit dictated by the requirement (*i.e.*, amount is greater than the difference between the upper and lower values in the range

[15] `http://esper.codehaus.org/`

limit); (b) it is a VerticalScalingAction, the requirement is of the respective type VerticalScaleRequirement and the update on a particular virtual machine characteristic is greater than the difference between the upper and lower values in the range limit dictated by the requirement for this virtual machine characteristic.

11.2 Actions

Figure 17 shows the part of the class diagram of the scalability package related to actions.

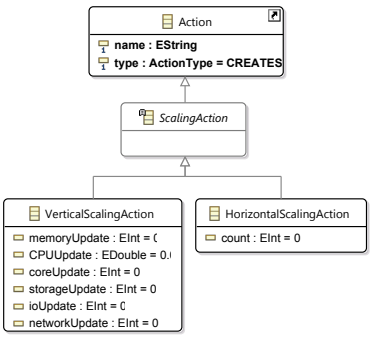

Figure 17: Part of the class diagram of the scalability package related to actions

An Action can be specialised into a ScalingAction. The ScalingAction, in turn, can be specialised into a HorizontalScalingAction or a VerticalScalingAction. The HorizontalScalingAction refers to a VM and an InternalComponent (both specified via the deployment package). In case such an action is executed, the specified component is scaled (out or in) along with the virtual machine hosting it. The property count defines the number of additional instances to create, or the number of existing instances to destroy. In contrast to horizontal scaling, the VerticalScalingAction refers to a concrete VMInstance. The properties *Update define the amount of virtual resources (*e.g.*, CPU cores, RAM, or storage) to be added to or removed from the virtual machine instance.

Note that the CAMEL user must provide correct action types for the corresponding actions. This means that HorizontalScalingActions must be mapped to action types of either SCALE_IN or SCALE_OUT and VerticalScalingActions must be mapped to action types of either SCALE_UP or SCALE_DOWN.

11.3 Events

Figure 18 shows the part of the class diagram of the scalability package related to events, while Figure 19 shows the portion related to enumeration types.

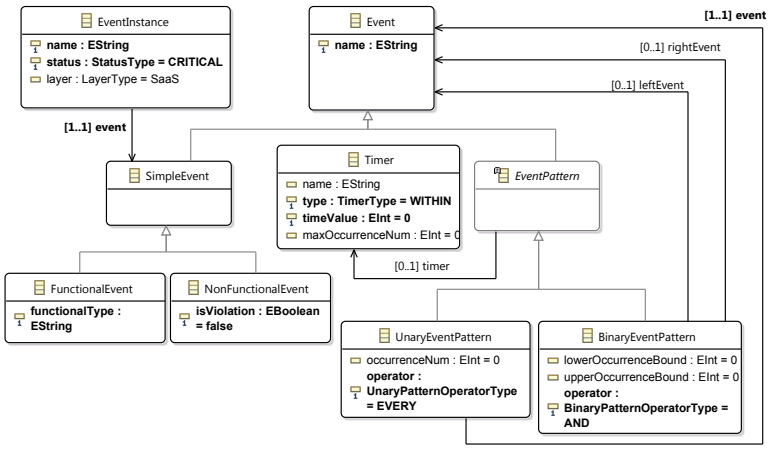

Figure 18: Part of the class diagram of the scalability package related to events

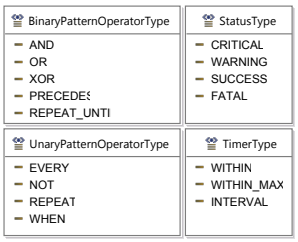

Figure 19: Part of the class diagram of the scalability package related to enumeration types

An Event can be specialised into a SimpleEvent or an EventPattern. The SimpleEvent, in turn, can be specialised into a FunctionalEvent or a NonFunctionalEvent. The FunctionalEvent represents a functional error (*e.g.*, a virtual machine or a component has failed). The NonFunctionalEvent represents the violation of a metric condition (*e.g.*, the response time of a component exceeds the target response time in an SLO). The NonFunctionalEvent refers to a MetricCondition, which defines the threshold for the metric.

An EventInstance represents the actual (measurement) data associated with a particular event that occurred in the system (*e.g.*, the actual measured value and the component producing the event). The property status represents the status of the event, *i.e.*, if it is fatal, critical, warning, or success. This property can provide useful insight (*e.g.*, for performing analysis on QoS) while also enabling the evaluation/assessment of the events.

Events are grouped by EventPatterns, which can be specialised into BinaryEventPatterns or UnaryEventPatterns.

A BinaryEventPattern uses a binary operator to associate either two Events with each other or one event with a Timer. The property operator can be set to one of the common, logical operators such as AND and OR, the order operator PRECEDES, and the occurrence operator REPEAT_UNTIL (see Figure 19). PRECEDES defines that an event has to occur prior to another one. REPEAT_UNTIL defines that an event has to occur multiple times until another event occurs. In this case, the CAMEL user should define the lower and/or upper bounds for the number of event occurrences.

A UnaryEventPattern refers to just one event along with a unary operator. The property operator can be set to NOT, EVERY, REPEAT, and WHEN. NOT defines that the negation of an event must be considered. EVERY defines that every occurrence of an event must be considered (*e.g.*, to define that a scalability rule must be triggered every time an event A occurs and not just once). REPEAT defines that an event has to occur multiple times. In this case, the CAMEL user must specify a value for the property occurrenceNum, which represents the number of event occurrences. WHEN defines that an event has to occur according to a particular time constraint defined by a Timer.

A Timer represents a time constraint for an event pattern. The property type represents the kind of timer. WITHIN defines that an event has to occur within a particular time frame. WITHIN_MAX defines that an event has to occur within a particular time frame, but only up to a specific number of times. In this case, the CAMEL user must specify a value for the property maxOccurrenceNum, which represents the maximum number of event occurrences. INTERVAL defines that an event has to occur after a particular amount of time.

The following are two examples that illustrate the composition of events in event patterns. (a) The condition A AND (B OR C) can be expressed as a BinaryEventPattern X_1 that comprises the SimpleEvent A and another BinaryEventPattern X_2 both connected by the AND operator. X_2, in turn, comprises the two SimpleEvents B and C connected by the OR operator. (b) The condition that either A or three times B occurs within two minutes can be expressed as a UnaryEventPattern U_1 that comprises a BinaryEventPattern B_1, the WHEN operator, and a Timer defining a two minutes threshold. B_1, in turn, comprises the SimpleEvent A and a UnaryEventPattern U_2 connected by the OR operator. Finally, U_2 comprises just the SimpleEvent B, the REPEAT operator, and a value of 3 for the property occurrenceNum.

Example

Assume that we have to specify scalability rules and metrics for the Scalarm use case. For the SimulationManager operating in three instances, the following pattern proved to be a good rule of thumb for triggering a scale out: (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*. Furthermore, it is sufficient to gather sensor values for the current CPU load every second. Suppose cpu_i represents the average CPU load for instance i, and cpu_j for instance j. In order to trigger the scalability rule, the following composite condition must be assessed:

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

Based on the above analysis, we created a scalability and metric model which specify 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.

Listing 9 shows the scalability model in textual syntax. This model encompasses one scalability rule that associates one binary event pattern with a scale-out action. The event pattern structure is also shown to illustrate the way the event conditions are specified. In this structure, the two non-functional events map to the conditions to be evaluated, while the binary event pattern impose the logical relation hierarchy between these conditions.

Listing 9: Scalarm scalability model

```
1  scalability model ScalarmScalability {
2
3    horizontal scaling action HorizontalScalingSimulationManager {
4      type: SCALE_OUT
5      vm: ScalarmModel.ScalarmDeployment.CPUIntensiveUbuntuGermany
6      internal component: ScalarmModel.ScalarmDeployment.
       SimulationManager
7    }
8
9    non-functional event CPUAvgMetricNFEAll {
10     metric condition: ScalarmModel.ScalarmMetric.
       CPUAvgMetricConditionAll
11     violation
12   }
13
14   non-functional event CPUAvgMetricNFEAny {
15     metric condition: ScalarmModel.ScalarmMetric.
       CPUAvgMetricConditionAny
16     violation
17   }
18
19   binary event pattern CPUAvgMetricBEPAnd {
20     left event: ScalarmModel.ScalarmScalability.CPUAvgMetricNFEAll
21     right event: ScalarmModel.ScalarmScalability.CPUAvgMetricNFEAny
22     operator: AND
23   }
24
25   scalability rule CPUScalabilityRule {
```

```
26      event: ScalarmModel.ScalarmScalability.CPUAvgMetricBEPAnd
27      actions [ScalarmModel.ScalarmScalability.
        HorizontalScalingSimulationManager]
28      scale requirements [ScalarmRequirement.
        HorizontalScaleSimulationManager]
29    }
30  }
31
32  requirement model ScalarmRequirement {
33
34    horizontal scale requirement HorizontalScaleSimulationManager {
35      component: ScalarmModel.ScalarmDeployment.SimulationManager
36      instances: 1..5
37    }
38  }
```

Listing 10 shows the metric model in textual syntax. The metrics mapping to the composite event structure in Listing 9 are shown along with their scheduling information. The two metrics map to common information for two families of metrics: (a) raw (sensor) metric measuring CPU load and (b) average CPU load metric, the latter one will be instantiated with two different contexts, once with a window of five minutes, and once with a window of one minute. So, the aggregated composite metrics are instantiated as metric instances two times per virtual machine, once per metric context.

Listing 10: Scalarm metric model

```
1   metric model ScalarmMetric {
2
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    property CPUProperty {
```

```
30      type: MEASURABLE
31      sensors [ScalarmMetric.CPUSensor]
32    }
33
34    sensor CPUSensor {
35      configuration: 'cpu_usage;de.uniulm.omi.cloudiator.visor.sensors.
        CpuUsageSensor'
36      push
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.Range_0_100
45    }
46
47    composite metric CPUAverage {
48      description: "Average of the CPU"
49      value direction: 1
50      layer: PaaS
51      property: ScalarmModel.ScalarmMetric.CPUProperty
52      unit: ScalarmModel.ScalarmUnit.CPUUnit
53
54      metric formula Formula_Average {
55        function arity: UNARY
56        function pattern: MAP
57        MEAN( ScalarmModel.ScalarmMetric.CPUMetric )
58      }
59    }
60
61    raw metric context CPUMetricConditionContext {
62      metric: ScalarmModel.ScalarmMetric.CPUMetric
63      sensor: ScalarmMetric.CPUSensor
64      component: ScalarmModel.ScalarmDeployment.SimulationManager
65      quantifier: ANY
66    }
67
68    raw metric context CPURawMetricContext {
69      metric: ScalarmModel.ScalarmMetric.CPUMetric
70      sensor: ScalarmMetric.CPUSensor
71      component: ScalarmModel.ScalarmDeployment.SimulationManager
72      schedule: ScalarmModel.ScalarmMetric.Schedule1Sec
73      quantifier: ALL
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 }
```

12 Providers

The provider package of the CAMEL metamodel is based on Saloon [24, 25, 26]. Saloon consists of a DSL along with a framework for specifying application requirements and user goals of cloud-based applications and selecting compatible cloud providers by leveraging upon feature models [2] and ontologies [12]. In the following, we describe and exemplify the main concepts in the provider package.

Figure 20 shows the class diagram of the provider package.

A ProviderModel has a root Feature and a set of Constraints. A Feature has a Feature Cardinality. The properties min and max represent the lower and upper bound of the cardinality, respectively, (where the upper bound can also take the value of -1 to represent infinity and upper should not be less than the lower value in the opposite case, *i.e.*, when positive) while the property value represents a value in this range. A Feature can also have sub-features and be specialised into Alternative, meaning that at least one feature in the group should be selected, or Exclusive, meaning that exactly one feature should be selected. Alternatives can also have a different Group Cardinality with arbitrary lower and upper bounds (*e.g.*, if the Alternative consists of a group of five choices, the Group Cardinality of 3..5 represents that at least three features in the group have to be selected). The variants of an Alternative should be also different from its sub-features.

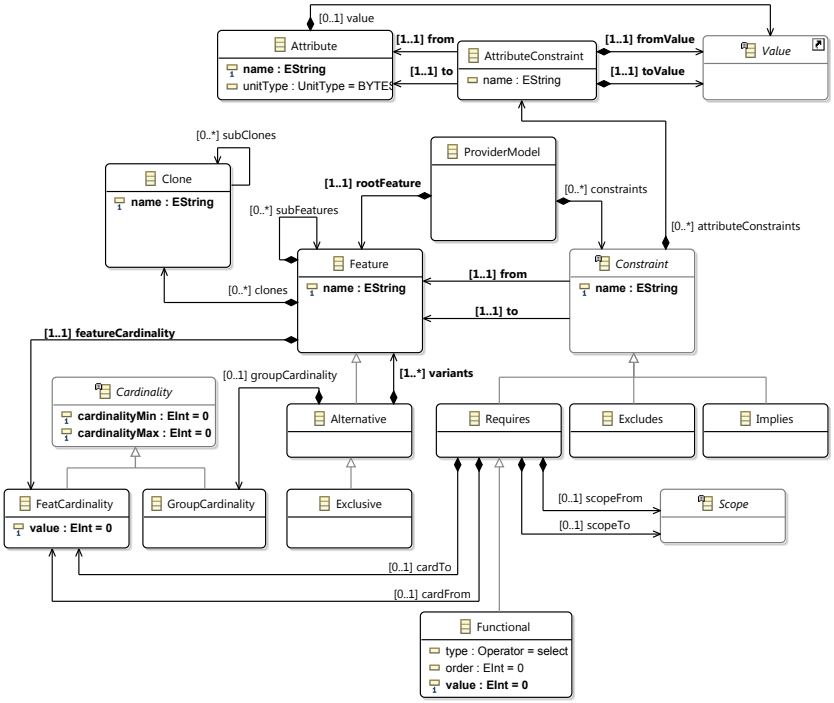

Figure 20: The class diagram of the provider package

A Constraint represents a typical restriction in binary feature models [14]. A Constraint can be an Implies constraint, meaning that a given feature requires another feature when selected (*i.e.*, both features have to be together in a valid configuration), or an Excludes constraint, meaning that one feature excludes another one when selected (*i.e.*, both features can not be together in a valid configuration). A Constraint can also be a Requires constraint, enabling the specification of restrictions of the form:

$$\boxed{[x', x'']A \rightarrow [y', y'']B} \text{ with } x', x'', y', y'' \in \mathbb{N}, \text{ and } x' \leq x'', y' \leq y''$$

This restriction represents that if cardinality of feature A is between x' and x'', the cardinality of feature B must be in $[y', y'']$.

A Requires constraint can be specialised into a Functional constraint, enabling the specification of restrictions of the form:

$$\boxed{[x', x'']A \rightarrow +[y]B} \text{ with } x', x'', y \in \mathbb{N}, \text{ and } x' \leq x''$$

This constraint represents that a feature A with cardinality between $[x', x'']$ requires y more instances of feature B in a valid configuration. Obviously, y should be positive in this case.

A Requires constraint can also be specialised into a Attribute Constraint, enabling the specification of restrictions of the form:

$$\boxed{(A).c = X \rightarrow (B).d = Y}$$

This constraint represents that if the attribute c of A has X as value, then the attribute d of B needs Y as value. Note that: (a) the attributes should be different, and (b) the value provided for the attributes should be included in the attributes' value type.

Example

Listing 11 shows an excerpt of the provider model for GWDG[16] specified using the CAMEL textual syntax, while Figure 21 depicts the same model using the FODA notation [14].

`root feature GWDG` specifies the attributes characterising the GWDG provider, such as the delivery model, service model, availability, driver, and endpoint. `attribute DeliveryModel` specifies that GWDG is a private cloud. `attribute ServiceModel` specifies that GWDG is a IaaS. `attribute Availability` specifies that the guaranteed availability of GWDG is 95%. `attribute Driver` specifies that the provider uses an OpenStack Nova API. `attribute EndPoint` specifies the endpoint of the provider's OpenStack Nova API.

`feature VM` specifies the attributes characterising the virtual machine flavours offered by the GWDG provider, 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 (*cf.* Section 17). For instance, `VMMemory` has `MemoryList`, a list of integer values (256, 512, 2048, etc.), as value type, and `MEGABYTES` as unit type.

`constraints` specifies the resources associated with the virtual machine flavours offered by the GWDG provider. For instance, the first `attribute constraint` specifies that the size of RAM of the `M1.LARGE` virtual machine flavour is 8192 (megabytes).

Listing 11: GWDG provider model (excerpt)

```
1  provider model GWDGProvider {
2
3    root feature GWDG {
4
5      attributes {
6
7        attribute DeliveryModel {
```

[16] http://www.gwdg.de/index.php

```
8         value: string value 'Private'
9         value type: ScalarmModel.GWDGType.StringValueType
10      }
11
12      attribute ServiceModel {
13        value: string value 'IaaS'
14        value type: ScalarmModel.GWDGType.StringValueType
15      }
16
17      attribute Availability {
18        unit type: PERCENTAGE
19        value: string value '95'
20        value type: ScalarmModel.GWDGType.StringValueType
21      }
22
23      attribute Driver {
24        value: string value 'openstack-nova'
25        value type: ScalarmModel.GWDGType.StringValueType
26      }
27
28      attribute EndPoint {
29        value: string value 'https://api.cloud.gwdg.de:5000/v2.0/'
30        value type: ScalarmModel.GWDGType.StringValueType
31      }
32    }
33
34    sub-features {
35
36      feature VM {
37
38        attributes {
39          attribute VMType {value type: ScalarmModel.GWDGType.
    VMTypeEnum}
40          attribute VMOS {value type: ScalarmModel.GWDGType.VMOsEnum}
41          attribute VMMemory {unit type: MEGABYTES value type:
    ScalarmModel.GWDGType.MemoryList}
42          attribute VMStorage {unit type: GIGABYTES value type:
    ScalarmModel.GWDGType.StorageList}
43          attribute VMCores {value type: ScalarmModel.GWDGType.
    CoresList}
44        }
45
46        feature cardinality {cardinality: 1 .. 8}
47      }
48
49      feature Location {
50
51        sub-features {
52
53          feature Germany {
54
55            feature cardinality {cardinality: 1 .. 1}
56          }
57        }
58
59        feature cardinality {cardinality: 1 .. 1}
60      }
61    }
62
```

```
63      feature cardinality {cardinality: 1 .. 1}
64    }
65
66    constraints {
67  ...
68      implies M1_LARGE_VM_Constraint_Mapping {
69
70        from: ScalarmModel.GWDGProvider.GWDG.VM
71        to: ScalarmModel.GWDGProvider.GWDG.VM
72
73        attribute constraints {
74
75          attribute constraint {
76            from: ScalarmModel.GWDGProvider.GWDG.VM.VMType
77            to: ScalarmModel.GWDGProvider.GWDG.VM.VMMemory
78            from value: string value 'M1.LARGE'
79            to value: int value 8192
80          }
81
82          attribute constraint {
83            from: ScalarmModel.GWDGProvider.GWDG.VM.VMType
84            to: ScalarmModel.GWDGProvider.GWDG.VM.VMCores
85            from value: string value 'M1.LARGE'
86            to value: int value 4
87          }
88
89          attribute constraint {
90            from: ScalarmModel.GWDGProvider.GWDG.VM.VMType
91            to: ScalarmModel.GWDGProvider.GWDG.VM.VMStorage
92            from value: string value 'M1.LARGE'
93            to value: int value 80
94          }
95        }
96      }
97  ...
98    }
99  }
100
101 type model GWDGType {
102
103   enumeration VMTypeEnum {
104     values [ 'M1.MICRO' : 0, ..., 'M1.LARGE' : 4, ..., 'C1.XXLARGE' :
        15 ]
105   }
106
107   enumeration VMOsEnum {
108     values [ 'Fedora 20 server x86_64' : 0, 'Ubuntu 14.04 LTS Server
        x86_64' : 1, ... ]
109   }
110
111   range MemoryRange {
112     primitive type: IntType
113     lower limit {int value 256 included}
114     upper limit {int value 32768 included}
115   }
116
117   range StorageRange {
118     primitive type: IntType
119     lower limit {int value 0 included}
```

```
120       upper limit {int value 160 included}
121     }
122
123     range CoresRange {
124       primitive type: IntType
125       lower limit {int value 1 included}
126       upper limit {int value 16 included}
127     }
128
129     string value type StringValueType {
130       primitive type: StringType
131     }
132
133     list StorageList {
134       values [ int value 0, int value 20, int value 40, int value 80,
          int value 160 ]
135     }
136
137     list MemoryList {
138       values [ int value 256, int value 512, int value 2048, int value
          4096, int value 8192, int value 16384, int value 32768 ]
139     }
140
141     list CoresList {
142       values [ int value 1, int value 2, int value 4, int value 8, int
          value 16 ]
143     }
144 }
```

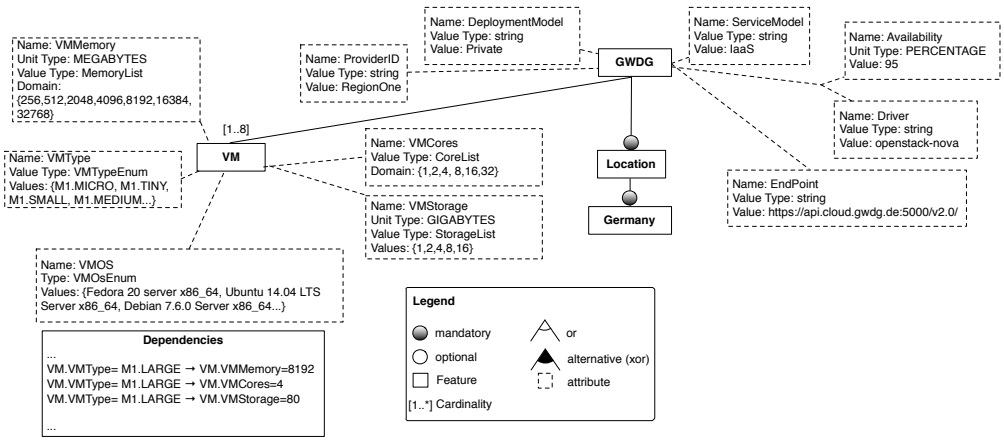

Figure 21: GWDG provider model diagram (Excerpt)

13 Organisations

The organisation package of the CAMEL metamodel is based on the organisation subset of CERIF [13]. CERIF is a modelling framework for specifying

organisations, users and other entities in the research domain. It is an EU recommendation[17] for information systems related to research databases used for standardising research information and fostering research information exchange. In the following, we describe and exemplify the main concepts in the organisation package.

Figure 22 shows the class diagram of the organisation package.

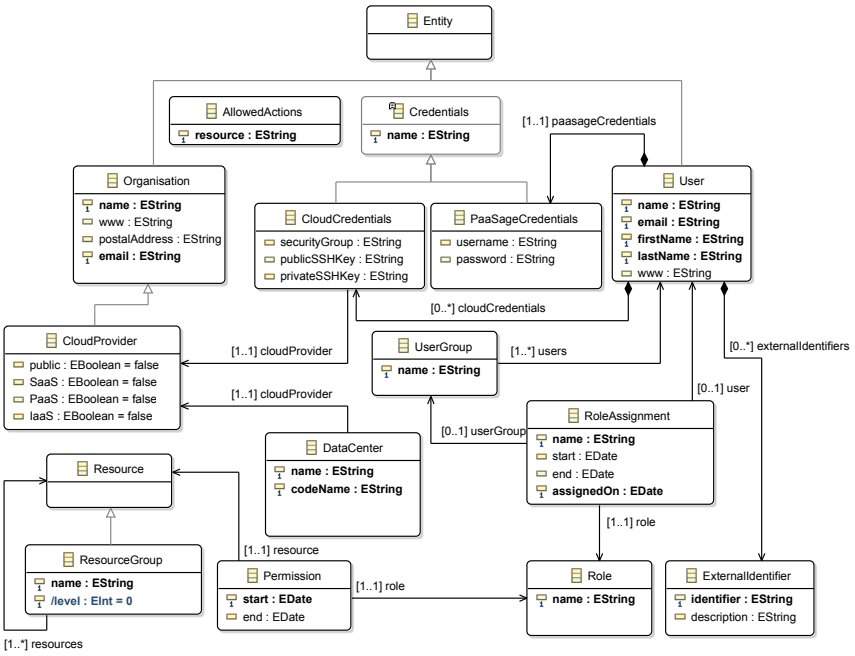

Figure 22: The class diagram of the organisation package

An Entity is a generic entity from CERIF. An entity can be an Organisation or a User.

An organisation can be specialised into a CloudProvider. The property public represents whether the cloud provider is public or not, while the properties PaaS, IaaS and SaaS represent whether the cloud provider supports the corresponding service models or not. A DataCenter represents a data centre of a cloud provider. The property codeName represents the internal code name used to identify the data centre at the cloud provider (*e.g.*, Amazon EC2 `eu-west-1`).

A user represents a person belonging to an organisation. It refers to PaaSageCredentials, which represent the credentials to authenticate a user in the PaaSage platform. It can also refer to CloudCredentials, which represent the credentials to authenticate and authorise the user in a cloud provider and hence enable performing tasks on behalf of the user on the cloud provider. A UserGroup represents a group of users.

[17] `http://cordis.europa.eu/cerif/`

A Role represents a role of a user or user group as in role-based access control (RBAC). A RoleAssignment represents the assignment of a role. It refers to either a user or a user group. The property assignmentTime represents the timestamp of the role assignment, while the properties startTime and endTime represent the start and end timestamps of the validity of the role.

A Permission represents the set of actions allowed to be performed by a role on a resource. It refers to Actions, which represent the actions themselves, and ResourceFilters, which represent filters on the sets of resources on which actions are performed. Filters can refer either to services (through a ServiceResourceFilter object), or information (through an InformationResourceFilter object). The properties startTime and endTime represent the start and end timestamps of the validity of the permission.

Example

Assume that we have to specify the organisation model for the Scalarm use case. Listing 12 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.

`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.

Listing 12: Scalarm organisation model

```
1  organisation model AGHOrganisation {
2
3    organisation AGH {
4      www: 'http://www.agh.edu.pl/en/'
5      postal address: 'al. Mickiewicza 30, 30-059 Krakow, Poland'
6      email: 'morzech@agh.edu.pl'
7    }
8
9    user MichalOrzechowski {
10     first name: Michal
11     last name: Orzechowski
12     email: 'morzech@agh.edu.pl'
13     password: '************'
14   }
15
16   role devop
17
18   role assignment MichalOrzechowskiDevop {
19     start: 2016-02-26
20     end: 2017-02-26
21     assigned on: 2016-02-25
22     user: AGHOrganisation.morzech
23     role: ScalarmModel.AGHOrganisation.devop
```

```
24     }
25 }
```

14 Security

The security package of the CAMEL metamodel provides the concepts to specify security aspects. It aims at enabling: (i) the specification of security requirements that can be used for filtering cloud providers; and (ii) the specification of security levels in SLOs, metrics, and scalability rules that can be used for adapting a cloud-based application in case of violations of the specified security levels.

Figure 23 shows the class diagram of the security package.

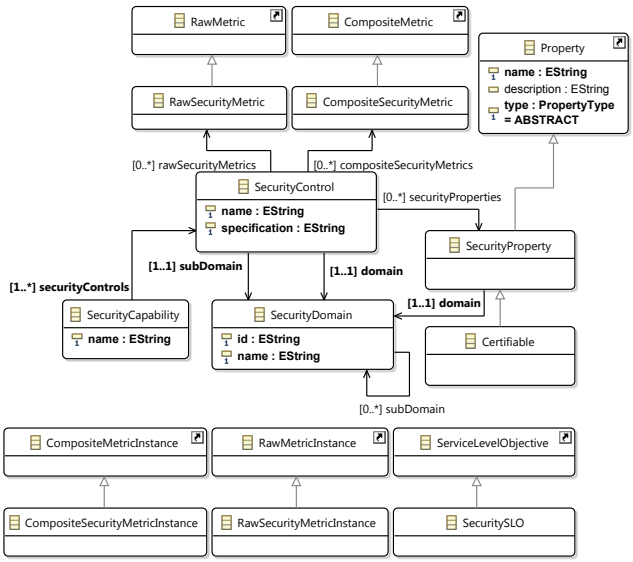

Figure 23: The class diagram of the security package

A SecurityControl represents a technical or administrative countermeasure that is aimed at addressing a security risk in a cloud-based application. The property specification is used to specify textual specifications of security controls provided by the different service providers specified in the CAMEL model.

A security control refers to the Raw- and CompositeSecurityMetrics, which specialise Raw- and CompositeMetrics (*cf.* Section 10), respectively, and represent the raw and composite security metrics associated with the security control. It also has two references to SecurityDomain, which represents the security domain associated with the security control (*e.g.*, "Identity & Access Management") and allows grouping security controls, properties, and metrics.

A SecurityProperty specialises Property (*cf.* Section 10) and represents an abstract security property that is not measurable. A Certifiable specialises security property and represents a certifiable security property that is measurable. It refers to a particular security metric.

A SecuritySLO specialises ServiceLevelObjective (*cf.* Section 8) and represents a security SLO.

Example

Assume that we have to specify the security model for the Scalarm use case. Listing 12 shows this specification in textual syntax.

`domain IAM` specifies the security 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 security property associated with the security domain IAM. `security control IAM_02` specifies a security control associated with the security sub-domain (CLCPM) and the property `IdentityAssurance`. Similarly, `security control IAM_11` specifies a security control associated with the security 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)[18].

`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: A set of security-related models which could be used to extend the Scalarm user case model

```
1  security model ScalarmSecurity {
2
3    domain IAM {
4      name: "Identity & Access Management"
5      sub-domains [ScalarmSecurity.IAM_CLCPM, ScalarmSecurity.IAM_CLCPM]
6    }
7
8    domain IAM_CLCPM {
9      name: "Credential Life Cycle/Provision Management"
10   }
11
12   domain IAM_UAR {
```

[18] https://cloudsecurityalliance.org/

```
13      name: "User Access Revocation"
14    }
15
16    property IdentityAssurance {
17      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."
18      type: ABSTRACT
19      domain: ScalarmSecurity.IAM
20    }
21
22    security control IAM_02 {
23      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."
24      domain: ScalarmSecurity.IAM
25      sub-domain: ScalarmSecurity.IAM_CLCPM
26      security properties [ScalarmModel.ScalarmSecurity.
        IdentityAssurance]
27    }
28
29    security control IAM_11 {
30      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 (\eg,
        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."
31      domain: ScalarmSecurity.IAM
32      sub-domain: ScalarmSecurity.IAM_UAR
33      security properties [ScalarmModel.ScalarmSecurity.
        IdentityAssurance]
34    }
35
36    security capability SecCap {
37      controls [ScalarmSecurity.IAM_02, ScalarmSecurity.IAM_11]
38    }
39  }
40
41  requirement model ScalarmExtendedReqModel {
42
43    security requirement AllIAMsSupported {
44      controls [ScalarmSecurity.IAM_02, ScalarmSecurity.IAM_11]
45    }
46  }
47
48  organisation model AmazonExt {
49
50    provider Amazon {
51      www: "www.amazon.com"
```

```
52      email: "contact@amazon.com"
53      PaaS
54      IaaS
55      security capability [ScalarmModel.ScalarmSecurity.SecCap]
56    }
57  }
58
59  unit model ScalarmUnit {
60    time interval unit {sec: SECONDS}
61  }
```

15 Execution

The execution package of the CAMEL metamodel provides the concepts to record historical data about the application execution, such as the measurements produced and the SLO assessments performed. This historical data can be used for auditing purposes only, but also for optimising the CAMEL model to better exploit the available cloud infrastructures. In PaaSage, the execution model is automatically manipulated by the PaaSage platform during the execution phase (*cf.* Section 2), and so it should be in the general case too. In the following, we describe and exemplify the main concepts in the execution package.

Figure 24 shows the class diagram of the execution package.

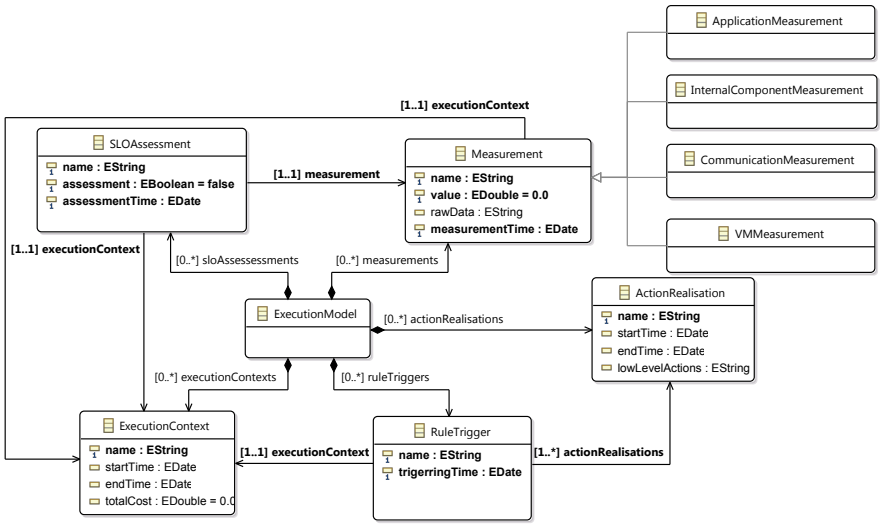

Figure 24: The class diagram of the execution package

An ExecutionContext represents the execution context for a particular deployment model. The properties startTime and endTime represent the date when the

application execution started and ended, respectively. The property totalCost represents the total cost for the application execution, calculated when the application execution has ended.

A Measurement represents a measurement produced during application execution. It refers to a `MetricInstance` (*cf.* Section 10) and an execution context. The property value represents the value of the measurement. The property rawData represents the raw data in the time series database (TSDB) (*cf.* D5.1.2 [6]). The property measurementTime represents the timestamp of the measurement. A Measurement can be an ApplicationMeasurement, an InternalComponentMeasurement, a CommunicationMeasurement, or a VMMeasurement, depending on which application or element within a deployment model it refers to.

An SLOAssessment represents an assessment of whether the metric condition in an SLO is violated or not. It refers to an execution context, a measurement, and an SLO (*cf.* Section 8). The property assessment represents whether the SLO was violated (value FALSE) or not (value TRUE). The property assessmentTime represents the timestamp of the assessment.

Similar to an SLO assessment, a RuleTrigger represents a triggering of a scalability rule. It refers to an execution context, a ScalabilityRule, and an EventInstance (*cf.* Section 11). The property triggeringTime represents the timestamp of the triggering. Additionally, a rule trigger refers to one or more ActionRealisations, which represent the adaptation actions performed when the scalability rule is triggered. The properties startTime and endTime represent the date when the adaptation started and ended, respectively. The property lowLevelActions represents the low level adaptation actions specific to a cloud provider.

Example

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

`vm binding ScalarmVMBinding` specifies the context for metric instances. It comprises a reference to the execution context and the virtual machine instance for which metric instances are produced (*cf.* Listing 6).

`execution context EC1` specifies the current execution context. It comprises a reference to the application being executed, a reference to the deployment model of the application, a reference to 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 (*cf.* Listing 16)

`vm measurement VM1` specifies the virtual machine measurement for the CPU metric instance. It comprises a reference to the execution context, a reference to the metric instance, a reference to the virtual machine instance (*cf.* Listing 6), 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: Sclarm execution model

```
metric model ScalarmMetric {

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

  raw metric instance RawCPUMetricInstance {
    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.
    CoreIntensiveUbuntuGermanyInst
    value: 95.0
    time: 2014-12-10
  }

  assessment A1 {
    execution context: ScalarmExecution.EC1
    measurement: ScalarmExecution.VM1
    slo: ScalarmRequirement.CPUMetricSLO
    violated
    time: 2014-12-10
  }
}
```

16 Types

The type package of the CAMEL metamodel is also based on Saloon [24, 25, 26]. It provides the concepts to specify value types and values used across CAMEL

models (*e.g.*, integer, string, or enumeration). In the following, we describe and exemplify the main concepts in the type package.

Figure 25 shows the class diagram of the type package.

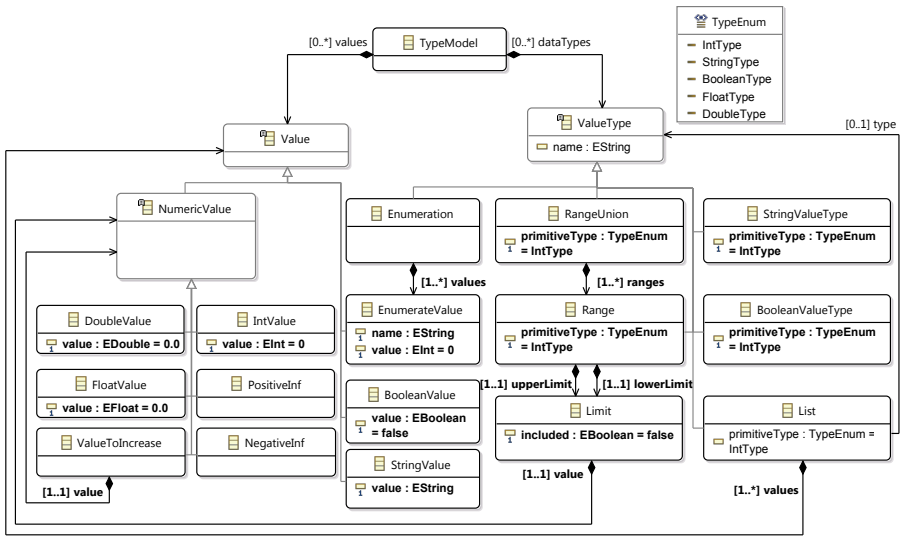

Figure 25: The class diagram of the type package

A Value represents a generic value. It can be specialised into a NumericValue, StringValue, BooleanValue, and EnumerateValue. A numeric value can be further specialised into the IntValue, DoubleValue, and FloatValue. The property `value` is typed by the corresponding Java type of Ecore. A numeric value can also be specialised into NegativeInf and PositiveInf, which represent negative and positive infinity, respectively, and can be used for specifying one of the two bounds of range-based value types. The StringValue and BooleanValue classes represent string and boolean values, respectively. The property `value` is typed by the corresponding Java type of Ecore. The EnumerateValue represents an enumerated value. The property `name` represents the string associated with the enumerated value, while the property `value` represents the integer associated with the value (or position in the enumeration).

ValueType represents a generic value type. It can be specialised into a StringValueType, BooleanValueType, Enumeration, List, Range and RangeUnion. StringValueType and BooleanValueType represent string and boolean value types, respectively. Enumeration represents an enumeration type that can take EnumerateValues. List represents a list type that can take either basic value type (*i.e.*, a numeric, string, or boolean value) or complex value type (*e.g.*, an enumeration or a range). The property `primitiveType` represents the basic value type, and it has to be used in the first case. The reference `type` represents the complex value type, and it has

to be used in the second case. A Range represents a range-based value type. It has two references to Limit, which represents the lower and upper limits for the value type of the range. The property `included` represents whether the lower and upper limits are included in the range or not. The RangeUnion represents a union of range-based value types. It refers to the contained range-based value types as well as to the primitive type that is common across all these contained range-based value types (*e.g.*, all range-based value types are integer-based).

Example

Assume that we have to record the types of the Scalarm use case. Listing 15 shows this specification in textual syntax.

The `range` statements specify two integer-based ranges and one double-based range. The first range is used to specify `CPUMetric` (*cf.* Listing 10 to represent that CPU metric values should be between 0 and 100, both included). The second range is used to specify the `ResponseTimeMetric` to represent that the response time values should be between 0, not included (*i.e.*, between 1), and 10000, included. The third range is used to specify the `AvailabilityMetric` to represent that the availability should be between 0.0 and 100.0, both included.

Listing 15: Scalarm type model

```
type model ScalarmType {

  range Range_0_100 {
    primitive type: IntType
    lower limit {int value 0 included}
    upper limit {int value 100}
  }

  range Range_0_10000 {
    primitive type: IntType
    lower limit {int value 0}
    upper limit {int value 10000 included}
  }

  range DoubleRange_0_100 {
    primitive type: DoubleType
    lower limit {double value 0.0 included}
    upper limit {double value 100.0 included}
  }
}
```

17 Units

The unit package of the CAMEL metamodel provides the concepts to specify units used across CAMEL models. These concepts 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. In the following, we describe and exemplify the main concepts in the unit package.

Figure 26 shows the class diagram of the unit package.

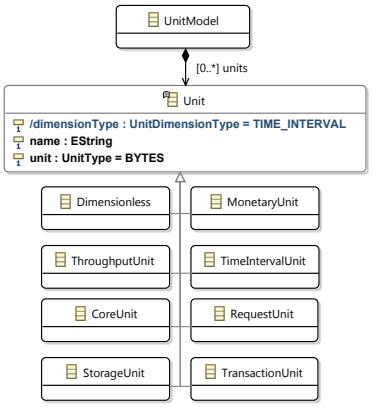

Figure 26: The class diagram of the unit package

A Unit represents an abstract unit. It can be specialised into the following concepts:

- CoreUnit, which represents the unit of CPU cores
- MonetaryUnit, which represents a monetary unit (*e.g.*, EUROS)
- RequestUnit, which represents the unit of number requests
- StorageUnit, which represents the unit of storage (*e.g.*, BYTES)
- ThroughputUnit, which represents the unit of throughput (*e.g.*, REQUESTS-_PER_SECOND)
- TimeIntervalUnit, which represents the unit of time interval (*e.g.*, SECONDS)
- TransactionUnit, which represents the number of transactions
- Dimensionless, which represents a unit without dimension (*e.g.*, a unit of PERCENTAGE is dimensionless)

The property unit refers to UnitType, which is an enumeration of all possible unit types. The property `dimensionType` refers to UnitDimensionType, which is an enumeration of all possible unit dimension types.

Figure 27 shows the class diagram of the enumerations in the unit package.

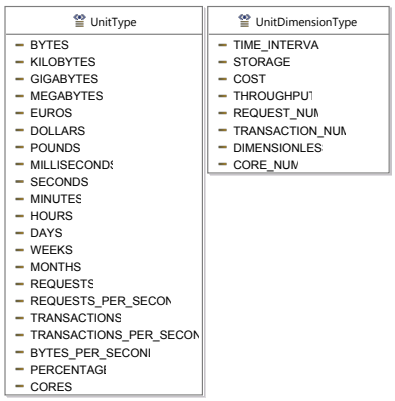

Figure 27: The class diagram of the enumerations in the unit package

Example

Assume that we have to specify the units of the Scalarm use case. Listing 16 shows this specification in textual syntax.

The unit model encompasses seven units that are used in the metric model (*cf.* Listing 10. The specification of each unit follows the pattern: `<unit_class> <unit_name>: <unit_type>`. For instance, `monetary unit {Euro: EUROS}` specifies a monetary unit named "euros" and typed `EUROS`).

Listing 16: Scalarm unit model

```
1  unit model ScalarmUnit {
2
3    monetary unit {Euro: EUROS}
4    throughput unit {SimulationsPerSecondUnit: TRANSACTIONS_PER_SECOND}
5    time interval unit {ResponseTimeUnit: MILLISECONDS}
6    time interval unit {ExperimentMakespanInSecondsUnit: SECONDS}
7    transaction unit {NumberOfSimulationsLeftInExperimentUnit:
       TRANSACTIONS}
8    dimensionless {AvailabilityUnit: PERCENTAGE}
9    dimensionless {CPUUnit: PERCENTAGE}
10 }
```

18 Java APIs and CDO

As mentioned, CAMEL consists of an Ecore model (*cf.* Section 6). This enables to specify CAMEL models using the CAMEL Textual Editor as well as to programmatically manipulate and persist them through Java APIs.

Listing 17 shows the creation of a `VM` of Scalarm (*cf.* Section 2). The classes that are instantiated and initialised in the code have been automatically generated by the EMF generator model based on the deployment package. All class

instances are obtained using the DeploymentFactory object specific for the deployment package. This object provides a set of methods that are used to make sure that the model objects are appropriately instantiated.

Listing 17: A sample VM definition

```
//create a ML VM
VM coreIntensiveVM = DeploymentFactory.eINSTANCE.createVM();
//First create VM requirement set & add it to the deployment model
VMRequirementSet coreIntensiveReqs = DeploymentFactory.eINSTANCE.
    createVMRequirementSet();
mlReqs.setName("CoreIntensiveReqs");
coreIntensiveVM.setVmRequirementSet(coreIntensiveReqs);
sensAppDeploymentModel.getVmRequirementSets().add(coreIntensiveReqs);
//Create a quantitative hardware requirement to include it in the
    requirement set
QuantitativeHardwareRequirement coreIntensiveRequirment =
    RequirementFactory.eINSTANCE.createQuantitativeHardwareRequirement
    ();
coreIntensiveRequirment.setName("CoreIntensive");
coreIntensiveRequirment.setMaxCores(32);
coreIntensiveRequirment.setMinCores(8);
coreIntensiveRequirment.setMaxRAM(8192);
coreIntensiveRequirment.setMinRAM(4096);
rm.getRequirements().add(coreIntensiveRequirment);
coreIntensiveReqs.setQuantitativeHardwareRequirement(
    coreIntensiveRequirment);
//Create a LOcation requirement imposing that the VM should be located
     in Scotland
LocationRequirement germanyRequirement = RequirementFactory.eINSTANCE.
    createLocationRequirement();
germanyRequirement.setName("GermanyReq");
germanyRequirement.getLocations().add(ScalarmLocationModel.germany);
rm.getRequirements().add(germanyRequirement);
coreIntensiveReqs.setLocationRequirement(germanyRequirement);
//Fix other details of the VM including its name and provided host
coreIntensiveVM.setName("CoreIntensiveVM");

ProvidedHost vmMLProv = DeploymentFactory.eINSTANCE.createProvidedHost
    ();
vmMLProv.setName("VMMLProv");

ml.getProvidedHosts().add(vmMLProv);
//Finally add the VM to the deployment model
scalarmDeploymentModel.getVms().add(ml);
```

Listing 18 shows the specification of Scalarm's ExperimentManager InternalComponent. This internal component has an associated Configuration, which specifies corresponding life cycle control scripts (*e.g.*, download and install commands). It also specifies communications with other internal components by creating corresponding RequiredCommunication, ProvidedCommunication entities and a dedicated type of host RequiredHost that the ExperimentManager requires.

Listing 18: A sample InternalComponent definition

```
//create a Scalarm InternalComponent give it a name
```

```
InternalComponent experimentManagerIc = DeploymentFactory.eINSTANCE.
    createInternalComponent();

//Associate it with a particular configuration
Configuration experimentManagerCompResourceConf = DeploymentFactory.
    eINSTANCE.createConfiguration();
experimentManagerCompResourceConf.setDownloadCommand("wget␣https://
    github.com/kliput/scalarm_service_scripts/archive/paasage.tar.gz␣&&
    ␣sudo␣apt-get␣update␣&&␣sudo␣apt-get␣install␣-y␣groovy␣ant␣&&␣tar␣-
    zxvf␣paasage.tar.gz␣&&␣cd␣scalarm_service_scripts-paasage");
experimentManagerCompResourceConf.setInstallCommand("cd␣
    scalarm_service_scripts-paasage␣&&␣./experiment_manager_install.sh"
    );
experimentManagerCompResourceConf.setStartCommand("cd␣
    scalarm_service_scripts-paasage␣&&␣./experiment_manager_start.sh");
experimentManagerIc.getConfigurations().add(
    experimentManagerCompResourceConf);

//Create a provided communication element on port 443
ProvidedCommunication experimentManagerProvidedCommunication =
    DeploymentFactory.eINSTANCE.createProvidedCommunication();
experimentManagerProvidedCommunication.setName("ExperimentManager");
experimentManagerProvidedCommunication.setPortNumber(443);
experimentManagerIc.getProvidedCommunications().add(
    experimentManagerProvidedCommunication);

//Create a required communication with Scalarm StorageManager
    component
RequiredCommunication experimentManagerReqStorageCommunication =
    DeploymentFactory.eINSTANCE.createRequiredCommunication();
experimentManagerReqStorageCommunication.setIsMandatory(true);
experimentManagerReqStorageCommunication.setName("
    ExperimentManager_consumes_SotrageManager");
experimentManagerReqStorageCommunication.setPortNumber(20001);
experimentManagerIc.getRequiredCommunications().add(
    experimentManagerReqStorageCommunication);

//Create a required host element
RequiredHost coreIntensiveUbuntuGermanyHostReq = DeploymentFactory.
    eINSTANCE.createRequiredHost();
coreIntensiveUbuntuGermanyHostReq.setName("
    coreIntensiveUbuntuGermanyHostReq");
experimentManagerIc.setRequiredHost(coreIntensiveUbuntuGermanyHostReq)
    ;

//Finally add the component to the deployment model
scalarmDeploymentModel.getInternalComponents().add(experimentManagerIc
    );
```

Listing 19 shows the creation of a `Communication` binding between the Scalarm's ExperimentManager and the Scalarm's StorageManager.

Listing 19: A sample `Communication` definition

```
//Create communication by also specifying its name and the provided
    and required communications
Communication experimentManagerToStorageManager = DeploymentFactory.
    eINSTANCE.createCommunication();
```

```
experimentManagerToStorageManager.setName("
    experimentManagerToStorageManager");
experimentManagerToStorageManager.setProvidedCommunication(
    storageManagerProvidedCommunicationMongodNnginx);
experimentManagerToStorageManager.setRequiredCommunication(
    experimentManagerReqStorageCommunication);

//Add communication to deployment model
scalarmDeploymentModel.getCommunications().add(
    experimentManagerToStorageManager);
```

Listing 20 shows the specification of a `Hosting` binding between the ExperimentManager `InternalComponent` and the `VM` of `coreIntensiveUbuntuGermanyHost` type.

Listing 20: A sample `Hosting` definition

```
//Create hosting, specify its name and the required and provided hosts
Hosting experimentManagerToCoreIntensiveUbuntuGermany =
    DeploymentFactory.eINSTANCE.createHosting();
experimentManagerToCoreIntensiveUbuntuGermany.setName("
    ExperimentManagerToCoreIntensiveUbuntuGermany");
experimentManagerToCoreIntensiveUbuntuGermany.setProvidedHost(
    coreIntensiveUbuntuGermanyHost);
experimentManagerToCoreIntensiveUbuntuGermany.setRequiredHost(
    coreIntensiveUbuntuGermanyHostReq);

//Add hosting to the deployment model
scalarmDeploymentModel.getHostings().add(
    experimentManagerToCoreIntensiveUbuntuGermany);
```

Listing 21 shows the process of saving a deployment model in a CDO repository. As mentioned, CDO uses a set of APIs that are designed after the JDBC APIs. In order to save a model, we need to first create a session and obtain a transaction over it. This example adopts a local database that is accessed using a TCP connector from the Net4j framework[19], a partner project used within CDO. Once the transaction is obtained, the deployment model refers to the `CDOResource` responsible for its persistence, and the transaction is committed.

Listing 21: Saving a deployment model in a CDO repository

```
//initialize and activate a container
final IManagedContainer container = ContainerUtil.createContainer();
Net4jUtil.prepareContainer(container);
TCPUtil.prepareContainer(container);
// CDONet4jUtil.prepareContainer(container);
container.activate();

// create a Net4j TCP connector
final IConnector connector = (IConnector) TCPUtil.getConnector(
    container, "localhost:2036");

// create the session configuration
```

[19] `https://www.eclipse.org/modeling/emf/?project=net4j`

```
CDONet4jSessionConfiguration config = CDONet4jUtil.
    createNet4jSessionConfiguration();
config.setConnector(connector);
config.setRepositoryName("repo1");

// create the actual session with the repository
CDONet4jSession cdoSession = config.openNet4jSession();

// obtain a transaction object
CDOTransaction transaction = cdoSession.openTransaction();

// create a CDO resource object
CDOResource resource = transaction.getOrCreateResource("/
    scalarmResource1");
EObject camelModel = ScalarmModel.createScalarmModel();

// associate the deployment model to the resource
resource.getContents().add(camelModel);

// commit the transaction to persist the model
transaction.commit();
```

Listing 22 shows the process of loading and modifying a deployment model. In this example, the host VM for the ExperimentManager is changed. We change it by setting the ProvidedHost of the Hosting to point to a virtual machine definition suitable for single CPU intensive tasks (assuming it is initially one suited for multi-core processing).

Listing 22: Loading and modifying a deployment model in a CDO repository

```
// open a new transaction
CDOTransaction transaction = cdoSession.openTransaction();

// load the existing resource of SensApp and get the top-most model
    which is a deployment one
CDOResource resource = transaction.getResource("/scalarmResource1");
assertTrue(resource.getContents().get(0) instanceof DeploymentModel);
DeploymentModel model = (DeploymentModel) resource.getContents().get
    (0);

// get provided host for the CPU intensive virtual machine (which we
    want to change to)
ProvidedHost CPUIntensiveProvidedHost = null;
for(VM vm: model.getVMs()){
  if (vm.getName().equals("CPUIntensiveUbuntuGermany")){
      CPUIntensiveProvidedHost = vm.getProvidedHost();
        break;
    }
}

// find the current hosting in the deployment model
Hosting currentHosting = null;
for (Hosting h2: model.getHostings()){
  if (h2.getName().equals("
      ExperimentManagerToCoreIntensiveUbuntuGermany")){
      currentHosting = h2;
        break;
```

```
    }
}

// modify hosting in the deployment model and replace the current
    provided host (which is presumably "CoreIntensiveUbuntuGermany")
currentHosting.setName("ExperimentManagerToCPUIntensiveUbuntuGermany")
    ;
currentHosting.setProvidedHost(CPUIntensiveProvidedHost);

// commit the transaction to persist the updated model
transaction.commit();
```

The example above show the Java code for programmatically saving, loading, and modifying part of a deployment model in a CDO repository. The Java code for programmatically saving, loading, and modifying models from other packages of the CAMEL metamodel is analogous. The full version of the Java code of the Scalarm example is available for reference in the Git repository at OW2[20].

19 Related Work

In the cloud community, libraries such as jclouds[21] or DeltaCloud[22] provide generic APIs abstracting over the heterogeneous APIs of IaaS providers, thus reducing cost and effort of deploying cross-cloud applications. More advanced frameworks such as Cloudify[23], Puppet[24], or Chef[25] provide DSLs that facilitate the specification and enactment of provisioning, deployment, monitoring, and adaptation of cloud-based applications, without being language-dependent. As for the research community, the mOSAIC [31] project tackles the vendor lock-in problem by providing an API for provisioning and deployment of cross-cloud applications. While these solutions effectively foster the deployment of cloud-based applications across multiple cloud infrastructures, they remain code-level solutions, which makes design changes difficult and error-prone. Moreover, these solutions do not provide support for models@run-time, which makes them unsuitable for enabling reasoning on the models and hence supporting self-adaptive cross-cloud applications.

The MODAClouds project[26] provides a family of DSLs collectively called MODACloudML. MODACloudML relies on the following three layers of ab-

[20] https://github.com/groundnuty/scalarm-paasage-camel/tree/master/model_in_java/src/main/java/eu/paasage/camel/agh

[21] http://www.jclouds.org

[22] http://deltacloud.apache.org/

[23] http://www.cloudifysource.org/

[24] https://puppetlabs.com/

[25] http://www.opscode.com/chef/

[26] http://www.artist-project.eu/

straction: *(i)* the *cloud-enabled computation independent model* (CCIM) to describe an application and its data, *(ii)* the CPIM (as in PaaSage) to describe cloud concerns related to the application in a cloud-agnostic way, and *(iii)* the CPSM (as in PaaSage) to describe the cloud concerns needed to provision and deploy the application on a specific cloud. In this respect, MODACloudML and CAMEL achieve similar goals, but with different approaches: while MODACloudML is a family of loosely coupled DSLs, CAMEL is a standalone language. The latter has the advantage of providing a minimal (although not small) set of modelling concepts along with a uniform syntax and semantics, which resulted from a non-trivial process of coupling and homogenising multiple DSLs [20].

The ARTIST project[27] provides the Cloud Application Modelling Language (CAML). CAML consists of an *internal* DSL [10] realised as a UML library along with UML profiles [3] rather than an *external* DSL such as CAMEL. The main rationale behind the latter stems from the goal of the ARTIST project to support migration of existing applications to the cloud, whereby UML models are reverse-engineered and tailored to a selected cloud environment. CAML and CAMEL both allow modelling multiple aspects of cross-cloud applications. However, the former does not support aspects of executions, while the latter provides full support for models@run-time, which facilitates self-adaptive cross-cloud applications.

The ARCADIA project[28] provides a methodology and a framework to support the development of highly-distributed applications (HDAs) that are reconfigurable by design. The ARCADIA Framework[28] 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[11] has multiple facets, such as the component model, the service graph model, the service deployment model, and the service run-time model. Different to CAMEL, however, the ARCADIA Context Model provides concepts that are specific to unikernel technology and microservices. CAMEL could possibly be extended to include such concepts.

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [23] is a specification developed by the OASIS consortium, which provides a language for specifying the components comprising the topology of cloud-based applications along with the processes for their orchestration. TOSCA supports the specification of types and templates, but not instances, in deployment models. In contrast, CAMEL supports the specification of types, templates, and instances. Therefore, in its current form, TOSCA can only be used at design-time, while CAMEL can be used at both design-time and run-time.

[27] `http://www.artist-project.eu/`

[28] `http://www.arcadia-framework.eu`

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 Noa Kuperberg from GigaSpaces. The work performed in this group will guarantee that the contribution of CAMEL will be partly integrated into the standard.

20 Conclusion and Future Work

CAMEL enables the specification of multiple aspects of cross-cloud applications, such as provisioning and deployment, service-level objectives, metrics, scalability rules, providers, organisations, users, roles, security requirements, execution contexts, and execution histories. It provides support for models@run-time, which makes it suitable for enabling reasoning on CAMEL models and hence supporting self-adaptive cross-cloud applications.

In this paper, we have described the modelling concepts, their attributes and their relations, as well as the rules for combining these concepts to specify valid models that conform to CAMEL. Moreover, we have exemplified how to specify models through the CAMEL Textual Editor as well as how to programmatically manipulate and persist them through CDO.

In the future, we will continue developing 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 (either within or outside the PaaSage platform). 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 already been adopted by another large project, namely CloudSocket[29]. This will guarantee the further development and validation of CAMEL in a wide variety of cloud computing settings.

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 to the challenge of maintaining multiple versions and automatically migrating CAMEL models [20] based on CDO and Edapt.

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

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

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 number 644690 (CloudSocket). Michal Orzechowski is also grateful to AGH University of Science and Technology for their support under grant number 15.11.230.212. The authors would like to thank the use case providers and component developers in PaaSage and CloudSocket for the constructive feedback on CAMEL.

References

[1] Colin Atkinson and Thomas Kühne. 'Rearchitecting the UML infrastructure'. In: *ACM Transactions on Modeling and Computer Simulation* 12.4 (2002), pp. 290–321. DOI: 10.1145/643120.643123.

[2] David Benavides, Sergio Segura and Antonio Ruiz Cortés. 'Automated analysis of feature models 20 years later: A literature review'. In: *Inf. Syst.* 35.6 (2010), pp. 615–636. DOI: 10.1016/j.is.2010.01.001.

[3] Alexander Bergmayr, Javier Troya, Patrick Neubauer, Manuel Wimmer and Gerti Kappel. 'UML-based Cloud Application Modeling with Libraries, Profiles, and Templates'. In: *CloudMDE 2014: 2nd International Workshop on Model-Driven Engineering on and for the Cloud*. Ed. by Richard F. Paige, Jordi Cabot, Marco Brambilla, Louis M. Rose and James H. Hill. Vol. 1242. CEUR Workshop Proceedings. CEUR, 2014, pp. 56–65. URL: http://ceur-ws.org/Vol-1242/paper7.pdf.

[4] Gordon S. Blair, Nelly Bencomo and Robert B. France. 'Models@run.time'. In: *IEEE Computer* 42.10 (2009), pp. 22–27. DOI: 10.1109/MC.2009.326.

[5] Jörg Domaschka, Kyriakos Kritikos and Alessandro Rossini. 'Towards a Generic Language for Scalability Rules'. In: *Advances in Service-Oriented and Cloud Computing—Workshops of ESOCC 2014*. Ed. by Guadalupe Ortiz and Cuong Tran. Vol. 508. Communications in Computer and Information Science. Springer, 2015, pp. 206–220. ISBN: 978-3-319-14885-4. DOI: 10.1007/978-3-319-14886-1_19.

[6] Jörg Domaschka et al. *D5.1.2—Product Executionware*. PaaSage project deliverable. Sept. 2015.

[7] Nicolas Ferry, Franck Chauvel, Alessandro Rossini, Brice Morin and Arnor Solberg. 'Managing multi-cloud systems with CloudMF'. In: *NordiCloud 2013: 2nd Nordic Symposium on Cloud Computing and Internet Technologies*. Ed. by Arnor Solberg, Muhammad Ali Babar, Marlon Dumas and Carlos E. Cuesta. ACM, 2013, pp. 38–45. ISBN: 978-1-4503-2307-9. DOI: 10.1145/2513534.2513542.

[8] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin and Arnor Solberg. 'Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems'. In: *CLOUD 2013: 6th IEEE International Conference on Cloud Computing*. Ed. by Lisa O'Conner. IEEE Computer Society, 2013, pp. 887–894. ISBN: 978-0-7695-5028-2. DOI: 10.1109/CLOUD.2013.133.

[9] Nicolas Ferry, Hui Song, Alessandro Rossini, Franck Chauvel and Arnor Solberg. 'CloudMF: Applying MDE to Tame the Complexity of Managing Multi-Cloud Applications'. In: *UCC 2014: 7th IEEE/ACM International Conference on Utility and Cloud Computing*. Ed. by Randall Bilof. IEEE Computer Society, 2014, pp. 269–277. DOI: 10.1109/UCC.2014.36.

[10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010. ISBN: 978-0321712943.

[11] Panagiotis Gouvas et al. *D2.2—Definition of the ARCADIA Context Model*. ARCADIA project deliverable. July 2015.

[12] Thomas R. Gruber. 'A translation approach to portable ontology specifications'. In: *Knowledge Acquisition* 5.2 (June 1993), pp. 199–220. ISSN: 1042-8143. DOI: 10.1006/knac.1993.1008.

[13] Keith Jeffery, Nikos Houssos, Brigitte Jörg and Anne Asserson. 'Research information management: the CERIF approach'. In: *IJMSO* 9.1 (2014), pp. 5–14. DOI: 10.1504/IJMSO.2014.059142.

[14] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak and A. Spencer Peterson. *Feature-Oriented Domain Analysis (FODA)—Feasibility Study*. Tech. rep. The Software Engineering Institute, 1990. URL: http://www.sei.cmu.edu/reports/90tr021.pdf.

[15] Kyriakos Kritikos, Jörg Domaschka and Alessandro Rossini. 'SRL: A Scalability Rule Language for Multi-Cloud Environments'. In: *CloudCom 2014: 6th IEEE International Conference on Cloud Computing Technology and Science*. Ed. by Juan E. Guerrero. IEEE Computer Society, 2014, pp. 1–9. ISBN: 978-1-4799-4093-6. DOI: 10.1109/CloudCom.2014.170.

[16] Kyriakos Kritikos et al. *D4.1.2—Product Database and Social Network System*. PaaSage project deliverable. Sept. 2015.

[17] Dariusz Król and Jacek Kitowski. 'Self-scalable services in service oriented software for cost-effective data farming'. In: *Future Generation Comp. Syst.* 54 (2016), pp. 1–15. DOI: 10.1016/j.future.2015.07.003.

[18] Thomas Kühne. 'Matters of (meta-)modeling'. In: *Software and Systems Modeling* 5.4 (2006), pp. 369–385. DOI: 10.1007/s10270-006-0017-9.

[19] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. Special Publication 800-145. National Institute of Standards and Technology, Sept. 2011.

[20] Nikolay Nikolov, Alessandro Rossini and Kyriakos Kritikos. 'Integration of DSLs and Migration of Models: A Case Study in the Cloud Computing Domain'. In: *Procedia Computer Science* 68 (2015). 1st International Conference on Cloud Forward: From Distributed to Complete Computing, pp. 53–66. ISSN: 1877-0509. DOI: 10.1016/j.procs.2015.09.223.

[21] Object Management Group. *Object Constraint Language*. 2.4. http://www.omg.org/spec/OCL/2.4/. Feb. 2014.

[22] Object Management Group. *Unified Modeling Language Specification*. 2.4.1. http://www.omg.org/spec/UML/2.4.1/. Aug. 2011.

[23] Derek Palma and Thomas Spatzier. *Topology and Orchestration Specification for Cloud Applications (TOSCA)*. Tech. rep. Organization for the Advancement of Structured Information Standards (OASIS), June 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cos01/TOSCA-v1.0-cos01.pdf.

[24] Clément Quinton, Nicolas Haderer, Romain Rouvoy and Laurence Duchien. 'Towards multi-cloud configurations using feature models and ontologies'. In: *MultiCloud 2013: International Workshop on Multi-cloud Applications and Federated Clouds*. ACM, 2013, pp. 21–26. ISBN: 978-1-4503-2050-4. DOI: 10.1145/2462326.2462332.

[25] Clément Quinton, Daniel Romero and Laurence Duchien. 'Cardinality-based feature models with constraints: a pragmatic approach'. In: *SPLC 2013: 17th International Software Product Line Conference*. Ed. by Tomoji Kishi, Stan Jarzabek and Stefania Gnesi. ACM, 2013, pp. 162–166. ISBN: 978-1-4503-1968-3. DOI: 10.1145/2491627.2491638.

[26] Clément Quinton, Romain Rouvoy and Laurence Duchien. 'Leveraging Feature Models to Configure Virtual Appliances'. In: *CloudCP 2012: 2nd International Workshop on Cloud Computing Platforms*. ACM, 2012, pp. 21–26. ISBN: 978-1-4503-1161-8. DOI: 10.1145/2168697.2168699.

[27] Alessandro Rossini. 'Cloud Application Modelling and Execution Language (CAMEL) and the PaaSage Workflow'. In: *Advances in Service-Oriented and Cloud Computing—Workshops of ESOCC 2015*. Ed. by Antonio Celesti and Philipp Leitner. Vol. 567. Communications in Computer and Information Science. Springer, 2016, pp. 437–439. ISBN: 978-3-319-33313-7. DOI: 10.1007/978-3-319-33313-7.

[28] Alessandro Rossini, Franck Chauvel, Panagiotis Gouvas, Anastasios Zafeiropoulos, Costantinos Vassilakis and Eleni Fotopoulou. *D3.1a—Smart Controller Reference Implementation*. ARCADIA project deliverable. Apr. 2016.

[29] Alessandro Rossini, Kiriakos Kritikos, Nikolay Nikolov, Jörg Domaschka, Frank Griesinger, Daniel Seybold and Daniel Romero. *D2.1.3—CAMEL Documentation*. PaaSage project deliverable. Oct. 2015.

[30] Alessandro Rossini, Adrian Rutle, Yngve Lamo and Uwe Wolter. 'A formalisation of the copy-modify-merge approach to version control in MDE'. In: *Journal of Logic and Algebraic Programming* 79.7 (2010), pp. 636–658. DOI: 10.1016/j.jlap.2009.10.003.

[31] Calin Sandru, Dana Petcu and Victor Ion Munteanu. 'Building an Open-Source Platform-as-a-Service with Intelligent Management of Multiple Cloud Resources'. In: *UCC 2012: 5th IEEE International Conference on Utility and Cloud Computing*. IEEE Computer Society, 2012, pp. 333–338. ISBN: 978-1-4673-4432-6. DOI: 10.1109/UCC.2012.54.

[32] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming (2nd edition)*. Addison-Wesley Professional, 2011. ISBN: 978-0-321-75302-1.