CAMEL Documentation v2015.9

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

30th September 2015

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 [15]. 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 multi-cloud applications (*i.e.*, applications that can be deployed across multiple private, public, or hybrid cloud infrastructures), which allow 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).

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

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

In this document, we provide the final version of the documentation of CAMEL. In this respect, we revise and extend the initial version of the documentation (*cf.* D2.1.2 [25]). In particular, we describe the modelling concepts, their attributes and their relations, as well as the rules for combining these concepts to specify valid 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.

Intended Audience

This document is a public document intended for readers with some experience in cloud computing and software engineering, and some familiarity with the initial architecture design (*cf.* D1.6.1 [9]) as well as CAMEL (*cf.* D2.1.1 [27]).

For the use case partners in PaaSage, this document provides the documentation that will facilitate modelling their use cases in CAMEL.

For the research partners in PaaSage, this document provides the documentation that will facilitate integrating CAMEL with the components of the PaaSage platform.

For the external reader, this document provides the documentation that will facilitate adopting CAMEL outside the PaaSage platform.

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 multi-cloud applications (*i.e.*, applications deployed across multiple private, public, or hybrid cloud infrastructures), which allow for 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) [19]. 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 multi-cloud applications, PaaSage adopts the Cloud Application Modelling and Execution Language (CAMEL). CAMEL integrates and extends existing DSLs, namely Cloud Modelling Language (CloudML) [6, 4, 5], Saloon [23, 22, 24], and the Organisation part of CERIF [8]. In addition, CAMEL integrates new DSLs developed within the project, such as the Scalability Rule Language (SRL) [11, 3].

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

The abstract syntax of a language describes the set of concepts, their attributes, and their relations, as well as the rules for combining these concepts to specify valid statements that conform to this abstract syntax. The concrete syntax of a language describes the textual or graphical notation that renders these concepts, their attributes, and their relations.

In MDE, the abstract syntax of a DSL is typically defined by its metamodel. That is, a metamodel describes the set of modelling concepts, their attributes, and their relations, as well as the rules for combining these concepts to specify valid models that conform to the metamodel [19]. Moreover, in MDE, the concrete syntax may vary depending on the domain, *e.g.*, a DSL could provide a textual notation as well as a graphical notation along with the corresponding serialisation in XML Metadata Interchange (XMI) [20].

In this document, we describe the abstract syntax of CAMEL and exemplify how to specify models using a textual editor as well as how to programmatically manipulate and persist them through Java APIs.

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, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, and 17 present the various packages of the CAMEL metamodel along with corresponding sample models. 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 some conclusions and outlines some plans for future work.

2 CAMEL and the PaaSage workflow

In order to facilitate the integration across the components managing the life cycle of multi-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 (see Figure 1).

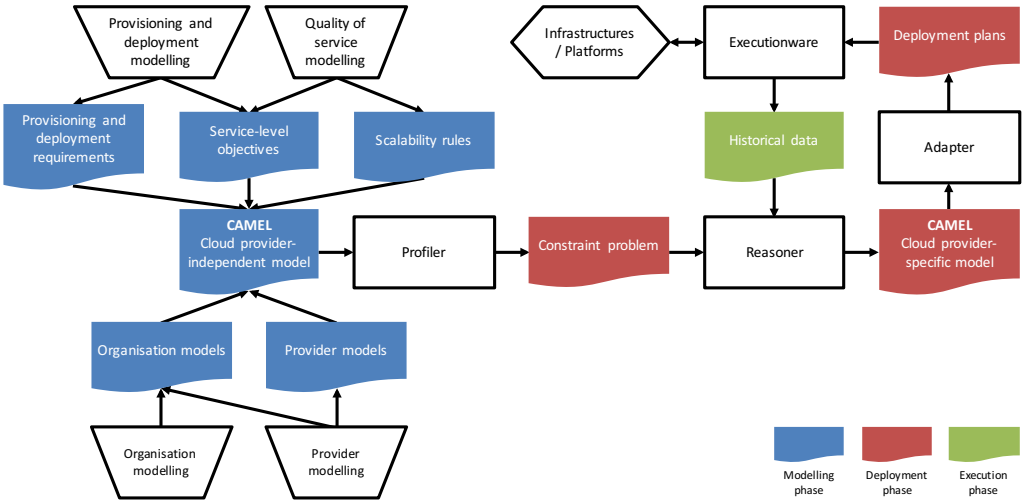

Figure 1: CAMEL models in the PaaSage workflow

Modelling phase The PaaSage users design a *cloud provider-independent model* (CPIM), which specifies the deployment of a multi-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 SensApp (see `http://sensapp.org/`), an open-source, service-oriented application for storing and exploiting large data sets collected from sensors and devices. Figure 2(a) shows the CPIM in graphical syntax. It consists of a SensApp servlet, which is hosted by a Tomcat servlet container, which in turn is hosted by a GNU/Linux virtual machine. Moreover, the SensApp servlet communicates with a MongoDB database, which is hosted by a GNU/Linux virtual machine in a data centre in Norway. Finally, the SensApp servlet must have a response time below 100 ms.

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 multi-cloud application along with its requirements and objectives in a cloud provider-specific way.

For instance, the Profiler could match the CPIM of SensApp 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 two SensApp servlet *instances*, which are hosted by two Tomcat container instances, which in turn are hosted by two Ubuntu 14.04 virtual machine instances at Amazon EC2 in the EU. Moreover, the SensApp servlet instances communicate with a MongoDB database instance, which is hosted by a CentOS 7 virtual machine instance 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. It also records historical data about the application execution, which allows the Reasoner to continuously revise the solution to the constraint problem to better exploit the cloud infrastructures.

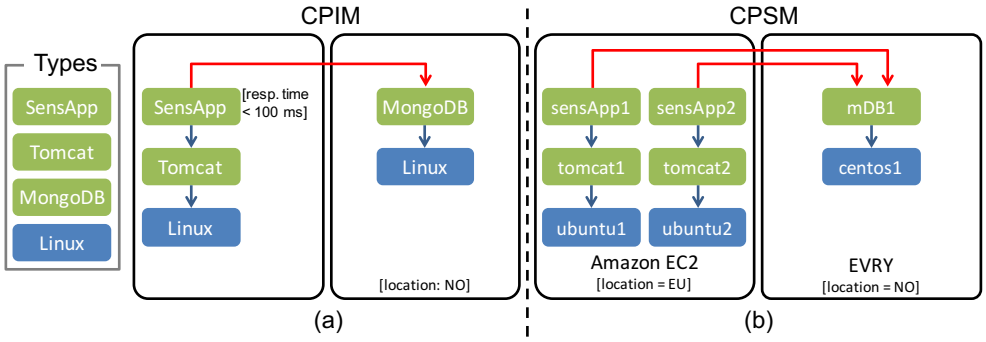

Figure 2: Sample CAMEL models: (a) CPIM; (b) CPSM

3 Technologies

In order to design and implement CAMEL, we adopted the Eclipse Modelling Framework (EMF)[1] along with Object Constraint Language (OCL) [17], Xtext[2], and Connected Data Objects (CDO)[3]. In this section, we outline these technologies and describe how they fit the requirements of the PaaSage platform.

3.1 Eclipse Modeling Framework (EMF)

EMF is a modelling framework that facilitates defining DSLs. EMF provides the Ecore metamodel, which is the core metamodel of EMF and 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).

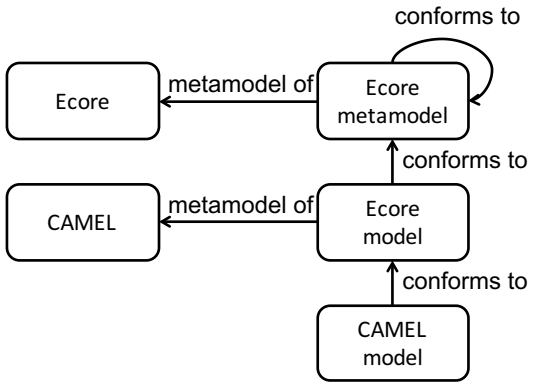

Figure 3: Ecore-based modelling stack in PaaSage

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) (see `https://www.eclipse.org/modeling/gmp/`) or Graphical Editing Framework (GEF) (see `https://www.eclipse.org/gef/`) to manually create a custom graphical editor.

[1] `https://www.eclipse.org/modeling/emf/`
[2] `https://eclipse.org/Xtext/`
[3] `https://www.eclipse.org/cdo/`

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 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 Meta-Object Facility (MOF)[16] models and metamodels. OCL is also an integral part of the Queries/Views/Transformations (QVT)[18] specification, where it is adopted in the context of model transformation. The Eclipse Model Development Tools (MDT) (see `http://www.eclipse.org/modeling/mdt/`) project includes the Eclipse OCL (see `http://wiki.eclipse.org/OCL`) component, which 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. However, the target PaaSage users (*i.e.*, DevOps), tend to prefer textual syntax over graphical syntax to specify CAMEL models. Therefore, we adopted Xtext to provide a textual syntax for CAMEL. Xtext 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 modellers not only to specify CAMEL models but also to syntactically and semantically validate them.

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 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 distribute their models. It provides features that satisfy the design-time and run-time requirements of the PaaSage platform, such as:

- 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.
- 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.
- Versioning: CDO supports *optimistic* versioning[26], 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. 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: Mars - http://download.eclipse.org/releases/mars
- 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: Mars - http://download.eclipse.org/releases/mars

- 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 this documentation**
- Edit the file

5 Naming Conventions

All elements in a CAMEL model are identified by a name. The name should be a string with no quotes surrounding it. The string should be a concatenation in meaningful names in CamelCase syntax. For instance, if we have to specify an average response time metric, we should use the name `AverageResponseTime`.

A reference in a CAMEL model (in textual syntax) should be a string with no quotes surrounding it. The string should 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 should use the fully qualified name `ScalarmModel.ScalarmMetric.AverageResponseTime`.

6 CAMEL Metamodel

CAMEL is designed as single metamodel organised into packages, whereby each package reflects the aspect (or domain) covered by the package.

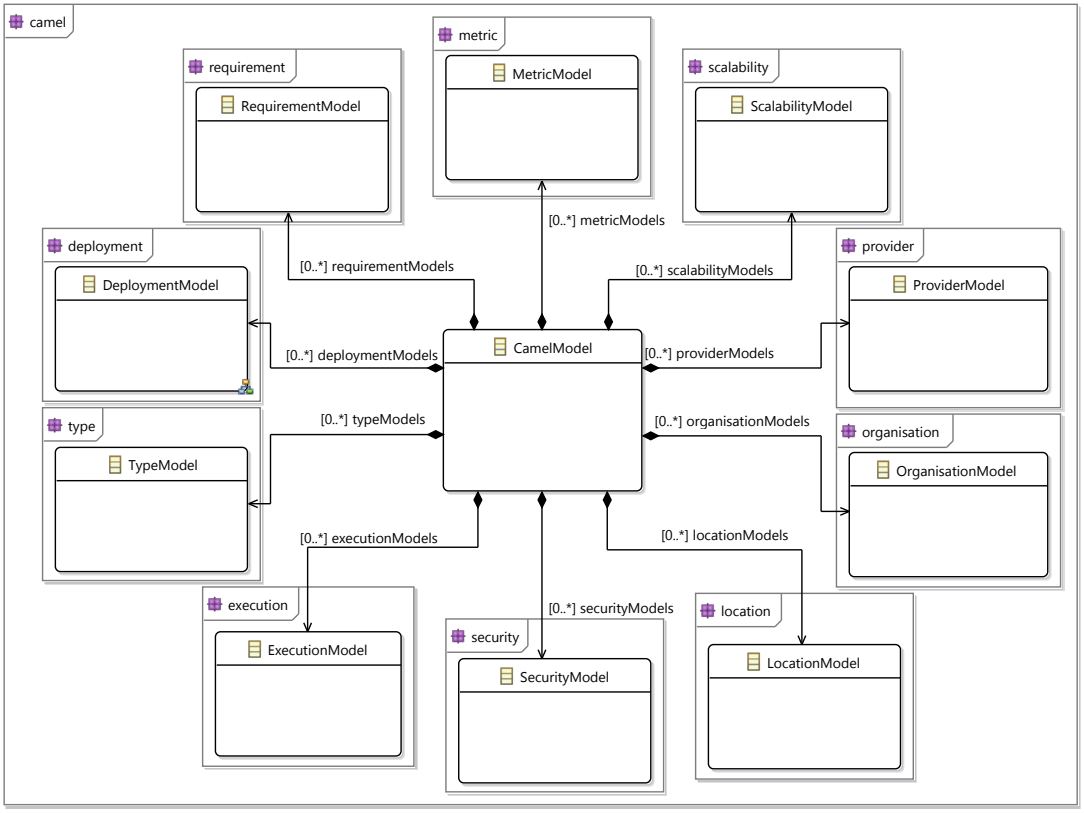

Figure 4: Class diagram of the CAMEL metamodel including packages

Figure 4 shows the top-level camel package of the CAMEL metamodel. A CamelModel is a collection of sub-models, which are discussed in detail in the following sections. These models are 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), and TypeModels (*cf.* Section 16).

7 Deployment

The deployment package of the CAMEL metamodel is based on CloudML[4] [6, 4, 5], which was developed in collaboration with the MODAClouds project[5]. CloudML consists of a tool-supported DSL for modelling and enacting the provisioning and deployment of multi-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 multi-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.

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 [29], 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.

To this end, 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* [14]. 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.

[4] http://cloumdml.org

[5] http://www.modaclouds.eu

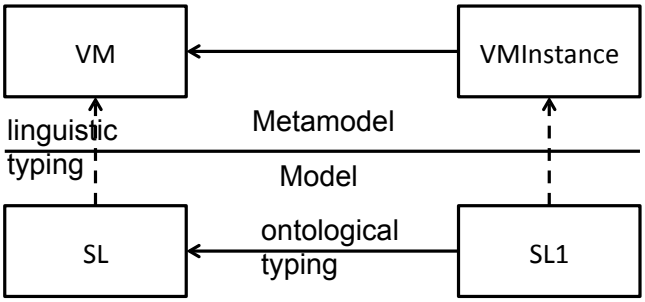

Figure 5: Linguistic and ontological typing

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

Figure 6 shows the type portion 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.

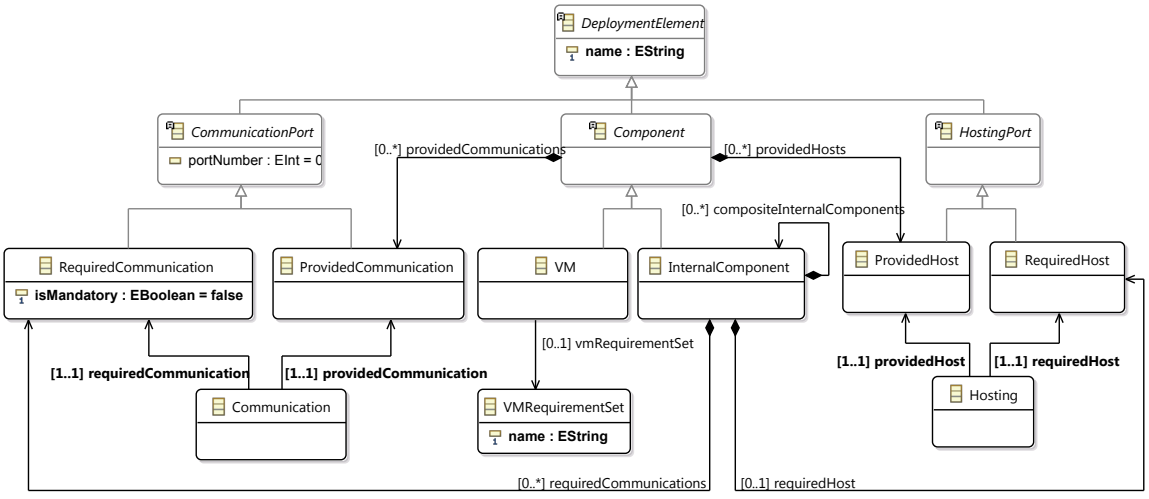

Figure 6: Class diagram of the type part of the deployment package

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

7.1 Components

A Component (see Figure 6) represents a reusable type of application component. A component can be an InternalComponent managed by the PaaSage platform, or a

VM (short for virtual machine) managed by the cloud provider. A virtual machine or a deployment model can refer to a VMRequirementSet, which represents a set of requirements for a single virtual machine or for all virtual machines, respectively, such as hardware requirements, operating system requirements, location requirements, etc. (*cf.* Section 8).

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.

In the following, we adopt the Scalarm[6] use case as a running example to exemplify how to specify CAMEL models in textual syntax. The complete Scalarm CAMEL model in textual syntax can be downloaded from the CAMEL Git repository at: `https://tuleap.ow2.org/plugins/git/paasage/camel`.

Example

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

The `internal component ExperimentManger` specifies a reusable type of the component Experiment Manager. The `provided communication ExpManPort` represents that the Experiment Manager provides its features through port 443. The `required communications 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 Manger has to be deployed and started before the Execution Manger). The `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 2).

The `configuration ExperimentManagerConfiguration` specifies the commands to handle the life cycle of the Experiment Manager. `download`, `install`, and `start`

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

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 `upload`, `configure`, and `stop` commands of a component.

The aforementioned commands are used by the Executionware during the execution phase (*cf.* Section 7.5) to enact the deployment of the application components and manage their life cycles.

Listing 1: A sample `InternalComponent`

```
1  deployment model ScalarmDeployment {
2    internal component ExperimentManager {
3      provided communication ExpManPort {port: 443}
4      required communication StoManPortReq {port: 20001 mandatory}
5      required communication InfSerPortReq {port: 11300}
6      required host CoreIntensiveUbuntuGermanyHostReq
7
8      configuration ExperimentManagerConfiguration {
9        download: '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'
10       install: 'cd scalarm_service_scripts-paasage && ./
       experiment_manager_install.sh'
11       start: 'cd scalarm_service_scripts-paasage && ./
       experiment_manager_start.sh'
12     }
13   }
14 ...
```

Then, assume that we have to specify the virtual machine on which the Experiment Manger can be deployed. Listing 2 shows this specification in textual syntax.

The `requirement set CoreIntensiveUbuntuGermanyRS` represents 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 6), 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).

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

Note that, although not shown in this example, it is possible to specify `configurations` for virtual machines the same way it is done for internal components.

Listing 2: A sample VM

```
...
  requirement set CoreIntensiveUbuntuGermanyRS {
    quantitative hardware: ScalarmRequirement.CoreIntensive
    os: ScalarmRequirement.Ubuntu
    location: ScalarmRequirement.GermanyReq
  }

  vm CoreIntensiveUbuntuGermany {
    requirement set CoreIntensiveUbuntuGermanyRS
    provided host CoreIntensiveUbuntuGermanyPort
  }
...
```

7.2 Communications

A Communication represents a reusable type of communication binding between a required- and a provided communication. The property type of Communication specifies whether the component requesting the feature and the component providing the feature have to be deployed on the same virtual machine instance (value LOCAL), on separate virtual machine instances (value REMOTE), or on any of the two options (value ANY).

Example

Assume that we have to specify the communication binding between the Experiment Manager and the Storage Manager. Listing 3 shows this specification in textual syntax.

The communication ExperimentManagerToStorageManager specifies a reusable type of communication binding between the Experiment Manager and the Storage Manager. The 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. The type: REMOTE specifies that the Experiment Manager and the Storage Manager must be deployed on separate virtual machine instances.

Listing 3: A sample Communication

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

7.3 Hostings

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

Example

Assume that we have to specify the hosting binding between the Experiment Manager and the virtual machine with a large number of CPU cores, running operating system Ubuntu, and located in Germany. Listing 4 shows this specification in textual syntax.

The `hosting ExperimentManagerToCoreIntensiveUbuntuGermany` specifies a reusable type of hosting binding between the Experiment Manager and the virtual machine with a large number of CPU cores, running operating system Ubuntu, and located in Germany. 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 4: A sample `Hosting`

```
1 ...
2   hosting ExperimentManagerToCoreIntensiveUbuntuGermany {
3     from ExperimentManager.CoreIntensiveUbuntuGermanyPortReq to
      CoreIntensiveUbuntuGermany.CoreIntensiveUbuntuGermanyPort
4   }
5 ...
```

7.4 Component, Communication, and Hosting instances

The types presented above can be instantiated in order to form a CPSM.

Note that these instances are added automatically by the Reasoner during the deployment phase (*cf.* Section 2) and should not be specified by the PaaSage users unless explicitly necessary.

Example

Listing 5 shows the specification of instances of the components, virtual machines, communications, and hostings from the previous examples (*cf.* Listings 1, 2, 3, and 4).

The `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 10), which is compatible with the requirement set of the virtual machine template `CoreIntensiveUbuntuGermany` (*cf.* Listing 8).

The `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 5: Sample instances of `internal component`, `vm`, `communication`, and `hosting`

```
...
  vm instance CoreIntensiveUbuntuGermanyInst typed ScalarmModel.
    ScalarmDeployment.CoreIntensiveUbuntuGermany {
    vm type: ScalarmModel.GWDGProvider.GWDG.VM.VMType
    vm type value: ScalarmModel.GWDGType.VMTypeEnum.M1.LARGE
    provided host instance CoreIntensiveUbuntuGermanyHostInst typed
    CoreIntensiveUbuntuGermany.CoreIntensiveUbuntuGermanyHost
  }

  internal component instance StorageManagerInst typed ScalarmModel.
    ScalarmDeployment.StorageManager {
    provided communication instance StoManPortInst typed
    StorageManager.StoManPort
    required communication instance InfSerPortReqInst typed
    StorageManager.InfSerPortReq
    required host instance StorageIntensiveUbuntuGermanyHostReqInst
    typed StorageManager.StorageIntensiveUbuntuGermanyHostReq
  }

  internal component instance ExperimentManagerInst typed ScalarmModel
    .ScalarmDeployment.ExperimentManager {
    provided communication instance ExpManPortInst typed
    ExperimentManager.ExpManPort
    required communication instance StoManPortReqInst typed
    ExperimentManager.StoManPortReq
    required communication instance InfSerPortReqInst typed
    ExperimentManager.InfSerPortReq
    required host instance CoreIntensiveUbuntuGermanyHostReqInst typed
     ExperimentManager.CoreIntensiveUbuntuGermanyHostReq
  }

  connect ExperimentManagerInst.StoManPortReqInst to
    StorageManagerInst.StoManPortInst typed ScalarmModel.
    ScalarmDeployment.ExperimentManagerToStorageManager named
    ExperimentManagerToStorageManagerInst

  host ExperimentManagerInst.CoreIntensiveUbuntuGermanyHostReqInst on
    CoreIntensiveUbuntuGermanyInst.CoreIntensiveUbuntuGermanyHostInst
    typed ScalarmModel.ScalarmDeployment.
    ExperimentManagerToCoreIntensiveUbuntuGermany named
    ExperimentManagerToCoreIntensiveUbuntuGermanyInst
...
```

7.5 Interplay with Executionware

It is important to understand that, in order to execute an application, which means deploying and executing at least one instance of each of its components, the Executionware relies on the information provided in the deployment model within a CAMEL model. The Executionware does not make up anything nor does it do any magic. 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.

For internal components and their instances in particular, handlers are called in the following order (note that upload is currently not supported):

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 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[7]. This has a consequence on user handling and networking. As for the users, this means that all commands are executed as `root` and also that no other existing users can be assumed. 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.

[7] `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:

- `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 Second, 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 1 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 1 is mapped to the following three environment variables; all consisting of a sequence of `ipv4:port` separated by comma (,).

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

- CLOUD_STOMANPORTREQ: provices 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>

- CONTAINER_STOMANPORTREQ: provices 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>

At the time being it is up to the user to decide which of the combinations; the user chooses for his set-up. Usually, the public ips and ports are a save bet even though they 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 you may not want to use more than one command in the install handler, as *e.g.*, `cd directory && ./run.sh` will not work just like `cd directory ; ./run.sh`.

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.

8 Requirements

The requirement package provides the concepts to specify requirements for multi-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 portion of the class diagram of the requirement package related to its main concepts.

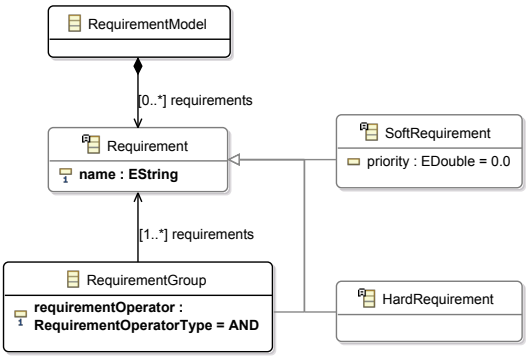

Figure 7: Portion of the class diagram of the requirement package related to its main concepts

A RequirementModel is a collection of Requirements. A requirement can be a HardRequirement, such as a service level objective (SLO) (*e.g.*, response time < 100ms), meaning that it is measurable and the PaaSage platform must satisfy it, or a SoftRequirement, such as a optimisation objective (*e.g.*, maximise performance), meaning that it is not measurable and the PaaSage platform will only aim at satisfying it with no guarantees. The property priority of SoftRequirement represents the priority of the soft requirements. These priorities allow the PaaSage platform to rank these soft requirements when reasoning on the application and generating a new CPSM for it. In particular, these priorities allow the generation of the utility function of the constraint problem corresponding to the end-user's soft requirements. For instance, these priorities could 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. In this way, the utility function could be generated as the weighted sum of the functions of the metrics involved in the end-user's soft requirements.

A RequirementGroup represents a logical group of requirements, which can be comprised of single requirements or requirement groups. The property requirementOperator of RequirementGroup represents the logical operator that is used to

connect these requirements and can take values AND (logical conjunction) or OR (logical disjunction). A requirement group refers to one or more Applications 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 to create a requirement group tree, which represents a tree of logically connected requirements (*e.g.*, service levels) that must be satisfied. For instance, a top-level requirement group (*e.g.*, identified with 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 with the name `Alternative`$_i$) could in turn contain single requirements, such as SLOs, logically connected by the AND operator. Note that a requirement group tree should not contain cycles (*i.e.*, requirement groups should not contain other requirements groups of a higher level of the requirement group tree). Finally, since each requirement group refers to a set of applications, a requirement group tree should not contain sub-requirement groups that refer to different sets of applications.

8.2 Hardware, OS & Image and Provider Requirements

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

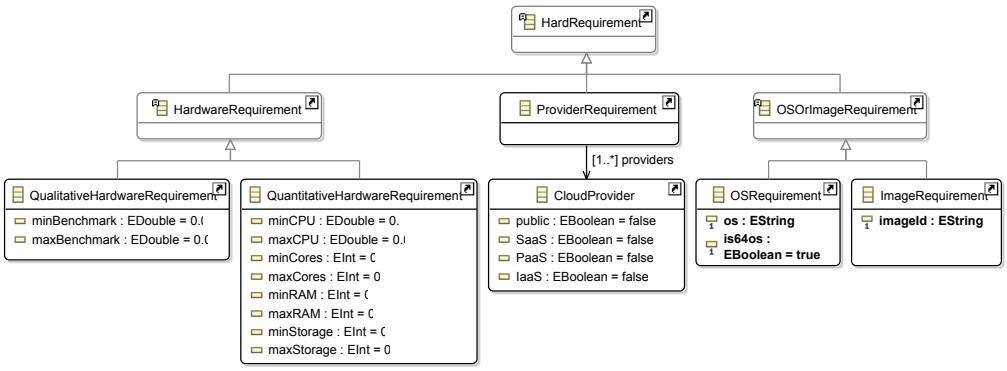

Figure 8: Portion of the class diagram of the requirement package related to hardware, OS, image, and provider requirements

A HardwareRequirement is a hard requirement. It can be referred to by a VMRequirementSet (*cf.* Section 7). It can be a QualitativeHardwareRequirement, which represents a requirement on the performance of a virtual machine instance, or a QuantitativeHardwareRequirement, which represent 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. Note that the benchmarking itself is currently not implemented by the PaaSage platform. 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 all these minimum and maximum properties, at least one of the bounds must be specified.

A OsOrImageRequirement is a hard requirement. It can be referred to by a VMRequirementSet (*cf.* Section 7). It can be 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", "Windows", etc.), 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. Note that the image must first be uploaded to the PaaSage platform in order to obtain an identifier before this identifier can be used in an image requirement.

A ProviderRequirement is a hard requirement. It can be referred to by a VMRequirementSet (*cf.* Section 7). It represents the set of cloud providers that must be considered for an application deployment (*e.g.*, Amazon and Rackspace only).

8.3 Location Requirements

Figure 9 shows the portion of the class diagram of the requirement package related to location and security requirements.

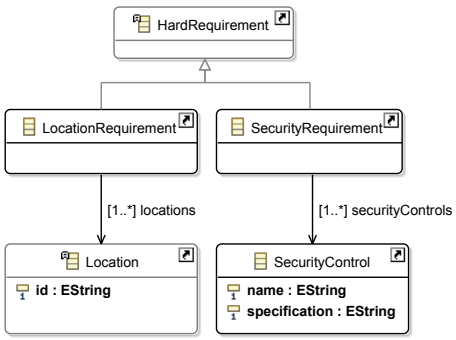

Figure 9: Location and security requirements classes

A LocationRequirement is a hard requirement. It can be attached to the specification of the requirements for a VM, or to a whole deployment model. In

the former case, it specifies that instances of the VM type must be deployed in a particular location. In the latter case, it specifies the locations of all VM instances within a deployment model. 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).

8.4 Security Requirements

A SecurityRequirement is a hard requirement and represents a security 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 component, then the security controls must be enforced on all applications and components specified in the CAMEL model.

8.5 Scale Requirements

Figure 10 shows the portion 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 the way scaling actions can be performed. A 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.

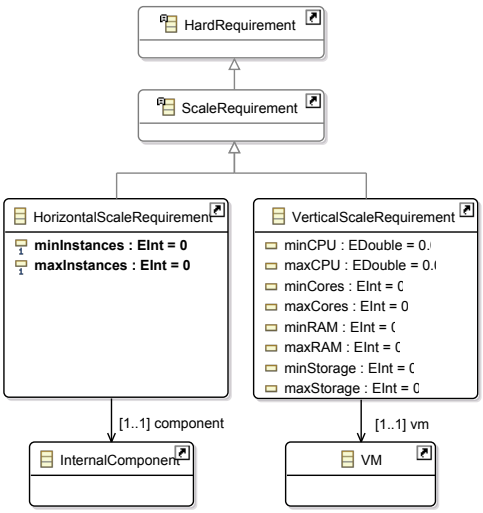

Figure 10: Scaling requirements classes

8.6 Service Level Objectives

Figure 11 shows the portion of the class diagram of the requirement package related to SLOs and optimisation requirements.

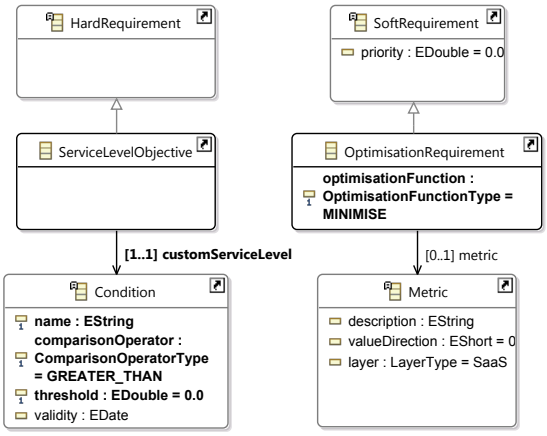

Figure 11: SLOs and optimisation requirements classes

A ServiceLevelObjective is a hard requirement and represents an SLO. SLOs are used to specify measurable performance objectives (*e.g.* availability, response time, throughput, etc.) of a cloud service provider. In CAMEL, ServiceLevelObjectives refer to a Condition, such as MetricCondition (*cf.* Section 10), which represents the metric condition that must be satisfied.

8.7 Optimisation Requirements

An OptimisationRequirement is a soft requirement. It refers to a Metric (*cf.* Section 10), which represents the metric 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 take values MINIMISE or MAXIMISE.

8.8 Example

Assume that we have to specify the requirements for the components of the Scalarm use case. Listing 6 shows this specification in textual syntax.

The `quantitative hardware` requirement `CoreIntensive` specifies that a VM must have from 8 to 32 CPU cores and from 4 to 8 GB of RAM. It is referred to by the `requirement set CoreIntensiveUbuntuGermanyRS` in the `deployment model ScalarmDeployment` (*cf.* Listing 2).

The `os` requirement `Ubuntu` specifies that a VM must run Ubuntu operating system 64-bit edition. It is referred to by the `requirement set CoreIntensiveUbuntuGermanyRS` in the `deployment model ScalarmDeployment` (*cf.* Listing 2).

The `location requirement GermanyReq` specifies that a VM must be deployed in Germany. It is referred to by the `requirement set CoreIntensiveUbuntuGermanyRS` in the `deployment model ScalarmDeployment` (*cf.* Listing 2). `locations` refers to the location `DE` in the location model `ScalarmLocation` (*cf.* Listing 7).

The `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 2).

The `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 9).

The `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 9), while `component` refers to the `internal component ExperimentManager` in the `deployment model ScalarmDeployment` (*cf.* Listing 2).

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

Listing 6: Sample 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]
  }

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

  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]
  }
}
```

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 shows the class diagram of the location package.

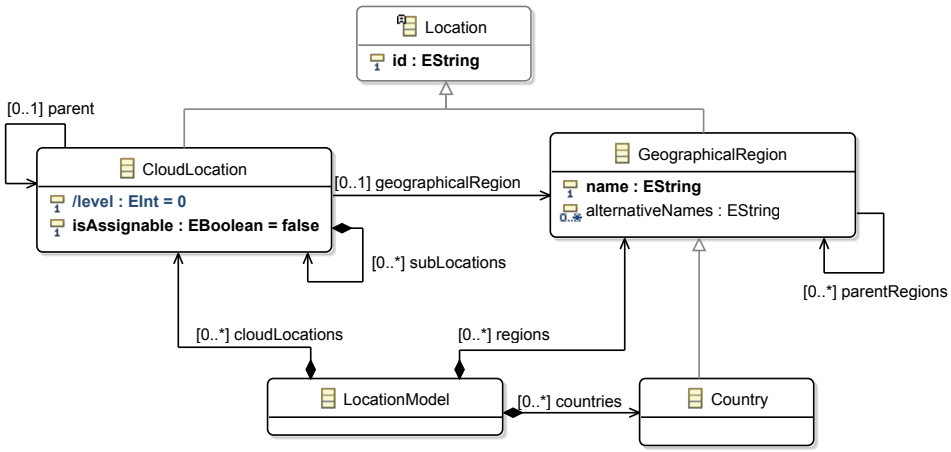

Figure 12: Class diagram of the location package

A Location represents a physical or virtual location. It can be 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. A GeographicalLocation can be a Country, which represents a distinct entity in political geography. Similar to the geographical region, a cloud location can refer to parent location, which allows creating hierarchies of cloud locations. Note that both the geographical region and the cloud location should not refer to themselves.

9.1 Example

Assume that we have to specify the locations for the Scalarm use case. Listing 7 shows this specification in textual syntax.

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

Listing 7: Sample location model

```
location model ScalarmLocation {
  region EU {
    name: 'Europe'
  }

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

  country UK {
    name: 'United Kingdom'
    parent regions [ScalarmLocation.EU]
  }
}
```

10 Metrics

The metric package of the CAMEL metamodel is based on the Scalability Rules Language (SRL) [11, 3]. SRL consists of a tool-supported DSL for specifying rules that support complex adaptation scenarios of multi-cloud applications. SRL aims at compensating for the limitations of existing solutions by providing the necessary modelling concepts for specifying behavioural patterns of multi-cloud applications, as well as the scaling actions to change the provisioning and deployment in response to these patterns. SRL is inspired by the OWL-Q language [12] with respect to the specification of quality of service (QoS) metrics. In the following, we describe and exemplify the main concepts in the metric package.

10.1 Metrics and Properties

Figure 13 shows the portion of the class diagram of the metric package related to its main concepts.

In order to identify event patterns in a scalability rule or to define SLOs, the components and virtual machines must be monitored. A Property represents a non-functional property of a component or virtual machine. The attribute type represents the kind of property, where a value of MEASURABLE denotes that the property can be measured, such as response time or availability, while a value of ABSTRACT denotes that the property is a non-measurable, abstract property realised 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.

A Metric represents a generic metric encapsulating the details for measuring properties (*e.g.*, an *availability* metric), which can refer to either raw measurements or aggregations on them. It can be a RawMetric, which represents a metric leading to the production of raw measurements, or a CompositeMetric, which represents a metric computed from other metrics. A metric, whether raw or composite, also refers to a measurable Property, which represents the property measured. Furthermore, it refers to the Unit of measurement (*e.g.*, the PERCENTAGE unit for an availability 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.

A specific layer in the cloud stack is also associated to a metric. IaaS or PaaS denote that the metric is related to the measurement of IaaS or PaaS services, respectively, while SaaS denotes that the metric is related to the measurement of an application as a whole, or its (internal or external) components, like third-party SaaS. Additional (sub-)layers further distinguish different levels

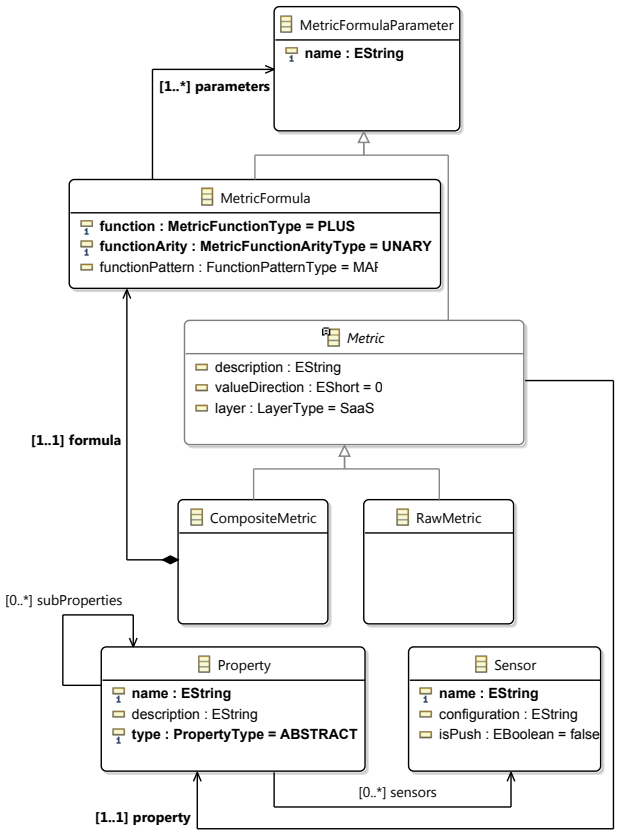

Figure 13: Portion of the class diagram of the metric package related to its main concepts

in the SaaS layer: SCC (short for Service Composition) denotes that service composition is concerned, while BPM (short for Business Process Management) denotes that business processes are concerned. The difference between these two layers lies on the fact that business processes are usually realised through service compositions and are therefore at a higher abstraction level. In fact, there must be a correlation between the metrics involved in these two sub-layers. For instance, a key performance indicator, which is critical for a business organisation, can be related to a specific metric which is computed from a metric used in a specific SLO for the service composition realising this business process.

Each CompositeMetric refers to a MetricFormula, which represents the aggregation function used over a set of composing metrics. A MetricFormula refers to one or more MetricFormulaParameters, which represent the input for the formula, and a pre-defined function that defines the operation to be executed by the formula. There exist three types of parameters. In case of constants, the parameter refers to a Value, while in case of non-constant parameters the parameter refers to Met-

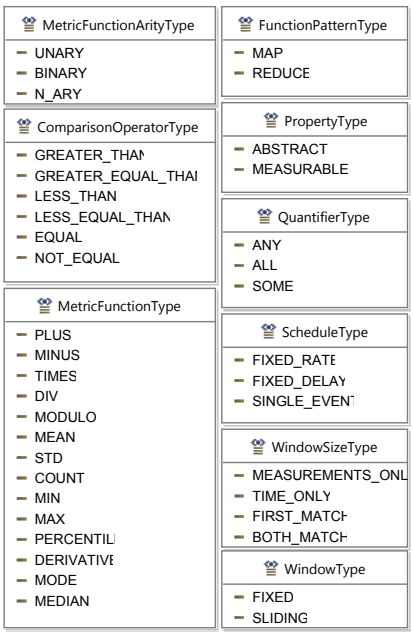

Figure 14: Portion of the class diagram of the metric package related to enumeration types

rics or MetricFormulas. In this way, metric formulas can involve not only constant values and other metrics but also recursive calls to other metric formulas. 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 defines the semantics of the operation by also restricting the number of parameters, their kind, and the way they are combined. For instance, MEAN should map only to one parameter, which should be of type metric or metric formula. The FunctionPatternType represents the number of metric instances that are created.

Figure 14 shows the portion of the class diagram of the metric package related to its main concepts.

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 VM or component. To this end, an instance of a metric references a particular object binding (see MetricObjectBinding class), which can be classified as a MetricComponentBinding, a MetricVMBinding, and a MetricApplicationBinding. Such information is crucial for properly configuring the monitoring system as it enables for instance to define on which virtual machine (*i.e.*, the entity or the virtual machine on which the entity is deployed) a sensor (*i.e.*, the measurement mean) will be deployed. A MetricComponentBinding associates a particular component

and respective application with a metric and leads to the deployment of one or more sensors reporting measurements for this component. The MetricVMBinding associates a virtual machine with a metric also indicating the need of deploying one or more sensors on this particular virtual machine to measure it. 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.

A Metric does not have measured values attached to it while, on the other hand, a MetricInstance has. To this end, a metric instance is classified into a raw metric instance and a composite metric instance. A raw metric instance refers to a particular sensor which is used in order to produce the respective measurements. The property configuration defines the configuration of the sensor (*e.g.*, the name of the probe to be installed on a monitoring system), while the property isPush defines whether the measured data points will be pushed by the sensor or have to be pulled by the SRL runtime system instead. On the other hand, a composite metric instance refers to other metric instances whose measurements are used to produce an aggregated value according to the metric formula of the metric of this instance. Thus, to perform the actual computations, the metrics associated with the formula are mapped to actual metric instances (which should be the composing instances of the composite metric instance) that do have measured values attached. This holds for the metrics used as input parameters, but also for the result of the computation.

In order to clarify the above mapping, consider that there is a CompositeMetricInstance CMI that has a formula f indirectly associated to it based on its metric M as well as a set of composing metric instances M_c directly associated to it. In order to perform the mapping, we have to have a one-to-one correspondence between the input metrics M_{fc} of f and the metrics of the instances in M_c. For better comprehension, let us extend on the example previously provided. Suppose that a specific metric instance CMI_1 is defined instantiating CM_1, then in CM_1's formula *Raw_M* metric should be equivalent to the metric of the respective component instance *RawI_M* of CMI_1.

10.2 Scheduling and Conditions

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

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. The property windowType, in turn, represents what happens when the window size is reached. SLIDING denotes that the window is slid by dropping superfluous elements, while FIXED denotes that the window is cleared.

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

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 respective 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.

A Condition represents an abstract condition that has a threshold value and a particular name to distinguish it from other conditions. 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 denotes for how long the condition will be valid. The concept of Condition enables expressing general requirements for cloud-based applications not tied to a particular concrete deployment model. It also enables the expression of generic requirements that might hold for a user irrespective of the applications.

A Condition can be classified as 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). Please consider here that we do not express constraints on metric instances but on their 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. In this way, we can indicate for example constraints on cost for the whole application or one or all of its components. Then, it is up to the platform to interpret these constraints 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, in contrast to the case of a metric, we

have created two references here which actually map to the case of evaluating cost: a monetary unit (*e.g.*, euros) and time interval unit (*e.g.*, seconds) reference to indicate that we assess cost per time and not in absolute terms. This modelling will of course be modified in case we need to define conditions on other properties apart from cost.

A condition, either pertaining to a metric or to a property, is mapped to a particular ConditionContext. The latter indicates the context under which the condition should hold. The context should clarify whether we need to enforce the condition on the whole application or a particular component or VM. It might also clarify for how many instances of the applications or its components the condition should be checked. Here we have catered for two different types of quantification: relative and absolute. Through the relative quantification, we indicate a minimum and maximum percentage of application or component instances on which the condition should hold. On the other hand, through the absolute quantification, we indicate the minimum and maximum amount of instances of an application or component on which the condition should be checked. At the modelling side, we have used four main modelling constructs to capture the two types of quantification: (a) quantifier: maps to a QuantifierType which indicates whether we should consider all instances, any or some in order to evaluate the condition – in case of some, then the rest of the constructs can be used to clarify the type and limits of quantification; (b) isRelative: indicates whether a relative or absolute quantification is concerned; (c) minQuantity: indicates the minimum relative or absolute value of instances; (d) maxQuantity: indicates the maximum relative or absolute value of instances. The modeller should be careful in providing correct min and max instance values in case the quantifier type is SOME. This means that, for absolute quantification, the max value should always be greater than or equal to the min 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 min one but also both should be in the range [0.0,1.0].

Figure 17 shows the portion of the class diagram of the metric package related to context.

Depending on which element is associated, *i.e.*, metric or property, a ConditionContext is further classified as a MetricContext and PropertyContext. A PropertyContext indicates the property to be measured and evaluated, while a MetricContext indicates the metric to be used in the evaluation of the condition. We should note here that we define the metric but then it is the responsibility of the measurement component of ExecutionWare to produce the instances of this metric and then, based on the produced values and the quantification information provided, evaluate the respective threshold in the condition according to the comparison operator specified. For instance, if the quantifier property is equal to ALL, then this

means that the measurement sub-system should create all instances of this metric in the respective execution context and then check that all values produced by these instances satisfy the condition posed. In order to assist in the generation of the metric instances, the MetricContext can indicate the schedule and window to be used for producing the measurements of the instances of the metric involved.

In case that the metric is composite, then we need to define the metric contexts for the components of the metric in order to further indicate the timing of the respective metric measurements. In this case, we have defined the CompositeMetricContext concept which includes a reference to the contexts of the composing metrics of the current metric. On the other had, in case that the metric is raw, we need to define the sensor to be used in order to produce the measurements of this metric. To this end, we have created the RawMetricContext concept which includes the respective reference to the sensor.

We should note here that it is not obligatory to define the composing contexts for all composite metric contexts. As it is considered that some information is inherited from the composite metric contexts to its composing contexts (so it can be absent), such a definition is obligatory only when such information should not be inherited but differentiated for a specific composing context. For instance, if we have the metric of response time which is calculated by the addition of the metrics of execution time and network latency. Then, in a context of the response time metric, the composing contexts of the composing metrics must not be specified as the timing/scheduling information is actually identical as well as the respective binding information to the respective application.

We must also note that when a composing context of a metric is not specified but its scheduling is more fined grained with respect to the metric of the enclosing context (as in the case of a raw metric versus a composite one), then this gives the freedom to the PaaSage platform to specify this scheduling information itself in an optimal way in order to reduce the load imposed in the monitoring sub-system (as constructed by the Executionware). However, if the user desires to have more control on how the measurement is performed, then the user must generate the composing contexts and the respective scheduling information.

As in the case of metric and metric instance, there can be some discrepancies in the modelling of metric contexts which include: (a) the case where a raw or composite metric context refers to a metric which is not raw or composite, respectively; (b) the case where a composite metric context contains composing metric contexts which refer to metrics which are not involved in the formula of the metric in this composite context.

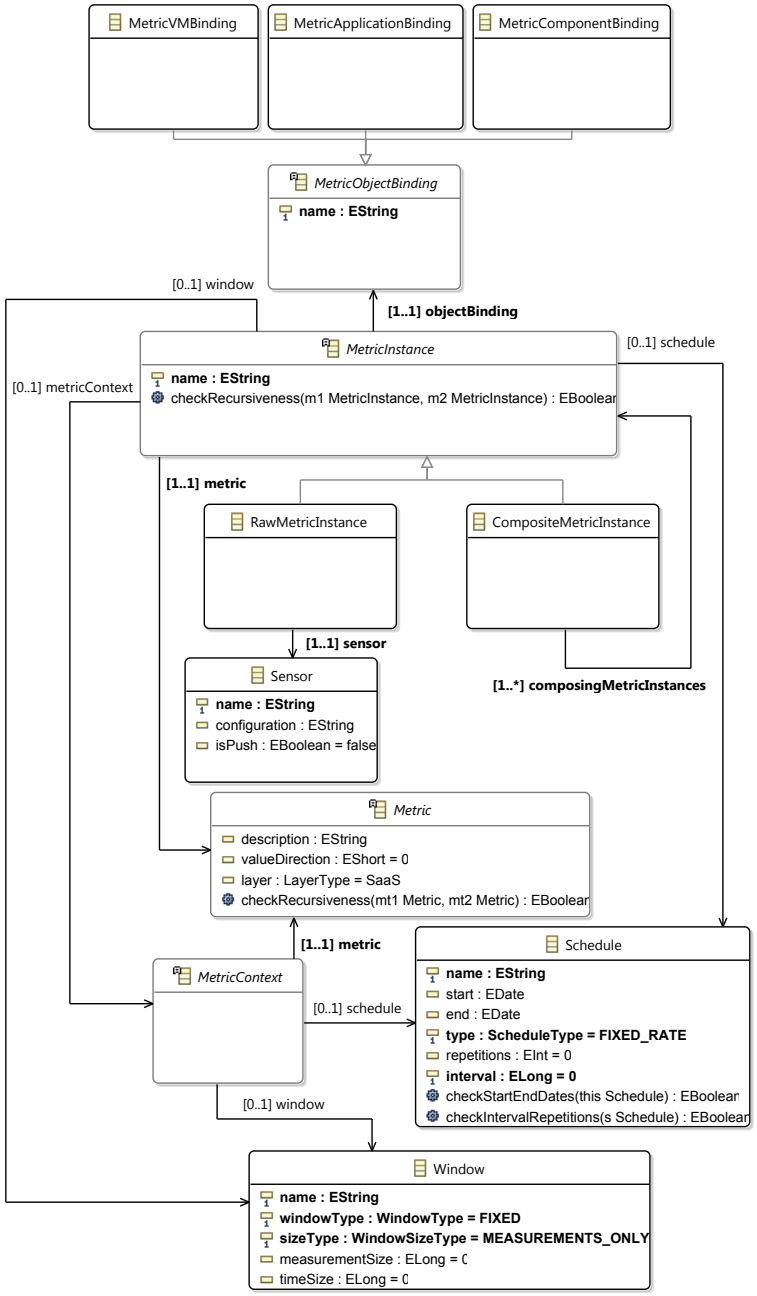

Figure 15: Portion of the class diagram of the metric package related to metric instances

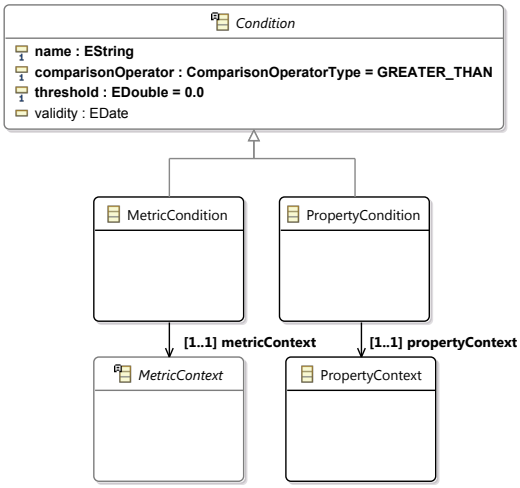

Figure 16: Portion of the class diagram of the metric package related to conditions

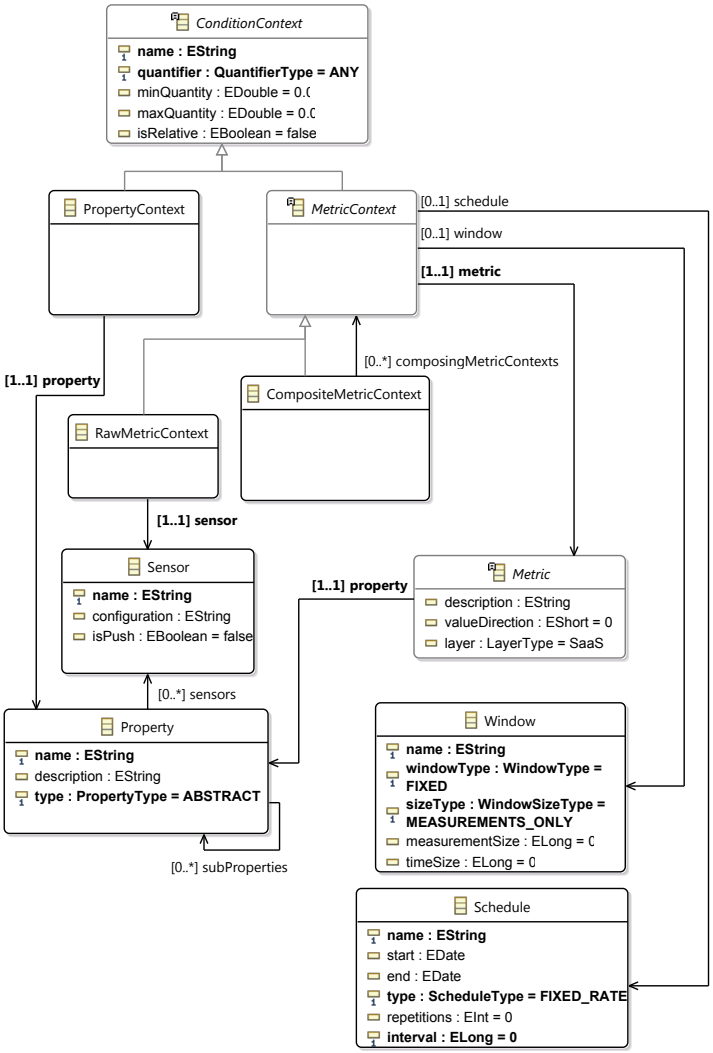

Figure 17: Portion of the class diagram of the metric package related to context

11 Scalability Rules

The scalability package of the CAMEL metamodel is also based on the Scalability Rules Language (SRL) [11, 3]. SRL is inspired by the Esper Processing Language (EPL)[8] with respect to the specification of event patterns with formulas including logic operators and timing. Basically, any scalability rule language has to encompass the definition of event patterns as well as scaling actions. The latter shall be executed whenever event patterns occur. Therefore, 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 multi-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). The ScalabilityRule refers to a set of ScaleRequirements that restrict how scaling actions are performed. Finally, the ScalabilityRule refers to Entities such as the user or the organisation (specified in the organisation package – *cf.* Section 13).

We should note that the user should be careful in the modelling of scalability rules in order not to provide conflicting scale requirements (*i.e.*, scale requirements of the same type which refer to the same internal component/VM) or scaling actions which conflict 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 higher and lower values in the range limit); (b) it is a VerticalScalingAction, the requirement is of the respective type VerticalScaleRequirement and the update on a particular VM characteristic is greater than the difference between the higher and lower values in the range limit dictated by the requirement for this VM characteristic.

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

11.2 Actions

Figure 18 shows the portion of the class diagram of the scalability package related to actions.

An Action can be classified as a ScalingAction or an EventCreationAction. The ScalingAction, in turn, can be classified as a HorizontalScalingAction or a VerticalScalingAction. The HorizontalScalingAction must refer 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 a virtual machine of the same type (hosting this component as indicated in the respective deployment model of the corresponding application). 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 must refer to a concrete VMInstance. The properties *Update define the amount of virtual resources (*e.g.*, computing cores, RAM, etc.) to be added to or to be removed from this concrete virtual machine.

We must note that the user should provide correct action types for the actions that the user generates. 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.

An EventCreationAction represents that the scaling actions are not sufficient to maintain the target QoS for a multi-cloud application. For instance, a multi-cloud application may still violate the target response time defined in an SLO despite the scale-out and scale-up actions performed.

11.3 Events

Figure 19 shows the portion of the class diagram of the scalability package related to events.

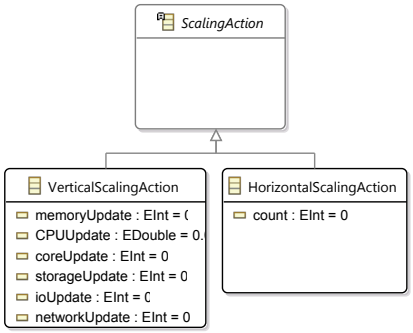

Figure 18: Portion of the class diagram of the scalability package related to actions

An Event can be classified as a SimpleEvent or an EventPattern. The SimpleEvent, in turn, can be classified as 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 QoS metric (*e.g.*, the response time of a component exceeds the target response time in an SLO). The NonFunctionalEvent refers to a MetricCondition that defines the threshold for the metric.

An EventPattern can be classified as a BinaryEventPattern or a UnaryEventPattern. The 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 20). 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 case that the latter operator is used, the user should define the lower occurrence or upper occurrence bounds or both (where in case of both, the upper bound should be greater or equal to the lower one).

Figure 20 shows the portion of the class diagram of the scalability package related to enumeration types.

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 and only this case, the user has to specify the occurrenceNum parameter value (which must be positive) to denote the number of event repetitions.WHEN defines that an event has to occur according to a particular time constraint defined by a Timer (which should be specified again only when this operator is provided).

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 (denoted by the value of the maxOccurrenceNum parameter that must be provided in this case). INTERVAL defines that an event has to occur after a particular amount of waiting time (which should be obviously positive).

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.

An EventInstance represents the actual (measurement) data associated with a particular event that occurred in the system (*e.g.*, the actual value measured, the component producing the event, etc.) 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 enables the evaluation/assessment of the events. The property layer represents the layer in the cloud stack where the event has occurred (*cf.* Section 10.1).

When modelling event instances, the user should be careful of correctly specifying and associating information. In particular, if the event instance layer is defined, then it needs to be the same with respect to the metric of the metric instance referenced by this event instance. In addition, the same metric should participate in the condition(s) of the instance's event.

11.4 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 been having an average CPU load beyond 50% for at least $5min$, and *(b)* concurrently at least one instance has been having an average CPU load beyond 80% for at least $1min$. 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. Thus, 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. In order to enable the deployment of the cluster, we complemented these two models with a deployment model conforming to the deployment package. In this model, the SimulationManager component is identified by SimulationManager, while the virtual machine on which its instances are deployed is identified by CPUIntensiveUbuntuGermany.

Listing 8 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 shows the metric model in textual syntax. The metrics mapping to the composite event structure in Listing 8 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 VM, once per `metric context`.

Note that the described rules show reasonable results only when applied to a SimulationManager cluster of up to five instances. Rules with a different content are required for larger clusters. Hence, reaching a size of five instances with a heavy load should trigger an EventCreationAction that would lead to changing the deployment model of the cluster.

Listing 8: Excerpt of SRL model showing a scalability rule with a composite event and the requirements

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

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

Listing 9: Excerpt of SRL model showing metrics and scheduling information

```
1   metric model ScalarmMetric {
2     window Win5Min {
3       window type: SLIDING
4       size type: TIME_ONLY
5       time size: 5
6       unit: ScalarmModel.ScalarmUnit.minutes
7     }
8
9     window Win1Min {
10      window type: SLIDING
11      size type: TIME_ONLY
12      time size: 1
13      unit: ScalarmModel.ScalarmUnit.minutes
14    }
15
16    schedule Schedule1Min {
17      type: FIXED_RATE
18      interval: 1
19      unit: ScalarmModel.ScalarmUnit.minutes
20    }
21
22    schedule Schedule1Sec {
23      type: FIXED_RATE
24      interval: 1
25      unit: ScalarmModel.ScalarmUnit.seconds
26    }
27
28    property CPUProperty {
29      type: MEASURABLE
30      sensors [ScalarmMetric.CPUSensor]
31    }
32
33    sensor CPUSensor {
34      configuration: 'de.uniulm.packagename.classname'
35      push
36    }
37
38    raw metric CPUMetric {
39      value direction: 0
40      layer: IaaS
41      property: ScalarmModel.ScalarmMetric.CPUProperty
42      unit: ScalarmModel.ScalarmUnit.CPUUnit
```

```
43      value type: ScalarmModel.ScalarmType.Range_0_100
44    }
45
46    composite metric CPUAverage {
47      description: "Average of the CPU"
48      value direction: 1
49      layer: PaaS
50      property: ScalarmModel.ScalarmMetric.CPUProperty
51      unit: ScalarmModel.ScalarmUnit.CPUUnit
52
53      metric formula Formula_Average {
54        function arity: UNARY
55        function pattern: MAP
56        MEAN( ScalarmModel.ScalarmMetric.CPUMetric )
57      }
58    }
59
60    raw metric context CPUMetricConditionContext {
61      metric: ScalarmModel.ScalarmMetric.CPUMetric
62      sensor: ScalarmMetric.CPUSensor
63      component: ScalarmModel.ScalarmDeployment.SimulationManager
64      quantifier: ANY
65    }
66
67    raw metric context CPURawMetricContext {
68      metric: ScalarmModel.ScalarmMetric.CPUMetric
69      sensor: ScalarmMetric.CPUSensor
70      component: ScalarmModel.ScalarmDeployment.SimulationManager
71      schedule: ScalarmModel.ScalarmMetric.Schedule1Sec
72      quantifier: ALL
73    }
74
75    composite metric context CPUAvgMetricContextAll {
76      metric: ScalarmModel.ScalarmMetric.CPUAverage
77      component: ScalarmModel.ScalarmDeployment.SimulationManager
78      window: ScalarmModel.ScalarmMetric.Win5Min
79      schedule: ScalarmModel.ScalarmMetric.Schedule1Min
80      composing metric contexts [ScalarmModel.ScalarmMetric.
        CPURawMetricContext]
81      quantifier: ALL
82    }
83
84    composite metric context CPUAvgMetricContextAny {
85      metric: ScalarmModel.ScalarmMetric.CPUAverage
86      component: ScalarmModel.ScalarmDeployment.SimulationManager
87      window: ScalarmModel.ScalarmMetric.Win1Min
88      schedule: ScalarmModel.ScalarmMetric.Schedule1Min
89      composing metric contexts [ScalarmModel.ScalarmMetric.
        CPURawMetricContext]
90      quantifier: ANY
91    }
92
93    metric condition CPUMetricCondition {
94      context: ScalarmModel.ScalarmMetric.CPUMetricConditionContext
95      threshold: 80.0
96      comparison operator: >
97    }
98    metric condition CPUAvgMetricConditionAll {
99      context: ScalarmModel.ScalarmMetric.CPUAvgMetricContextAll
```

```
100       threshold: 50.0
101       comparison operator: >
102     }
103
104     metric condition CPUAvgMetricConditionAny {
105       context: ScalarmModel.ScalarmMetric.CPUAvgMetricContextAny
106       threshold: 80.0
107       comparison operator: >
108     }
109   }
```

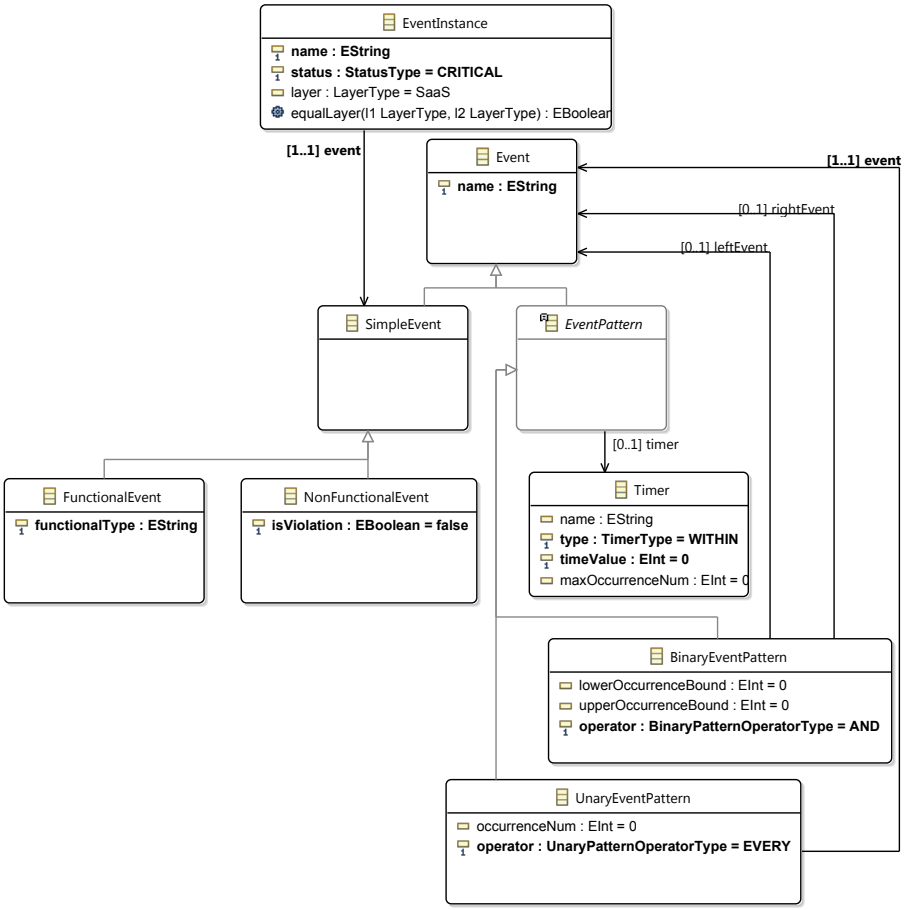

Figure 19: Portion of the class diagram of the scalability package related to events

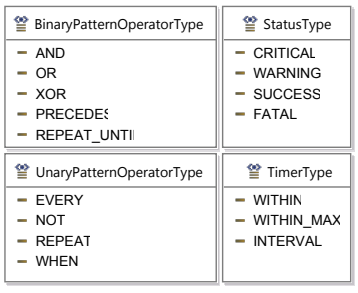

Figure 20: Portion of the class diagram of the scalability package related to enumeration types

12 Providers

The `provider` package of the CAMEL metamodel is based on Saloon [22, 23, 24]. Saloon consists of a DSL along with a framework for specifying application requirements and user goals of multi-cloud applications and selecting compatible cloud providers by leveraging upon feature models [2] and ontologies [7]. In the following, we describe and exemplify the main concepts in the provider package.

Figure 21 shows the class diagram of the provider package.

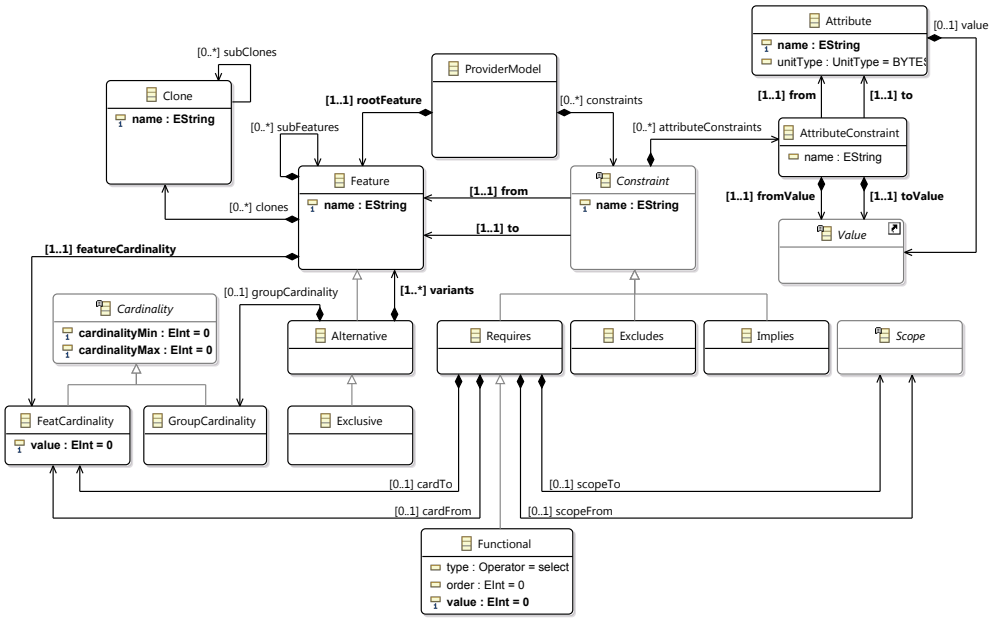

Figure 21: Class diagram of the provider package

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 max can also take the value of -1 to indicate 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 subfeatures and be specialised to 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 denotes 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.

A Constraint represents a typical restriction in binary feature models [10]. 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 denotes 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 to 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 denotes 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 to a Attribute Constraint, enabling the specification of restrictions of the form (! this has been modified in the current version of the deployment package – attribute constraint is not subclass of requires constraint):

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

This constraint denotes that if the attribute c of A has X as value, then the attribute d of B needs Y as value.

Here the user needs to be careful according to the following two aspects: (a) the attributes should be different and (b) the value provided for the attributes should be included in the attributes' value type.

12.1 Example

Listing 10 shows an excerpt of the provider model for GWDG[9] specified using the CAMEL textual syntax.

[9] `http://www.gwdg.de/index.php`

Listing 10: AmazonEC2 model (Excerpt)

```
provider model GWDGProvider {
  constraints {
...
    implies M1_LARGE_VM_Constraint_Mapping {
      from: ScalarmModel.GWDGProvider.GWDG.VM
      to: ScalarmModel.GWDGProvider.GWDG.VM
      attribute constraints {
        attribute constraint {
          from: ScalarmModel.GWDGProvider.GWDG.VM.VMType
          to: ScalarmModel.GWDGProvider.GWDG.VM.VMMemory
          from value: string value 'M1.LARGE'
          to value: int value 8192
        }
        attribute constraint {
          from: ScalarmModel.GWDGProvider.GWDG.VM.VMType
          to: ScalarmModel.GWDGProvider.GWDG.VM.VMCores
          from value: string value 'M1.LARGE'
          to value: int value 4
        }
        attribute constraint {
          from: ScalarmModel.GWDGProvider.GWDG.VM.VMType
          to: ScalarmModel.GWDGProvider.GWDG.VM.VMStorage
          from value: string value 'M1.LARGE'
          to value: int value 80
        }
      }
    }
...
  }
  root feature GWDG {
    attributes {
      attribute DeploymentModel {
        value: string value 'Private'
        value type: ScalarmModel.GWDGType.StringValueType
      }
      attribute ServiceModel {
        value: string value 'IaaS'
        value type: ScalarmModel.GWDGType.StringValueType
      }
      attribute Availability {
        unit type: PERCENTAGE
        value: string value '95'
        value type: ScalarmModel.GWDGType.StringValueType
      }
      attribute Driver {
        value: string value 'openstack-nova'
        value type: ScalarmModel.GWDGType.StringValueType
      }
      attribute EndPoint {
        value: string value 'https://api.cloud.gwdg.de:5000/v2.0/'
        value type: ScalarmModel.GWDGType.StringValueType
      }
    }
    sub-features {
      feature VM {
        attributes {
          attribute VMType {
            value type: ScalarmModel.GWDGType.VMTypeEnum
```

```
59            }
60            attribute VMOS {
61              value type: ScalarmModel.GWDGType.VMOsEnum
62            }
63            attribute VMMemory {
64              unit type: MEGABYTES value type: ScalarmModel.GWDGType.
      MemoryList
65            }
66            attribute VMStorage {
67              unit type: GIGABYTES value type: ScalarmModel.GWDGType.
      StorageList
68            }
69            attribute VMCores {
70              value type: ScalarmModel.GWDGType.CoresList
71            }
72          }
73          feature cardinality {
74            cardinality: 1 .. 8
75          }
76        }
77        feature Location {
78          sub-features {
79            feature Germany {
80              feature cardinality {
81                cardinality: 1 .. 1
82              }
83            }
84          }
85          feature cardinality {
86            cardinality: 1 .. 1
87          }
88        }
89      }
90      feature cardinality {
91        cardinality: 1 .. 1
92      }
93    }
94  }
95
96  type model GWDGType {
97    enumeration VMTypeEnum {
98      values [ 'M1.MICRO' : 0,
99  ...
100     'M1.LARGE' : 4,
101 ...
102     'C1.XXLARGE' : 15 ]
103   }
104   enumeration VMOsEnum {
105     values [ 'Fedora 20 server x86_64' : 0,
106     'Ubuntu 14.04 LTS Server x86_64' : 1,
107 ...
108     ]
109   }
110   range MemoryRange {
111     primitive type: IntType
112     lower limit {
113       int value 256 included
114     }
115     upper limit {
```

```
      int value 32768 included
    }
  }
  range StorageRange {
    primitive type: IntType
    lower limit {
      int value 0 included
    }
    upper limit {
      int value 160 included
    }
  }
  range CoresRange {
    primitive type: IntType
    lower limit {
      int value 1 included
    }
    upper limit {
      int value 16 included
    }
  }
  string value type StringValueType {
    primitive type: StringType
  }
  list StorageList {
    values [ int value 0,
    int value 20,
    int value 40,
    int value 80,
    int value 160 ]
  }
  list MemoryList {
    values [ int value 256,
    int value 512,
    int value 2048,
    int value 4096,
    int value 8192,
    int value 16384,
    int value 32768 ]
  }
  list CoresList {
    values [ int value 1,
    int value 2,
    int value 4,
    int value 8,
    int value 16 ]
  }
}
```

13 Organisations

The organisation package of the CAMEL metamodel is based on the organisation subset of CERIF [8]. CERIF is a modelling framework for specifying organisations, users and other entities in the research domain. It is an EU recommendation[10] for information systems related to research databases used for standardising research information and fostering research information exchange. In the following, we describe the most important classes and corresponding properties and references in the organisation package as well as sample models conforming to this package.

The class diagram (apart from the root class which complicates the respective figure) of the organisation package is shown in Figure 22. The root class of the organisation package is OrganisationModel which represents the actual CERIF model of an organisation. This class encapsulates all the appropriate building blocks for defining a complete CERIF model for an organisation through respective containment references, such as the list of data centres offered by the organisation (see dataCentres reference), the organisation itself (see organisation reference), its users (see users reference) and user groups (see userGroups reference) as well as the permissions (see permission reference) and role assignments (see roleAssignments reference) issued by the organisation.

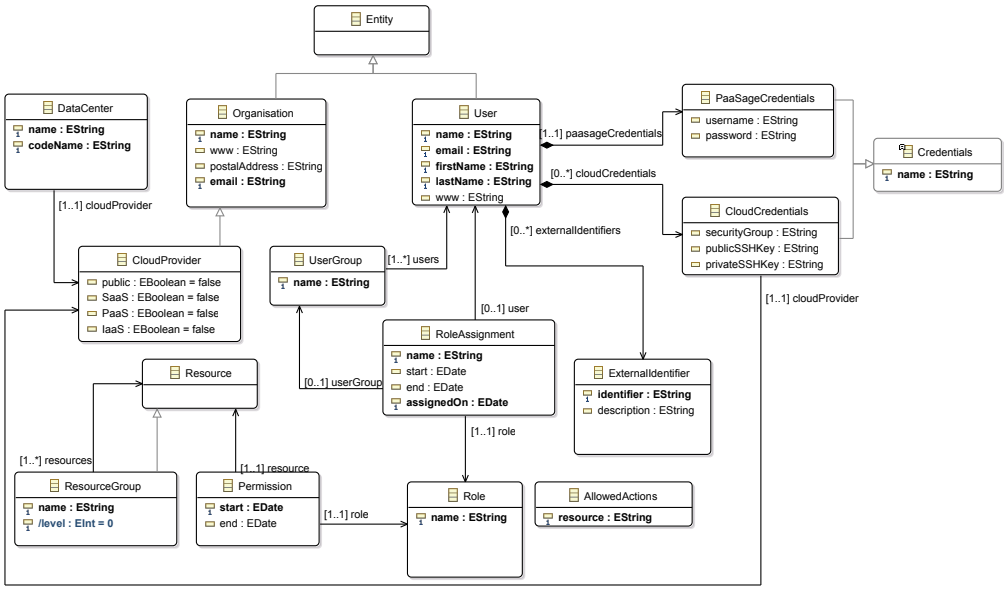

Figure 22: Class diagram of the organisation package

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

The Entity class represents any entity which could be used to express requirements and exploit the PaaSage platform. It is distinguished into organisations and users. An organisation, represented by the Organisation class, is mapped to particular information that identifies it, such as its name, web and postal address, and its email. Such an organisation can exploit the PaaSage platform in order to deploy applications and adapt them at run time. It can also constitute a cloud provider. In this case, the CloudProvider class has been modelled as a subclass of Organisation which provides additional cloud provider information for an organisation. This cloud provider information maps to whether the cloud offered is public (see public boolean property), the types of cloud services offered (see PaaS, IaaS and SaaS boolean properties which can be set independently to indicate whether the specific type of service is offered or not by the cloud provider), the security capabilities offered (see SecurityCapability class in security package) and the description of the provider's service offerings in terms of a provider model (see providerModel reference).

A cloud provider typically constructs one or more data centres which host the resources and cloud services offered. These data centres are represented by the DataCenter class which is identified by its name and code type and refers to the cloud provider owning it and to a respective location (see Location class in location package – Section 9) in which it is situated (*e.g.*, country). The modelling of data centres was included in the organisation package with the rationale that we have to differentiate between the same services offered from the same cloud provider in different data centres as there is usually a differentiation in cost and performance for these service offerings. The modeller should be careful not to create two data centres for the same cloud provider with the same name and codeName.

A user is another type of an entity which can function on its own or on behalf of one or more organisations. It is represented by the User class which in the latter case refers to the organisations that the user represents or works for. A user is identified by its first and last name, its email, its credentials (see Credential concept), and the URL of its personal web site (see www property). It is also associated to a set of requirements that it has set to the platform (see requirements reference), to a set of deployment models for one or more applications (see deploymentModels reference) that it has specified, and to a set of external identifiers that can be used to identify the user (see reference externalIdentifiers) for the purposes of the platform. An external identifier is characterised by its ID (as a String) along with a name and a textual description. In term of credentials, a user refers to specific credentials (see paasageCredentials reference) that are used to authenticate the user in the PaaSage platform, which are of type PaaSageCredentials, thus allowing the user to exploit the facilities offered by the platform and to a set of credentials of type CloudCredentials (see cloudCredentials references) that

can be used for authenticating the user against a cloud provider and thus enabling the platform to perform cloud-specific tasks for the user. Any credential is represented by the Credentials class which is characterised by its respective name (placeholder for an identifier value). To accommodate for the two different types of credentials, two subclasses of Credentials have been specified, namely PaaSageCredentials and CloudCredentials. PaaSage credentials are characterised by the respective userName and password used to authenticate the user, while cloud credentials involve the specification of the following information: (a) the reference to the respective cloud provider on which these credentials should be used, (b) the public SSH key and (c) the private SSH key.

An organisation, apart from specifying its users, can also specify groups of users (see UserGroup class) which are more manageable and can be used for specifying permissions in a more compact way. Such groups are identified by the name and associated to the list of users that they represent. In this way, as a step towards defining permissions, after all organisation users and user groups have been defined, these users and groups can be assigned a particular role (*e.g.*, administrator) which can then be granted permissions for performing particular tasks. A role is represented by the Role class with just a String-based name property while the assignment of roles to users or user groups is represented by the RoleAssignment class. The latter class refers to the user or user group to which the role is assigned, to the organisation that performs the assignment, to the role actually assigned and is also identified by three particular dates: (a) the date of the assignment (see assignedOn property), (b) the start date from which the assignment holds and (c) the end date after which this assignment will be invalidated.

On the other hand, permissions map a role to a set of actions allowed to be performed by it (and thus to the users or groups mapping to this role). Permissions are represented by the Permission class which refers to the granted role, to the set of actions allowed (see reference actions mapping to Action class of CAMEL), the resource on which the actions will be performed and to the organisation that issues it while it is identified by two dates indicating the start and end date of the permission. For instance, a particular permission could indicate that from now until the next year, the administrator role is granted to perform deployments of VMs on the private cloud of the organisation.

In a similar fashion, the AllowedActions class is used to indicate all possible actions that can be performed on a particular resource class (*e.g.*, all possible actions, such as deploy or destroy for any instance of the VM concept). In this way, when a particular permission is specified in an organisation model, then it is checked if it is correct in terms of the actions identified and those allowed for the class on which the permission resource belongs.

Any resource (*e.g.*, application) is represented by the Resource class, while resources can also be grouped into ResourceGroups such that we can specify permissions in a more compact way for not just one but many resources. To this end, to have a consistent modelling, a ResourceGroup has been made subclass of Resource. A resource group is identified by its name and level (especially if we regard that a tree or a forest of trees is created when resource groups are modelled) and refers to the resources it contains.

13.1 Example

Continuing the example of the textual specification of the Scalarm CAMEL model, we now analyse the case of the encompassing organisation model. This model is textually defined according to the content of the Listing 11. As it can be seen, this model defines one organisation (AGH) and one user (Michal Orzechowski). While basic information is defined for the organisation, the user specification includes the definition of the user's credentials in the PaaSage platform in the form of a `username` and `password`. Note that both users and credentials have identifiers (which are ID-based values given to the `name` property) so the respective header textual specification should not contain any arbitrary string (that is why we have defined the token `MichalOrzechowski` instead of just providing the first and last name of this user – as this token maps to an identifier, we will then be able to distinguish between users with the same first and last name).

Listing 11: The enclosing organisation model of the Scalarm model in textual syntax

```
1  organisation model AGHOrganisation {
2    organisation AGH {
3      www: 'http://www.agh.edu.pl/en/'
4      postal address: 'al. Mickiewicza 30, 30-059 Krakow, Poland'
5      email: 'morzech@agh.edu.pl'
6    }
7
8    user MichalOrzechowski {
9      first name: Michal
10     last name: Orzechowski
11     email: 'morzech@agh.edu.pl'
12
13     paasage credentials MorzechCredentials {
14       username: morzech
15       password: 'morzech_at_agh_dot_edu_dot_pl'
16     }
17   }
18 }
```

14 Security

In the context of the initial work in WP4, the MDDB schema generated, which is documented in D4.1.1 [13], included a part which has been devoted in the description of security requirements and capabilities. This part actually maps to a new package which is devoted to describing security requirements and capabilities in terms of security controls and linking these controls to the security properties that need to be monitored and assessed. In this way, this package complements requirement, metric and scalability packages as it is able to describe security aspects that are well integrated with these packages in order to be able to completely capture the security domain in the context of cloud computing. The benefits of this new package is that: (a) it can be used for matching providers based on end-user security requirements thus enabling an additional filtering of the cloud provider space apart from the functional filtering based on the respective cloud service capabilities as well as paves the way for negotiation; (b) it is well integrated with SLOs and as such it can connect security controls to particular security levels that need to be sustained by a cloud provider based on a particular requirement model defined via the requirement package. In this way, it can be used in user-provider negotiations to establish either security or complete (functional, non-functional and security) SLOs; (c) it is well integrated with the metric and scalability models and as such it enables the monitoring of the specified security levels which are incarnated in a set of conditions on security metrics and the development of adaptation rules which can be used for remedying any breach of the specified security levels.

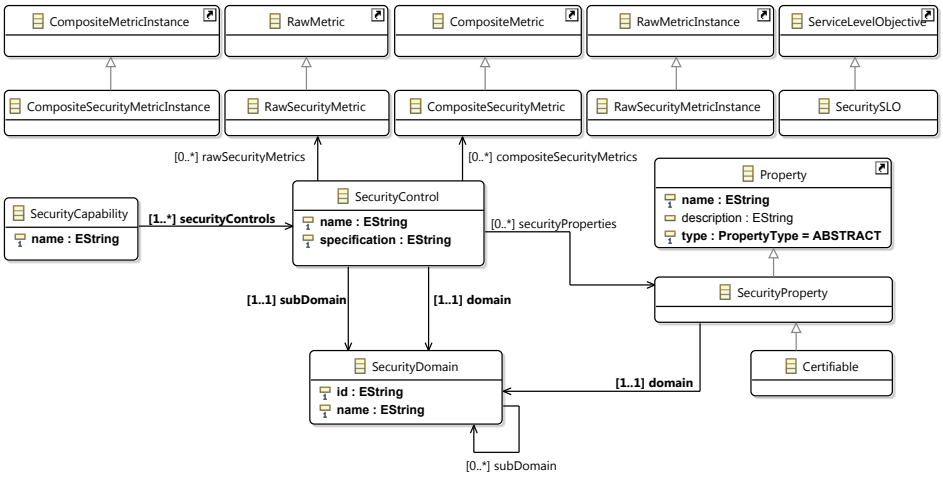

Figure 23: Class diagram of the security package

Figure 23 graphically depicts the class diagram of the security package. The security package comprises a central class called `SecurityModel` which can be

used to describe security controls and capabilities as well as other basic constructs, such as security properties and metrics that can be used in (security) SLOs. In this way, a SecurityModel refers to all possible security constructs that can be defined by a user/modeller.

The SecurityControl is another important class which represents security controls. Each security control is identified by a name, a particular domain and subDomain (of type SecurityDomain), a textual description and to a set of security properties and metrics that it links to. It should be noted here that one of the goals in the PaaSage project has been to create a digital form of the specification of all possible security controls as they have been identified by Cloud Security Alliance (SCA) and store all these specifications in MDDB. In this way, when security requirements will need to be defined, the end-user will have the opportunity to select the security controls that better satisfy the user's needs by either browsing the respective security control list or making focused searches. In the latter case, it would have been interesting to support such searching functionality by exploiting all the information that has been modelled for security controls. This has already been supported as the specification and storage of a particular security model which encloses both all the SCA security controls as well as common metrics and properties has been performed.

A SecurityDomain represents a specific security domain which is identified by a particular id and name. It also includes a set of sub-domains. For instance, an example of a security domain would be "Application & Interface Security" with "AIS" as an id and "Data Security/Integrity" as its sub-domain. This concept is particularly important and useful as it enables us to group security controls, properties and metrics in logical and meaningful categories which can represent a coarse-grain level of consideration for the user. As such, the user can first choose the particular security domain on which the user can specify particular elements or constraints and then go to more fine-grained levels by, *e.g.*, creating security SLOs which must involve only security metrics or properties of this domain. We should note here that an SecuritySLO is just a sub-class of ServiceLevelObjective, thus relying on the capabilities of the metric and requirement packages in order to define all the information required but referring to conditions which refer only to security metrics or properties.

A SecurityProperty, a sub-class of Property in SRL, represents a security property which can be either abstract or certifiable and has a particular domain. Abstract security properties are not directly measurable but rather represent a more generic type of property which can then be concretised through other particular but measurable properties. We should mention that this characteristic is not only attributed to security properties but any type of property (see also documentation on metric package and especially the Property class). On the other hand, certifiable properties are measurable and are thus associated to a particular security

metric which can be defined via the metric package. As already indicated above and can be easily understood by the description, security properties represent the linking point between the security models, SLOs and scalability models as they can be used to map security controls to particular security levels which are specified in a SLA and are monitored and assessed based on the respective metric and condition information that is captured in SRL. To distinguish between security and different type of metrics, we have also modelled the RawSecurityMetric & CompositeSecurityMetric classes that are just subclasses of the corresponding RawMetric and CompositeMetric classes of the metric package. In this way, we just make the distinction and rely on the modelling information attributed to these Metric classes to fully specify security metrics. We should highlight that actually the main distinction at the modelling side between security properties and metrics with respect to normal properties and metrics (as defined in the metric package) is the reference to the associated security domain.

Similarly to the case of the two types of security metrics, two types of security metric instances have been created, namely RawSecurityMetricInstance and CompositeSecurityMetricInstance, which are sub-classes of the respective metric instance types defined in the metric package (*i.e.*, the RawMetricInstance and CompositeMetricInstance classes, respectively), thus inheriting all appropriate information to cover the definition of a security metric instance.

Finally, to enable the matching of security requirements, one or more security capabilities can be defined for a cloud provider in the respective organisation model to indicate the security controls that this provider supports. Each security capability can map to security control realizations in a particular data centre. Thus, it is expected that there can be a differentiation in security capabilities exhibited in a specific cloud depending on the respective data centres involved (otherwise, only one security capability can be specified which holds for the whole cloud). Then, matching can actually be realised through checking whether the user's security controls, encompassed in the form of a specific security requirement, are included in those supported by the cloud provider, encompassed in the form of a specific security capability. Such type of matching can be performed at the global or local VM level. At the global level, this will indicate whether all VMs required in a deployment model map only to those cloud service offerings for which the cloud providers exhibit security capabilities that match the respective security requirements posed by the user for the corresponding application. At the local level, this will indicate that a required VM maps to only those cloud service offerings for which the cloud providers exhibit security capabilities that match the respective security requirements posed by the user for the corresponding (application) component to be hosted on this VM.

14.1 Example

While the focus of the Scalarm use case is not at the current moment on security aspects, we now provide a specific example which showcases how security information can be specified and exploited by the PaaSage platform and be used in conjunction to the current textual Scalarm model which is analysed in separate sections of this report. This example is textually defined through the content of the Listing 12. As it can be seen, a CAMEL model has been defined which comprises a security, a requirement and an organisation model. The security model specifies a set of security controls and respective security properties attributed to these controls, the security domains of these controls and properties as well as a security capability defined as the conjunction of all the security controls specified. The set of security controls defined is also used for the specification of a security requirement which could then be used for the matching of security capabilities and requirements. As the example content indicates, the security capability could be exhibited by the Amazon cloud provider while the security requirement could be part of the requirement model of the Scalarm use case. In this sense, as this security requirement does not specify any particular component or VM, it is a generic requirement that should hold for all the deployments of the Scalarm application. Thus, it is always used to match any cloud provider which could satisfy other types of requirements which can be specific to Scalarm components or VMs.

By focusing on the main content of the enclosing security model, we can now identify the main constructs used for each object/instance that is defined. Starting with security domains, we can see that they are identified by a particular ID given in the header of the respective textual specification, while their main body includes the definition of the domain's name and a reference to its sub-domains. For instance, the security domain identified by `IAM` has as its name "Identity & Access Management" and as its sub-domains a reference to the other two domains specified.

Security properties are first identified by their name. Then, their main body include a textual description of what do they represent, their type and their domain. The sole property specification in the example concerns the security property of "identity assurance" which has a particular textual description, is `ABSTRACT` and belongs to the domain of "Identity & Access Management" (`IAM`).

Security controls are initially identified by their name. Their body specification includes the definition of their specification, their main and sub-domain and the properties and metrics that they include (which can be used to go from higher-level to low-level requirements such that evaluation of security levels is enhanced). For instance, the first specification in the example defines the security control `IAM_02` which has a quite long specification (such a specification

can be automatically produced from external files defining standardised security control models), the IAM domain as its main domain and the IAM_CLPM domain as its sub-domain and includes the property of "identity assurance".

Finally, a security capability is identified by a particular name and contains references to one or more security controls. In the particular capability defined in the model, the name specified is SecCap while all security controls that have been defined are referenced.

Listing 12: A set of security-related models which could be used to extend the Scalarm user case model

```
unit model ScalarmUnit {
  time interval unit {sec:SECONDS}
}

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

  domain IAM_CLPM {
    name: "Credential Life cycle / Provision Management"
  }

  domain IAM_UAR {
    name: "User Access Revocation"
  }

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

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

```
45
46    security control IAM_11 {
47      specification: "Timely de-provisioning (revocation or
48        modification) of user access to data and
49        organizationally-owned or managed (physical and virtual)
50        applications, infrastructure systems, and network
51        components, shall be implemented as per established
52        policies and procedures and based on user's change in
53        status (\eg, termination of employment or other
54        business relationship, job change or transfer). Upon
55        request, provider shall inform customer (tenant) of
56        these changes, especially if customer (tenant) data
57        is used as part the service and/or customer (tenant) has
58        some shared responsibility over implementation of control."
59      domain: ScalarmSecurity.IAM
60      sub-domain: ScalarmSecurity.IAM_UAR
61      security properties [ScalarmModel.ScalarmSecurity.
        IdentityAssurance]
62    }
63
64    security capability SecCap {
65      controls [ ScalarmSecurity.IAM_02, ScalarmSecurity.IAM_11]
66    }
67  }
68
69  requirement model ScalarmExtendedReqModel {
70    security requirement AllIAMsSupported{
71      controls [ScalarmSecurity.IAM_02, ScalarmSecurity.IAM_11]
72    }
73  }
74
75  organisation model AmazonExt {
76    provider Amazon {
77      www: "www.amazon.com"
78      email: "contact@amazon.com"
79      SaaS
80      PaaS
81      IaaS
82      security capability [ScalarmModel.ScalarmSecurity.SecCap]
83    }
84  }
```

15 Execution

The purpose of the execution package, which has relied on the respective MDDB schema part of the work conducted in WP4 (see Deliverable D4.1.1[13]), is to able to capture all execution historical information for one or more deployment models of an application, including the measurements produced and the SLO assessments performed. As such, this information could be exploited in order to guide the user in selecting the best possible application deployment according to its requirements as well as appropriately characterising the profile of an application through analysing this information. The execution package has been designed in such a way that: (a) there is traceability between the user requirements for a particular application and the deployment model produced from them according to a reasoning process as well as the measurement information generated when executing the application according to this deployment model. In this way, when a particular SLO violation occurs, we will be able to know which user requirement was violated for which application and according to which deployment model; (b) not only measurement but also adaptation history information is modelled in order to enable the user and the PaaSage platform components to know which concrete adaptation actions were performed for which rule according to which measurement/SLO violation. Such information could provide insight on some common scalability rule patterns which could be identified in order to be adopted later on for end-user applications and also on some problematic situations which should be avoided in the future by allowing the expert to either not select a specific deployment model or specify particular scalability rules that prevent this problematic situation from happening; (c) it is well-integrated with other packages and especially with the metric one as it maps to metric information related to measurements.

A main model class (see class diagram for execution package in Figure 24), called ExecutionModel, has been defined which is considered as the root of an execution model under which all respective building blocks are specified. One of the most important building block classes is ExecutionContext which captures the whole execution context for a particular application deployment model. This class is characterised by its name (used as a unique identifier), start and end time (denoting the time when the corresponding application execution was initiated and finished) and the total deployment/execution cost (obviously calculated when the application execution has been finished and specified along with the respective cost unit) and refers to the application, to the respective application deployment model and to a set of requirements (see requirementGroup reference) given by a user for the particular application (which have lead to the selection of the respective deployment model).

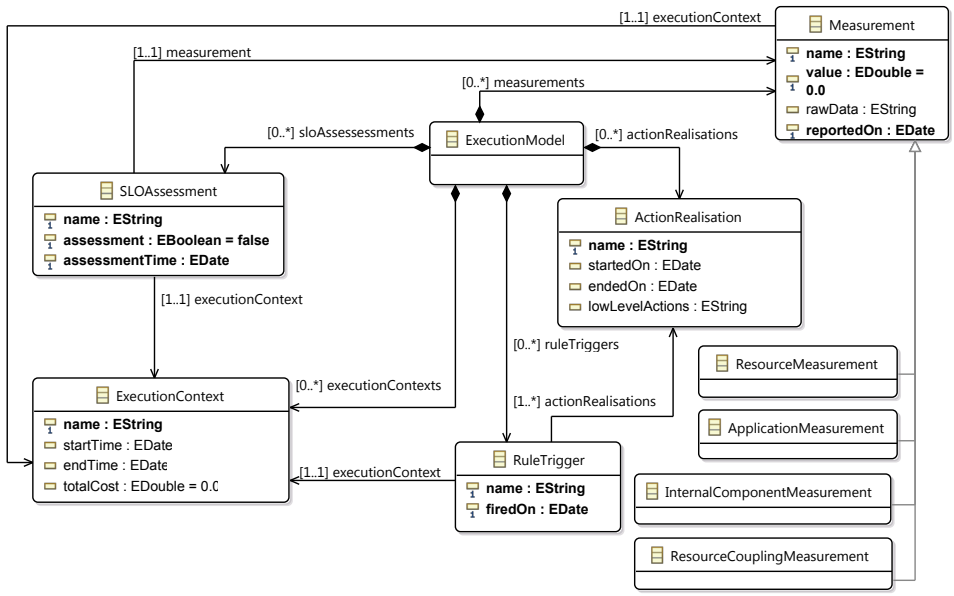

Figure 24: Class diagram of the execution package

When a particular application deployed according to a specific model is executed, measurements are produced which are included in the respective execution model and are represented by the Measurement class. These measurements are associated to a specific metric and execution context and are identified by a particular name. They are also associated to the value that is measured/reported, the reporting date for the value as well as the raw data (stored in TSDB) used to derive this value, the event instance triggered based on the measured value and the respective ServiceLevelObjective (if a particular user SLO was provided for the corresponding metric). As such, through all of these measurement information it can be known what SLO/user requirement needs to assessed according to the respective value measured and which event is triggered in case of SLO/metric condition violation. As such, it can be assessed whether any corresponding scalability rule should be triggered in order to perform the respective adaptation action.

It is the responsibility of the user to provide correct references in the context of a specific measurement. This spans the following checks: (a) if an SLO is referenced, then it should indirectly map to a metric whose instance is the metric instance referenced by the measurement; (b) if an event instance is referenced, then the metric instance associated to this event instance should be identical to the one referenced by the measurement; (c) if the metric instance maps to a metric with a specific value type, then the measurement value should belong to

this value type; (d) the metric instance has a binding which refers to the same execution context as the one associated to the measurement.

Different types of measurement have been identified according to the different types of bindings for metrics involved in these measurements. For resource-based metrics, the ResourceMeasurement class has been modelled which additionally carries the information about which resource is being measured (*i.e.*, a VM instance). In this way, measurements about *e.g.*, the CPU utilisation of a VM can be modelled. For network/communication-based metrics, the ResourceCouplingMeasurement class has been modelled which refers to the respective source and destination VM instances. For component-based metrics, the InternalComponentMeasurement has been modelled which refers to the internal component (instance) being measured. Finally, for application-based metrics, the ApplicationMeasurement class has been modelled which refers to the application being measured.

When particular measurement values are produced for metrics for which an SLO has been specified, an SLOAssessment takes place in order to evaluate whether the respective metric condition in the SLO is violated or not. To this end, an SLOAssessment refers to the measurement and the execution context under which this measurement took place and as well as to the SLO concerned. It also indicates when the assessment took place and whether it was successful or not (see assessment boolean property) (so obviously a value of true means that the SLO was not violated and a value of false that it was violated).

Measurements can lead to the generation of event instances (see respective scalability package class) which in turn can lead to the triggering of scalability rules which relies on the occurrence of these events. In this respect and to be also able to capture the adaptation history for an application, the RuleTrigger class has been modelled which refers to the event instances that triggered it, the execution context under which this triggering took place, the scalability rule triggered and the actual adaptation actions performed. Apart from this, a rule trigger also specifies when it was triggered. The actual adaptation actions performed for a rule trigger are represented by the class ActionRealisation as they constitute realisations of the actions that must be performed when the rule is triggered and that are part of the scalability rule definition. To this end, an action realisation refers to the action that it realizes and specifies three types of information: (a) the low level actions performed that are specific to a cloud provider and as a whole constitute the respective action realisation, (b) the date when the action realisation started to execute and (c) the date when the action realisation stopped execution. Obviously, the first action realisation initiated by a rule triggering should occur at the same time or after the rule triggering. The latter three types of information are quite important as: (a) the low level actions show how a particular action is realised in the context of a cloud provider and also show whether an action

realisation has been modified in the course of time, (b) the date information can be used to infer when a particular rule trigger finished execution by considering the end time of the last action realisation of the rule trigger.

15.1 Example

By relying on the Scalarm CAMEL textual model and assuming that this model is used as input by the PaaSage platform in order to discover the respective deployment plan and execute it, we now showcase the concrete camel model that could be generated by the platform from the deployment of the Scalarm application. This CAMEL model actually contains two main sub-models: (a) a metric model which specifies particular metric instances that have been generated to produce the respective measurements and (b) an execution model which comprises an execution context mapping to the current Scalarm deployment as well as one measurement and a corresponding SLOAssessment.

Turning now to the textual syntax, it can first be highlighted the ability to import a concrete model that has already been defined. Through such ability, the user can re-use objects that have already been specified and make appropriate references to them. In the case of the Scalarm CAMEL model imported, there is a re-use of one metric, one vm instance (please assume that it is added after the concrete deployment plan is produced by the Reasoner), one metric condition, one unit, one application, one `ServiceLevelObjective`, one requirement group and one deployment model definition.

Concerning the definition of metric instances, we can see that the respective metric model first defines a `VMMetricBinding` in order to set up the context for the definition of the respective metric instance by indicating what is the current execution context and what is the instance of the VM concerned. Next, is the definition of the metric instance itself which comprises a reference to the metric and sensors involved and of course to the previously defined binding.

The definition of the execution model starts with the specification of an execution context which comprises an indication of the total cost along with the reference to the respective monetary unit, a reference to the application that is being measured, a reference to the deployment model of the application which has been executed and a reference to a set of requirements incarnated in the form of a requirement group which have lead to the production of the concrete deployment model and must be satisfied. Next, we can see the specification of one resource measurement produced by the respective metric instance previously defined which comprises the value measured, the date of its production, and the respective execution context involved. As a measurement concerns a metric used in an SLO, an SLOAssessment is also specified which indicates apart from again providing the appropriate references (to the SLO and the measurement causing

the assessment) that the corresponding SLO has been violated and the actual date of the assessment.

Listing 13: A metric and an execution model which could be generated when the Scalarm use case model is deployed

```
import "Scalarm.camel"
camel model ScalarmCamelExecModel{
  metric model ScalarmMetricInstanceModel{
    vm binding ScalarmVMBinding{
      execution context: ScalarmExecModel.EC1
      vm instance: ScalarmModel.ScalarmDeployment.
    StorageIntensiveUbuntuGermanyInst
    }

    raw metric instance RawCPUMetricInstance{
      metric: ScalarmModel.ScalarmMetric.CPUMetric
      sensor: ScalarmMetric.CPUSensor
      binding: ScalarmCamelExecModel.ScalarmMetricInstanceModel.
    ScalarmVMBinding
    }

  }
  execution model ScalarmExecModel {
    execution context EC1 {
      total cost: 10.0
      application: ScalarmModel.ScalarmApplication
      cost unit: ScalarmModel.ScalarmUnits.Euro
      deployment model: ScalarmModel.ScalarmDeployment
      requirement group: ScalarmRequirement.ScalarmRequirementGroup
    }

    resource measurement RM1{
      vm instance: ScalarmModel.ScalarmDeployment.
    StorageIntensiveUbuntuGermanyInst
      value: 10.0
      reported on: 2014-12-10
      execution context: ScalarmExecModel.EC1
      metric instance: ScalarmMetricInstanceModel.RawCPUMetricInstance
    }

    assessment ScalarmAssessment{
      measurement: ScalarmExecModel.RM1
      execution context: ScalarmExecModel.EC1
      assessment time: 2014-12-10
      slo: ScalarmRequirement.CPUMetricSLO
      violated
    }
  }
}
```

16 Types

The Saloon framework [24, 23, 22] includes a type package which is mainly used to indicate the value type for property features as well as for CP variables in the respective package of WP3. However, as it was spotted that the type package could be exploited by the other packages of the CAMEL family, it was decided to be extended according to the following two aspects: (a) make a distinction between values and value types which was not in place in the original version of the type package, (b) create OCL constrains which can be used to check that the value types generated are correct, (c) create class methods for each value type which indicate whether a specific value is included in the value type. The latter extension was fundamental for being able to check whether feature properties take the correct values in feature constraints as well as the measurement values of metrics are included in the value type associated to these metrics.

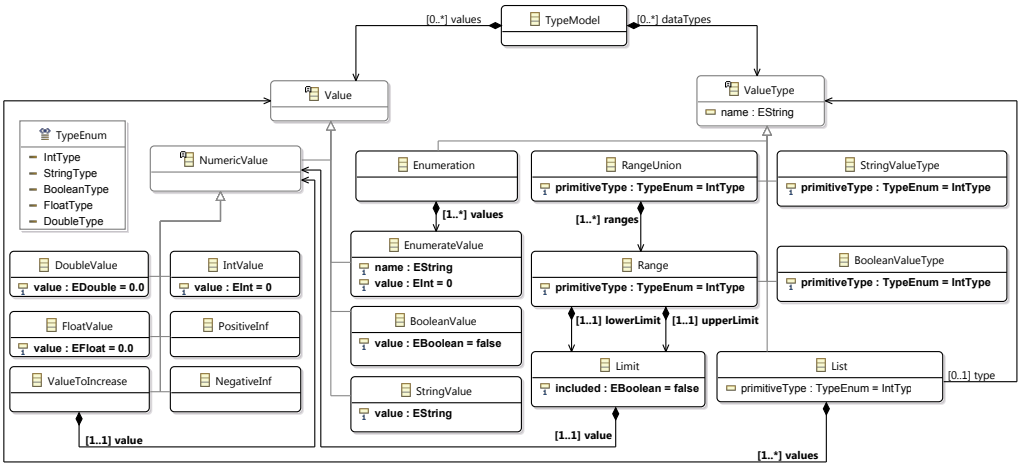

Figure 25: Class diagram of the type package

The main class in the type package, as it can be seen from Figure 25, is the TypeModel one which actually represents the container for the values and value types generated by the users/modellers. Any value is represented by the Value class which is further distinguished into the NumericValue, StringValue, BooleanValue and EnumerateValue classes. A numeric value represents any numeric value and is further distinguished into the IntValue, DoubleValue and FloatValue classes which are used to represent integer, double and float values, respectively, and each of these classes has a respective value property which maps to the respective equivalent Java type of Ecore. Special sub-classes of the NumericValue class are the NegativeInf and PositiveInf classes which represent negative and positive infinity, respectively. The instances of these classes can be used in defining one

of the two bounds of range-based value types. The StringValue and BooleanValue classes represent string and boolean values, respectively, and have a value property which maps to the respective equivalent Java type of Ecore. The EnumerateValue represents an enumerated value used for enumerations which is characterised by its content (as an integer) and name (as a String). This is very close to the way EnumeratedTypes work in Ecore where they comprise values which are identified by their name and relative position (or content) in the type.

To be able to generate range-based value types, a special class has been generated called Limit which can be used to specify the lower and upper values in the range of the value type. This class indicates the value of the limit which is of type Value as well as whether this value is included or not for the value type (see included boolean property). Through using the specific type of the limit value, we can associate not only normal values but also instances of the NegativeInf and PositiveInf classes to a limit for ranges that are open and have infinite limits in their lower or upper part.

A value type is represented by the ValueType class which is extended by six classes, namely StringValueType, BooleanValueType, Enumeration, List, Range and RangeUnion. The first two classes actually represent a basic value type which can take string or boolean values, respectively. The Enumeration class represents enumerated lists/types and has a set of EnumerateValues as a member. The List class represents lists which can have whatever types of values the user requests with the sole exception that all list values should map to only one type of value. To make this more clear, the values in a List can be of a basic type (*i.e.*, any basic sub-class of Value) or belong to a (complex) value type (*e.g.*, a integer-based range or an enumeration). To enforce this distinction, the List class contains two members from which only one must be provided with a value: a property which indicates the basic value type (through an EnumeratedType named as TypeEnum) and a reference to the complex value type (*i.e.*, instance of ValueType class). Obviously, the list of values contained is also included in the List class definition. A range-based value type is represented by the Range class which indicates the low and upper limits for the value type and the primitive type of its values (which maps to a member of the TypeEnum enumeration). In case more complex numeric value types need to be specified, the RangeUnion class can be exploited which represents a union of range-based value types. This class refers to the included range-based value types as well as to the primitive type which should be common across all the range-based value types included (*e.g.*, all range-based value types should be integer-based).

We should note here that various OCL constraints have been specified to capture and check the following cases: (a) the correct values are given for a value type (*e.g.*, the value of the limit should be of the primitive type declared for the value type), (b) correct property value combinations are given (*e.g.*, a

List cannot have values for both primitiveType property and valueType reference), (c) the values (members) of a value type should be correct (*e.g.*, a union cannot have ranges which are conflicting in terms of their limits or a List cannot have a value which is not included in the indicated basic or derived value type) and (d) the ranges in a RangeUnion should not be conflicting (*e.g.*, an integer range [1,3] and the integer set [2,4] should not be both allowed as members of a range union).

16.1 Example

Continuing the Scalarm textual model specification example, we provide in the following Listing 14 the encompassed type model which is re-used for the definition of some metrics in the respective metric model of the Scalarm CAMEL textual model. As it can be seen, two integer-based ranges and one range of doubles have been defined. The first range is used in the definition of the `CPUMetric` in Scalarm (to denote that CPU utilisation values should go from 0, included, to 100, again included), the second in the definition of the `ResponseTimeMetric` (to denote that the response time values should be from 0, non-inclusive, to 10000, inclusive, with the unit of measure being `MILLISECONDS`, and the third in the `AvailabilityMetric` definition (to denote a range of percentage of values from 0.0 to 100.0, where both bounds are included – in this way, we are able to define 99.*9 percentage values based on what is usually offered by major cloud providers).

The textual definition starts with the specification of the type model, including its name, which encloses the specification of the three ranges. Each range is defined according to the following pattern: range <range_name> primitive type: <TypeEnum> lower limit <type_of_value_spec> value <actual_value> (included)? upper limit <type_of_value_spec> value <actual_value> (included)?. For instance, in the case of the last range, we define as <range_name> the String value “DoubleRange_0_100”, as <TypeEnum> the `DoubleType` member of the enumeration type `TypeEnum`, as <type_of_value_spec> the value of `double`, as <actual_value> in the case of the lower limit the value of 0.0 and we indicate that this lower limit is included.

Listing 14: The enclosing type model of the Scalarm model in textual syntax

```
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
```

```
10      lower limit {int value 0}
11      upper limit {int value 1000 included}
12    }
13
14    range DoubleRange_0_100 {
15      primitive type: DoubleType
16      lower limit {double value 0.0 included}
17      upper limit {double value 100.0 included}
18    }
19  }
```

17 Unit

The unit of measurement is a concept that has to be clearly specified in the context of producing measurements and checking whether such measurements satisfy particular conditions. As such concept is re-used across different packages and involves good but independent modelling effort, it has been decided to be covered in a new package called unit. The exploitation of this concept is inherent in the following packages: (a) metric, where it is used to define the unit of measurement for a metric, (b) execution package where it is used to define the monetary unit for the cost of a particular application execution, and (c) the provider package where it is used to define the unit for a particular feature attribute (*e.g.*, a CoreUnit for a feature/VM property mapping to the number of cores).

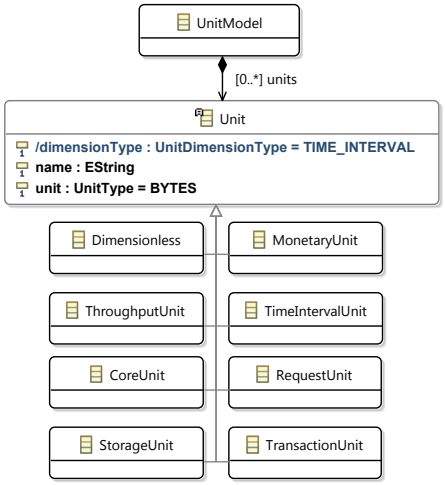

Figure 26: Class diagram of the unit package

The main class in the unit package, as it can be seen from Figure 26, is the Unit one which represents any unit. This class is abstract but is distinguished into 8 sub-classes which can be instantiated. These 8 sub-classes are the following:

- CoreUnit: represents a unit which maps to the dimension of number of cores
- Dimensionless: represents a unit which does not actually map to any dimension (*e.g.*, a unit of PERCENTAGE is dimensionless)
- MonetaryUnit: represents a monetary unit (*e.g.*, EUROS)
- RequestUnit: represents a unit which maps to the dimension of number of requests

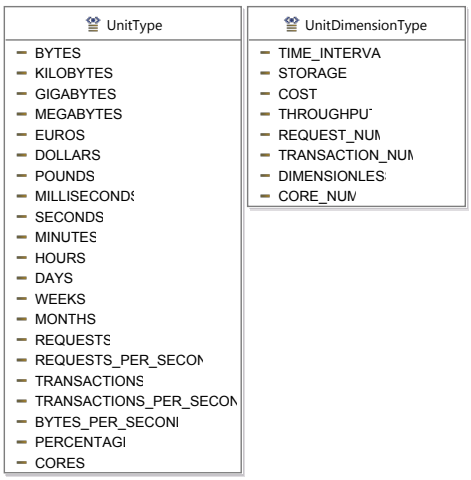

Figure 27: The enumerations defined in the class diagram of the unit package

- StorageUnit: represents a storage unit (*e.g.*, BYTES)
- ThroughputUnit: represents a unit of throughput (*e.g.*, REQUESTS_PER_SECOND)
- TimeIntervalUnit: represents a unit of time interval (*e.g.*, SECONDS)
- TransactionUnit: represents a unit which maps to the dimension of number of transactions

A unit, irrespectively of its type, is characterised by two main properties: (a) the particular UnitType it maps to and (b) the dimension (see UnitDimensionType enumeration) it concerns. UnitType is an enumeration of all possible metrics while UnitDimensionType is an enumeration of all possible unit dimensions ("all" here refers to the context of this project). Both enumerations are depicted in 27. Through specifying the class for a particular unit instance, a value for the second property does not have to be provided as it is automatically produced via the use of an OCL expression (which considers the mapping of unit sub-classes to the respective dimensions considered). Thus, in the end, the user just have to define only the actual unit to be exploited (taking values from the UnitType enumeration) (*e.g.*, SECONDS for a TimeIntervalUnit). However, the user must be careful to provide a correct actual unit. Fortunately, to assist in this latter task, an OCL constraint has been formulated which does the checking of whether the actual unit provided maps to the respective unit type specified (*e.g.*, REQUESTS do not map correctly to a unit of type TimeIntervalUnit so a validation error will be visualized).

17.1 Example

We should highlight that we can actually create a complete unit model including all possible units that can be specified based on the `UnitType` enumeration and the different types/sub-classes of units. Then, any possible unit re-use will be performed based on this model and no other unit models are further needed. Such a model could be available through the CAMEL's textual editor in order to assist in the specification of related concepts of other CAMEL packages.

Continuing the textual specification of the Scalarm model, Listing 15 depicts the part of the model pertaining to the respective unit model contained in the overall Camel model. As it can be seen, seven units have been defined which are then re-used in the context of metric definitions in the corresponding metric model. The textual definition starts with the specification of the unit model and its name and then continues with the specification of each enclosing unit. The latter specification is quite short and maps to the following pattern: <unit_class> <name_of_unit>:<unit_type>. For instance, the first unit specification indicates that a `MonetaryUnit` is defined with the "euros" name and the `EUROS` type of unit (`UnitType`).

Listing 15: The enclosing unit model of the Scalarm model in textual syntax

```
1  unit model ScalarmUnit {
2    monetary unit {Euro: EUROS}
3
4    dimensionless {CPUUnit: PERCENTAGE}
5
6    time interval unit {ResponseTimeUnit: MILLISECONDS}
7
8    dimensionless {AvailabilityUnit: PERCENTAGE}
9
10   throughput unit {SimulationsPerSecondUnit: TRANSACTIONS_PER_SECOND}
11
12   time interval unit {ExperimentMakespanInSecondsUnit: SECONDS}
13
14   transaction unit {NumberOfSimulationsLeftInExperimentUnit:
       TRANSACTIONS}
15 }
```

18 Java APIs and CDO

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

Listing 16 shows the creation of a `VM` equivalent to the one in Listing 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 in Ecore. All class instances are obtained using the `DeploymentFactory` object specific for the deployment package. This object provides a set of methods which are used to make sure that the model objects are appropriately instantiated.

Listing 16: A sample `VM` definition

```
// create a ML VM
VM ml = DeploymentFactory.eINSTANCE.createVM();
//First create VM requirement set & add it to the deployment model
VMRequirementSet mlReqs = DeploymentFactory.eINSTANCE.
    createVMRequirementSet();
mlReqs.setName("ML_VM_REQS");
ml.setVmRequirementSet(mlReqs);
sensAppDeploymentModel.getVmRequirementSets().add(mlReqs);
//Create a quantitative hardware requirement to include it in the
    requirement set
QuantitativeHardwareRequirement mlHardReq = RequirementFactory.
    eINSTANCE.createQuantitativeHardwareRequirement();
mlHardReq.setName("ML_VM_HARD_REQS");
mlHardReq.setMaxCores(0);
mlHardReq.setMaxRAM(0);
mlHardReq.setMaxStorage(0);
mlHardReq.setMinCores(2);
mlHardReq.setMinRAM(4096);
mlHardReq.setMinStorage(512);
rm.getRequirements().add(mlHardReq);
mlReqs.setQuantitativeHardwareRequirement(mlHardReq);
//Create a LOcation requirement imposing that the VM should be located
     in Scotland
LocationRequirement mlLocReq = RequirementFactory.eINSTANCE.
    createLocationRequirement();
mlLocReq.setName("ML_LOC_REC");
mlLocReq.getLocations().add(scotland);
rm.getRequirements().add(mlLocReq);
mlReqs.setLocationRequirement(mlLocReq);
//Fix other details of the VM including its name and provided host
ml.setName("ML");

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

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

Listing 17 shows the creation an `InternalComponent` equivalent to the one in Listing 1. This `InternalComponent` has an associated `Resource` representing a WAR file along with the corresponding life cycle control scripts (*e.g.*, download and install commands). It also has a Servlet container `RequiredHostingPort`, a MongoDB `RequiredCommunication`, and an HTTP `ProvidedCommunication`.

Listing 17: A sample `InternalComponent` definition

```
// create a SensApp InternalComponent give it a name
InternalComponent sensApp = DeploymentFactory.eINSTANCE.
    createInternalComponent();
sensApp.setName("SensApp");

//Associate it with a particular configuration
Configuration sensAppRes = DeploymentFactory.eINSTANCE.
    createConfiguration();
sensAppRes.setDownloadCommand("wget␣-P␣~␣http://github.com/downloads/
    SINTEF-9012/sensapp/sensapp.war;␣wget␣-P␣~␣http://cloudml.org/
    scripts/linux/ubuntu/sensapp/install_start_sensapp.sh");
sensAppRes.setInstallCommand("cd␣~;␣sudo␣bash␣install_start_sensapp.sh
    ");
sensAppRes.setName("SensAppRes");
sensApp.getConfigurations().add(sensAppRes);

//Create a provided communication element on port 8080
ProvidedCommunication restProv = DeploymentFactory.eINSTANCE.
    createProvidedCommunication();
restProv.setName("RESTProv");
restProv.setPortNumber(8080);

sensApp.getProvidedCommunications().add(restProv);

//Create a required communication with MongoDB component
RequiredCommunication mongoDBReq = DeploymentFactory.eINSTANCE.
    createRequiredCommunication();
mongoDBReq.setIsMandatory(true);
mongoDBReq.setName("MongoDBReq");
mongoDBReq.setPortNumber(0);

sensApp.getRequiredCommunications().add(mongoDBReq);

//Create a required host element which will map to the containment of
    the component by a servlet
RequiredHost servletContainerSensAppReq = DeploymentFactory.eINSTANCE.
    createRequiredHost();
servletContainerSensAppReq.setName("ServletContainerSensAppReq");

sensApp.setRequiredHost(servletContainerSensAppReq);

//Finally add the component to the deployment model
sensAppDeploymentModel.getInternalComponents().add(sensApp);
```

Listing 18 shows the creation of a `Communication` binding between the SensApp and the SensApp AdminInternalComponents, equivalent to the one in Listing 3.

Listing 18: A sample `Communication` definition

```
//Create communication by also specifying its name and the provided
    and required communications
Communication sensAppToAdmin = DeploymentFactory.eINSTANCE.
    createCommunication();
sensAppToAdmin.setName("SensAppToAdmin");
sensAppToAdmin.setProvidedCommunication(restProv);
sensAppToAdmin.setRequiredCommunication(restReq);

//Create a configuration for the communication's provided port
Configuration sensAppToAdminRes = DeploymentFactory.eINSTANCE.
    createConfiguration();
sensAppToAdminRes.setDownloadCommand("get␣-P␣~␣http://cloudml.org/
    scripts/linux/ubuntu/sensappAdmin/configure_sensappadmin.sh");
sensAppToAdminRes.setInstallCommand("cd␣~;␣sudo␣bash␣
    configure_sensappadmin.sh");
sensAppToAdminRes.setName("SensAppToAdminRes");

sensAppToAdmin.setProvidedPortConfiguration(sensAppToAdminRes);

//Add communication to deployment model
sensAppDeploymentModel.getCommunications().add(sensAppToAdmin);
```

Listing 18 shows the creation of a `Hosting` binding between the Jetty `InternalComponent` and the SL `ExternalComponent`, equivalent to the one in Listing 4.

Listing 19: A sample `Hosting` definition

```
//Create hosting, specify its name and the required and provided hosts
Hosting jettySCToVMML = DeploymentFactory.eINSTANCE.createHosting();
jettySCToVMML.setName("JettySCToVMML");
jettySCToVMML.setProvidedHost(vmMLProv);
jettySCToVMML.setRequiredHost(vmJettySCReq);

//Add hosting to the deployment model
sensAppDeploymentModel.getHostings().add(jettySCToVMML);
```

Listing 20 shows the process of saving a deployment model in a CDO repository. As mentioned, CDO uses a set of APIs which 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[11], 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 20: Saving a deployment model in a CDO repository

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

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

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

// create the session configuration
CDONet4jSessionConfiguration config = CDONet4jUtil.
    createNet4jSessionConfiguration();
config.setConnector(connector);
config.setRepositoryName("CloudMLCDORepository");

// 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("/
    sensAppResource");

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

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

Listing 21 shows the process of loading and modifying a deployment model. In this example, an SL VM is moved from the Flexiant to the AWS-EC2 cloud.

Listing 21: 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("/sensAppResource");
assertTrue(resource.getContents().get(0) instanceof DeploymentModel);
DeploymentModel model = (DeploymentModel) resource.getContents().get
    (0);

//get provided & required hosts required
RequiredHost rh = null;
for (InternalComponent ic: model.getInternalComponents()){
  if (ic.getName().equals("JettySC")){
      rh = ic.getRequiredHost();
        break;
    }
}
ProvidedHost ph = null;
for(VM vm: model.getVMs()){
  if (ic.getName().equals("ML")){
      ph = vm.getProvidedHost();
        break;
    }
}

//find previous hosting and remove it from deployment model
```

```
Hosting h = null;
for (Hosting h2: model.getHostings()){
  if (h2.getName().equals("JettySCToVMML")){
      h = h2;
        break;
    }
}
model.getHostings().remove(h);

//create new hosting and add it to the deployment model
Hosting jettySCToVMSL = DeploymentFactory.eINSTANCE.createHosting();
jettySCToVMSL.setName("JettySCToVMSL");
jettySCToVMSL.setProvidedHost(vmSLProv);
jettySCToVMSL.setRequiredHost(vmJettySCReq);
sensAppDeploymentModel.getHostings().add(jettySCToVMSL);

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

The examples above show the Java code for programmatically saving, loading, and modifying a deployment model in a CDO repository. The Java code for programmatically saving, loading, and modifying models from other packages of the CAMEL metamodel are analogous. This is because the CAMEL metamodel consists of an Ecore model, which allows for using the same Java APIs. The full version of the Java code of the SensApp example is available for reference in `http://git.cetic.be/paasage/cdo_client/blob/master/src/eu/paasage/camel/examples/SensAppCDO.java` as well as in the appendix of this document.

19 Related Work

In the cloud community, libraries such as jclouds[12] or DeltaCloud[13] provide generic APIs abstracting over the heterogeneous APIs of IaaS providers, thus reducing cost and effort of deploying multi-cloud applications. While these libraries effectively foster the deployment of cloud-based applications across multiple cloud infrastructures, they remain code-level solutions, which make design changes difficult and error-prone. More advanced frameworks such as Cloudify[14], Puppet[15], or Chef[16] 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 [28] project tackles the vendor lock-in problem by providing an

[12] `http://www.jclouds.org`
[13] `http://deltacloud.apache.org/`
[14] `http://www.cloudifysource.org/`
[15] `https://puppetlabs.com/`
[16] `http://www.opscode.com/chef/`

API for provisioning and deployment of multi-cloud applications. This solution is also limited to the code level.

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [21] 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.

As part of the joint standardisation effort of MODAClouds and PaaSage, I have 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, on 3 September, approved by unanimous consent the formation of the Instance Model Ad Hoc group to be co-led by Alessandro Rossini and Sivan Barzily from GigaSpaces. This will guarantee that the contribution of CAMEL will be partly be integrated into the standard.

20 Conclusions and Future Work

In this document, we have provided the final version of the documentation of CAMEL. In particular, 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 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.

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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Keith Jeffery and Tom Kirkham. *D1.6.1 – Initial Architecture Design*. PaaSage project deliverable. Oct. 2013.

[10] 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.

[11] 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.

[12] Kyriakos Kritikos and Dimitris Plexousakis. “Semantic QoS Metric Matching”. In: *ECOWS 2006: 4th European Conference on Web Services*. IEEE Computer Society, 2006, pp. 265–274. DOI: 10.1109/ECOWS.2006.34.

[13] Kyriakos Kritikos et al. *D4.1.1 – Prototype Metadata Database and Social Network*. PaaSage project deliverable. Mar. 2014.

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

[15] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. Special Publication 800-145. National Institute of Standards and Technology, Sept. 2001.

[16] Object Management Group. *Meta-Object Facility Specification*. 2.4.2. http://www.omg.org/spec/MOF/2.4.2/. Apr. 2014.

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

[18] Object Management Group. *Query/View/Transformation*. 1.1. http://www.omg.org/spec/QVT/1.1/. Jan. 2011.

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

[20] Object Management Group. *XML Metadata Interchange Specification*. 2.4.2. http://www.omg.org/spec/XMI/2.4.2/. Apr. 2014.

[21] 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.

[22] 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.

[23] 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.

[24] 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, 2:1–2:6. ISBN: 978-1-4503-1161-8. DOI: 10.1145/2168697.2168699.

[25] Alessandro Rossini, Nikolay Nikolov, Daniel Romero, Jörg Domaschka, Kiriakos Kritikos, Tom Kirkham and Arnor Solberg. *D2.1.2 – CloudML Implementation Documentation (First version)*. PaaSage project deliverable. Apr. 2014.

[26] 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.

[27] Alessandro Rossini, Arnor Solberg, Daniel Romero, Jörg Domaschka, Kostas Magoutis, Lutz Schubert, Nicolas Ferry and Tom Kirkham. *D2.1.1 – CloudML Guide and Assesment Report*. PaaSage project deliverable. Oct. 2013.

[28] 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.

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