CAMEL Documentation v2015.9

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

18th February 2016

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 [16]. 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). This solution allows exploiting the peculiarities of each cloud service and hence optimising performance, availability, and cost of the applications.

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

*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, 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 CAMEL documentation. 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.

Compared to the initial version of the CAMEL documentation (previously known as CLOUDML implementation documentation, *cf.* D2.1.2 [25]), we have added the description of the new concepts in CAMEL, revised the description of the existing concepts, and migrated the examples from tree-based to textual syntax.

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 [10]) 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). This solution allows exploiting the peculiarities of each cloud service and hence optimising performance, availability, and cost of the applications.

Models can be specified using general-purpose languages like the Unified Modeling Language (UML) [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) [7, 5, 6], Saloon [23, 22, 24], and the organisation part of CERIF [9]. In addition, CAMEL integrates new DSLs developed within the project, such as the Scalability Rule Language (SRL) [12, 3].

CAMEL enables PaaSage users to specify multiple aspects of multi-cloud applications, such as provisioning and deployment, 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].

Structure of the document

The remainder of the document is organised as follows. Section 2 describes the role of CAMEL models in the PaaSage workflow. Section 3 presents some technologies used to design and implement CAMEL. Sections 6-17 present the various packages of the CAMEL metamodel along with corresponding sample models in concrete syntax. Section 18 exemplifies the usage of Java APIs to programmatically manipulate and persist models. Finally, Section 19 compares the proposed approach with related work, while Section 20 draws conclusions and outlines plans for future work.

2 CAMEL and the PaaSage workflow

In order to facilitate the integration across the components managing the life cycle of 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.

Figure 1 shows the PaaSage workflow. The white trapezes represent the activities performed by the PaaSage user. The white rectangles represent the processes executed by the PaaSage platform. The coloured shapes represent the modelling artefacts, whereby the blue ones pertain to the modelling phase, the red ones to the deployment phase, and the green ones to the execution phase.

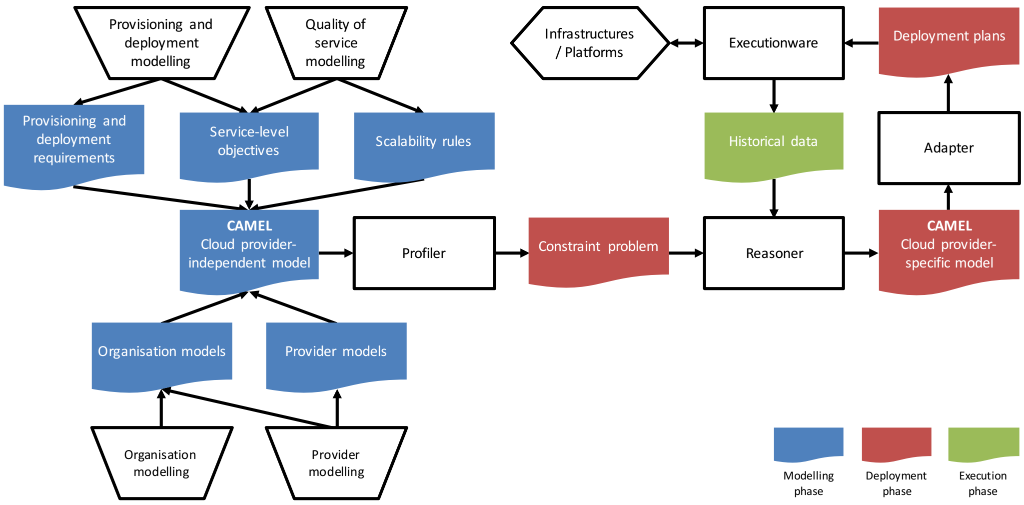

Figure 1: CAMEL models in the PaaSage workflow

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[1], 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.

[1]http://sensapp.org/

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. The PaaSage platform records data about the application execution from the Executionware, which allows the Reasoner to continuously revise the solution to the constraint problem to better exploit the cloud infrastructures.

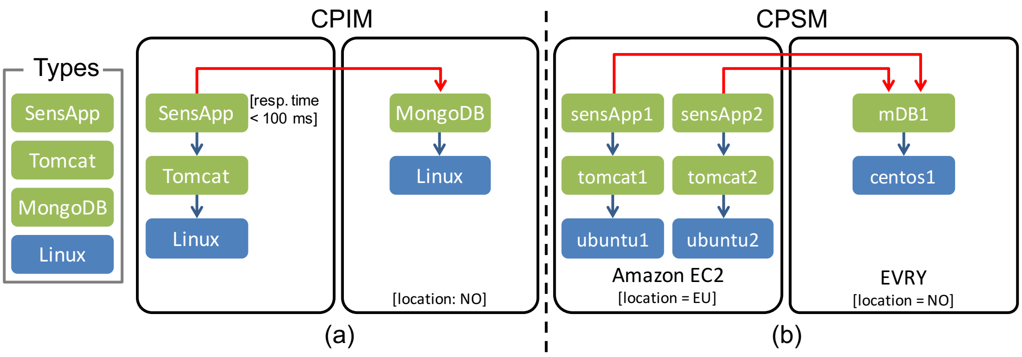

Figure 2: Sample CAMEL models: (a) CPIM; (b) CPSM

3 Technologies

In order to design and implement CAMEL, we adopted the Eclipse Modelling Framework (EMF)[2] along with Object Constraint Language (OCL) [18], Xtext[3], and Connected Data Objects (CDO)[4]. 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 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).

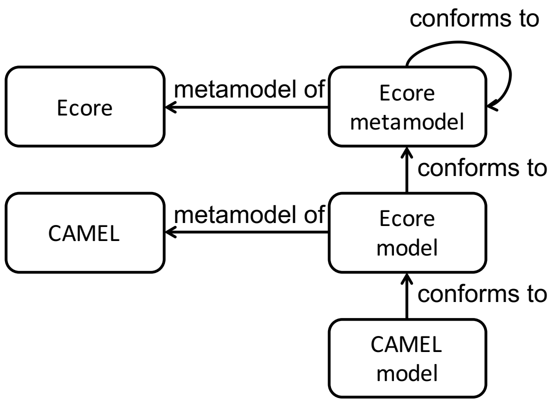

Figure 3: Ecore-based modelling stack in PaaSage

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)[5] or Graphical Editing Framework (GEF)[6] to manually create a custom graphical editor.

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

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 models and metamodels.

Eclipse OCL[7] component 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 PaaSage users 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 in the Metadata Database (MDDB) (*cf.* D4.1.2 [14]) 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:

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

- 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. These steps have been tested with the latest version of Eclipse, which at the time of writing is Eclipse Mars v4.5.1. 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 the remainder of this document**
- Edit the file

5 Naming Conventions

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

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

6 CAMEL Metamodel

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

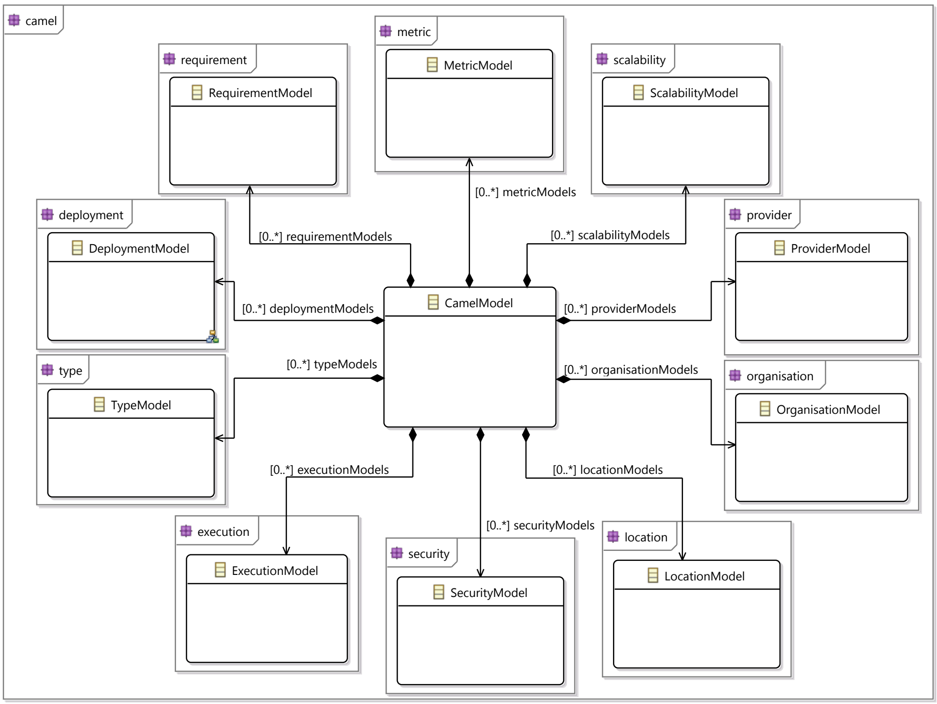

Figure 4: Class diagram of the CAMEL metamodel including packages

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[8] [7, 5, 6], which was developed in collaboration with the MODAClouds project[9]. 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* [15]. 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.

[8] http://cloumdml.org

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

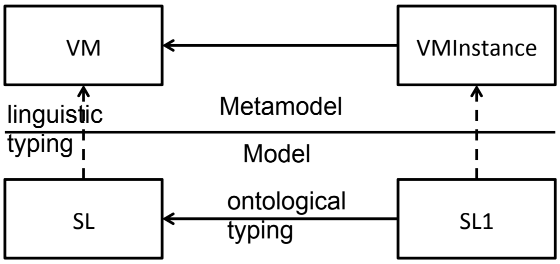

Figure 5: Linguistic and ontological typing

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.

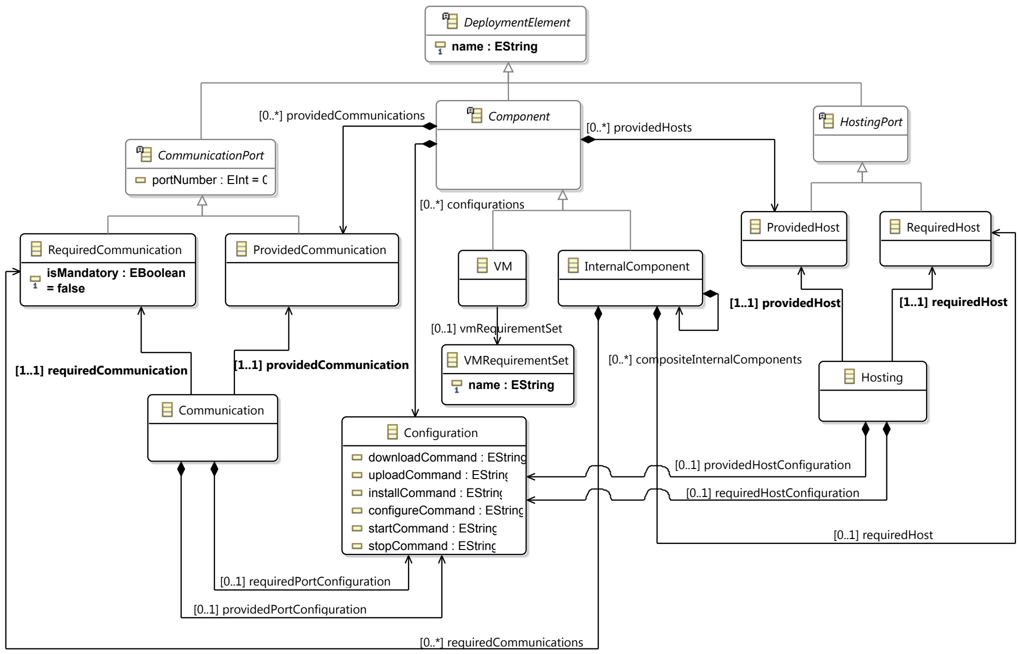

Figure 6: Class diagram of the type part of the deployment package

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) maintained 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[10] 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?p=camel.git&a=blob&f=examples/Scalarm.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.

`internal component ExperimentManager` specifies a reusable type of the component Experiment Manager. `provided communication ExpManPort` represents that the Experiment Manager provides its features through port 443. `required communication StoManPortReq` and `InfSerPortReq` specify that the Experiment Manager requires features from the Information Service through port 11300 and from the Storage Manager through port 20001, respectively. The property `mandatory` of the latter specifies that the Execution Manager depends on the features of the Storage Manager (hence, the Storage Manager has to be deployed and started before the Execution Manager). `required host CoreIntensiveUbuntuGermanyReq`

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

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).

`configuration ExperimentManagerConfiguration` specifies the commands to handle the life cycle of the Experiment Manager. `download`, `install`, and `start` specify the Unix shell scripts for downloading, installing, and starting the Experiment Manager, respectively.

Note that, although not shown in this example, it is also possible to specify the `configure`, and `stop` commands of a component.

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

Listing 1: Scalarm sample `internal component`

```
camel model ScalarmModel {

deployment model ScalarmDeployment {

  internal component ExperimentManager {
    provided communication ExpManPort {port: 443}
    required communication StoManPortReq {port: 20001 mandatory}
    required communication InfSerPortReq {port: 11300}
    required host CoreIntensiveUbuntuGermanyHostReq

    configuration ExperimentManagerConfiguration {
      download: 'cd ~ && wget https://github.com/kliput/scalarm_service_scripts/archive/paasage.tar.gz && sudo apt-get update && sudo apt-get install -y groovy ant && tar -zxvf paasage.tar.gz && cd ~/scalarm_service_scripts-paasage'
      install: 'cd ~/scalarm_service_scripts-paasage && ./experiment_manager_install.sh'
      start: '~/scalarm_service_scripts-paasage/experiment_manager_start.sh'
    }
  }
...
```

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

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

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

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

Listing 3: Scalarm 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 port.

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.

`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. `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: Scalarm sample `hosting`

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

7.4 Component, Communication, and Hosting instances

Note: This subsection is for PaaSage developers only. The instances within the deployment model are automatically generated by the PaaSage platform during the deployment phase (*cf.* Section 2).

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

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) in textual syntax for illustrative purposes. Note that the instances within the deployment model must otherwise be manipulated through the Java API (*cf.* Section 18).

`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).

`internal component instance ExperimentManagerInst` specifies an instance of the component `ExperimentManager`. `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: Scalarm 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, *i.e.*, deploying and executing at least one instance of each of its components, the Executionware can solely rely on the information provided in the deployment model within a CAMEL model. Hence, the Executionware does not make any assumptions besides the information provided. In particular, this means: only ports of communications specified in the CAMEL model are guaranteed to be opened; other ports may be blocked by either the cloud infrastructure or operating system.

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

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

Life Cycle Scripts for Unix-based Applications

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

Note that the different handlers are not necessarily executed in the very same instance of the Unix shell. This means that custom environment variables set in a handler (*e.g.*, in the `download` command) are not necessarily available in later handlers. If such information is required, the only approach is to write the necessary data to a file and `source` this file in later handlers. For GNU/Linux deployments, all component instances are run within an own Docker container[11] in order to enable a maximum of isolation between the instances. This has a consequence on user handling and networking: As for the users, this means that all commands are executed as `root`. Also, the handlers cannot assume that any other user beside `root` exists in the system. Hence, if further users are required, the handlers are responsible for creating them. As for networking, the effects on both IP addresses and port numbers are discussed in the following.

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

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

Port Numbers in the Execution Environment: Moreover, the port numbers used within the container do not necessarily match the port numbers as used by the operating system hosting the Docker container. Indeed, the Executionware will not force the use of any fixed port numbers outside the container in order to allow maximum flexibility. Again, the port numbers are passed to the `configuration` and `start` handlers as environment variables. The name of the variable is based on the name of the `provided communication` from the deployment model. For instance, the provided port `ExpManPort` from Listing 1 is mapped to the following three environment variables:

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

- `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`: provides access to the public IP addresses and public ports of all downstream component instances.
  `<stoman1publicip>:<public_port>,<stoman2publicip>:<public_port>`
- `CLOUD_STOMANPORTREQ`: provides access to the cloud-internal IP addresses and cloud-internal ports of all downstream component instances. Note that addresses of component instances not hosted in the same cloud as the local component instance are still in the list, but very likely traffic cannot be routed to them.
  `<stoman1cloudip>:<cloud_port>,<stoman2cloudip>:<cloud_port>`
- `CONTAINER_STOMANPORTREQ`: provides access to the container-internal IP addresses and container-internal ports of all downstream component instances. Note that addresses of component instances not hosted in the very same container as the local component instance are still in the list, but very likely cannot be routed to.
  `<stoman1containerip>:<container_port>,<stoman2containerip>:<container_port>`

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

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

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

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

Life Cycle Scripts for Windows-based Applications

For Windows-based deployments the Executionware relies on the same deployment model as GNU/Linux deployments, mainly the `configuration` block of `internal components`. All handlers are executed as a single Powershell script. The return value will be interpreted and a value different from `0` will move the instance to an error state. Information about ports, connection and the local host is provided via environment variables.

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

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

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

- `PUBLIC_IP`: the public IP address of the virtual machine running the component, if available.

Port Numbers in the Execution Environment As Windows components just run in an unique folder and not in a container like Unix components there is a small difference. The port numbers of Windows components match the port numbers of the virtual machine's operating system. The port numbers are passed to the `configuration` and `start` handlers as environment variables. The name of the variable is based on the name of the provided communication from the deployment model. Considering the example from Listing 1 as a Windows component the resulting two environment variables are set:

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

Outgoing Connections in the Execution Environment The mapping of required communications is similar to Unix components (*cf.* Section 7.5) except there is no need to map the communication to a container-internal IP. Again considered Listing 1 as a Windows component the required port is mapped to the following environment variables:

- `PUBLIC_STOMANPORTREQ`: provides access to the public IP addresses and public ports of all downstream component instances.
  <stoman1publicip>:<public_port>,<stoman2publicip>:<public_port>
- `CLOUD_STOMANPORTREQ`: provides access to the cloud-internal IP addresses and cloud-internal ports of all downstream component instances. Note that addresses of component instances not hosted in the same cloud as the local component instance are still in the list, but very likely cannot be routed to.
  <stoman1cloudip>:<cloud_port>,<stoman2cloudip>:<cloud_port>

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

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

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

8 Requirements

The requirement package provides the concepts to specify requirements for 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.

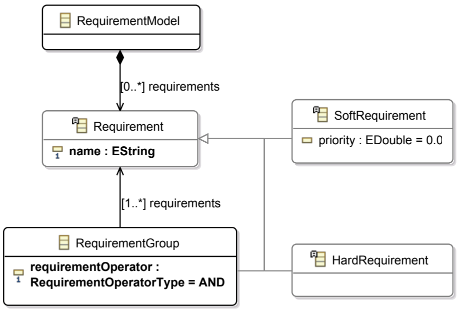

Figure 7: Portion of the class diagram of the requirement package related to its main concepts

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. These priorities must be specified in the scale from 0.0 to 1.0 and the respective soft requirements could refer to metrics that are generated based on normalisation functions also taking values from 0.0 to 1.0.

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

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

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.

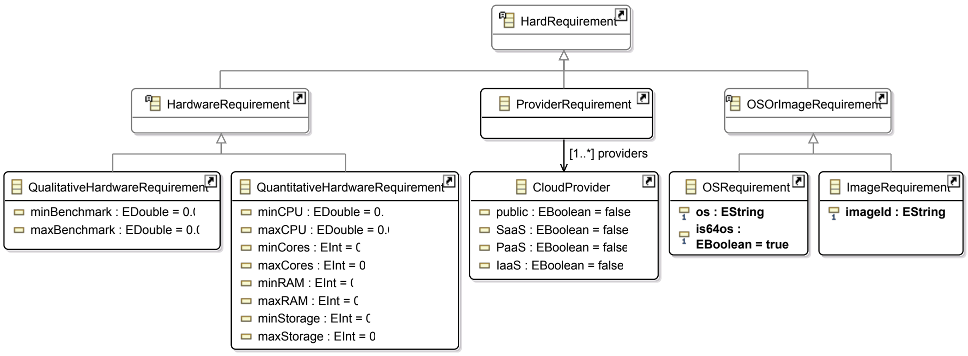

Figure 8: Portion of the class diagram of the requirement package related to hardware, OS, image, and provider requirements

A HardwareRequirement is a hard requirement. It can be referred to by a VMRequirementSet (*cf.* Section 7). It can be specialised into a QualitativeHardwareRequirement, which represents a requirement on the performance of a virtual machine instance, or a QuantitativeHardwareRequirement, which represents a requirement on the virtual hardware of a virtual machine. The properties min- and maxBenchmark of QualitativeHardwareRequirement represent the range of benchmark results that a virtual machine instance must satisfy. Note that the benchmarking itself is currently not supported 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 VM-RequirementSet (*cf.* Section 7). It can be specialised into a OSRequirement, which represents a requirement on the operating system run by a virtual machine, or a ImageRequirement, which represents a requirement on the image deployed on a virtual machine. The property os of OSRequirement represents the required operating system (*e.g.*, "Ubuntu", "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.

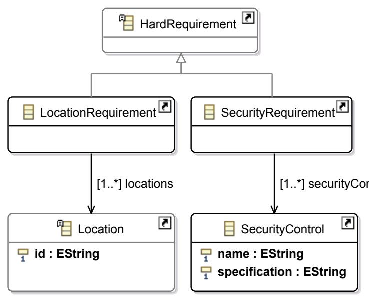

Figure 9: Location and security requirements classes

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.

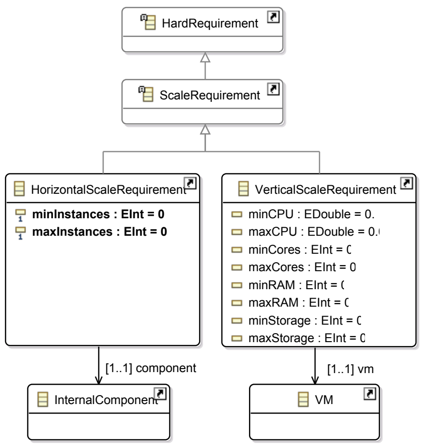

Figure 10: Scaling requirements classes

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.

8.6 Service Level Objectives

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

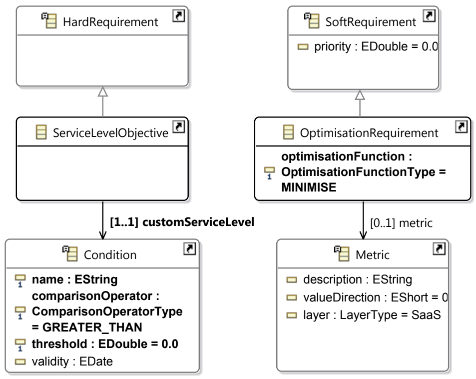

Figure 11: SLOs and optimisation requirements classes

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 (*i.e.*, the corresponding measurement values must not cross a particular threshold).

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 be assigned 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.

`quantitative hardware 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).

`os 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).

`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).

`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).

`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).

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

Listing 6: Scalarm requirement model

```
requirement model ScalarmRequirement {

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

  os Ubuntu {os: 'Ubuntu' 64os}

  location requirement GermanyReq {
    locations [ScalarmLocation.DE]
```

```
  }

  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.

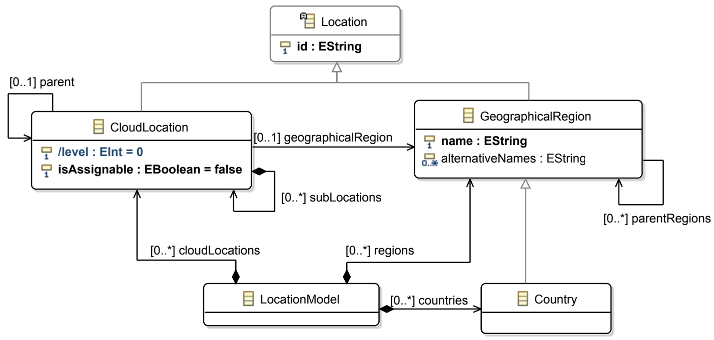

Figure 12: Class diagram of the location package

A Location represents a physical or virtual location. It can be specialised into a GeographicalRegion, which represents a geographical region, or a CloudLocation, which represents virtual cloud location in a cloud (*e.g.*, Amazon EC2 eu-west-1). The property name of GeographicalRegion represents the name in English, while the property alternativeNames represents possible alternative names in other natural languages. A geographical region can refer to a parent region, which allows creating hierarchies of geographical regions. 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 a 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.

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

Listing 7: Scalarm 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 Rule Language (SRL) [12, 3]. SRL enables the specification of rules that support complex adaptation scenarios of multi-cloud applications. In particular, SRL provides mechanisms for specifying multi-cloud behaviour patterns, metric aggregations, and the scaling actions to be enacted in order to change the provisioning and deployment of an application. In the following, we describe and exemplify the main concepts in the metric package, namely metrics and properties in Section 10.1 and scheduling and conditions in Section 10.3. The scalability aspects are then presented in Section 11.

10.1 Metrics and Properties

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

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

Metrics

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

Metric Formulas

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

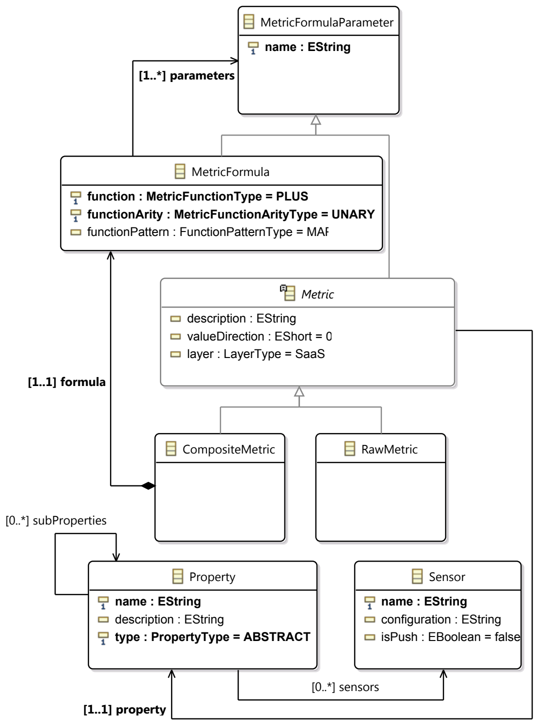

Figure 13: Portion of the class diagram of the metric package related to its main concepts

The MetricFunctionType represents the pre-defined function types, which include mean (MEAN), standard deviation (STD), addition (ADD), subtraction (MINUS), division (DIV), average (AVG), 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 maps only to one parameter, which should be of type metric or metric formula.

The FunctionPatternType classifies the strategy used to combine sets of metric instances (*cf.* Section 10.1).

Properties

Any Metric also refers to a measurable Property, which represents the measured non-functional property of a component or virtual machine. The attribute type represents the kind of property, where a value of MEASURABLE represents that the property can be measured, such as response time or CPU load, while a value of ABSTRACT represents that the property is not measurable. An abstract property is realised by its sub-properties.

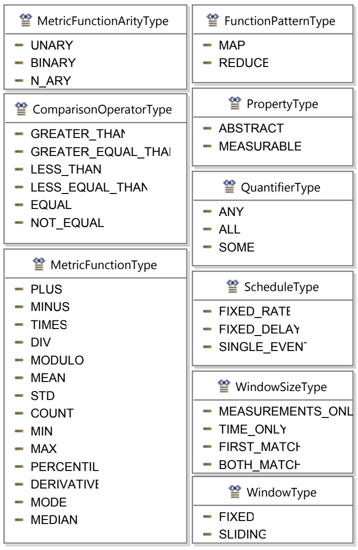

Figure 14: Portion of the class diagram of the metric package related to enumeration types

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.

Metric Instances

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

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

The data associated to a composite metric instance CMI is computed from the data of other metric instances according to the computation specified by the metric CMI is an instance of. It is important to note that from the descriptions given so far, there exist multiple ways of creating composite metric instances

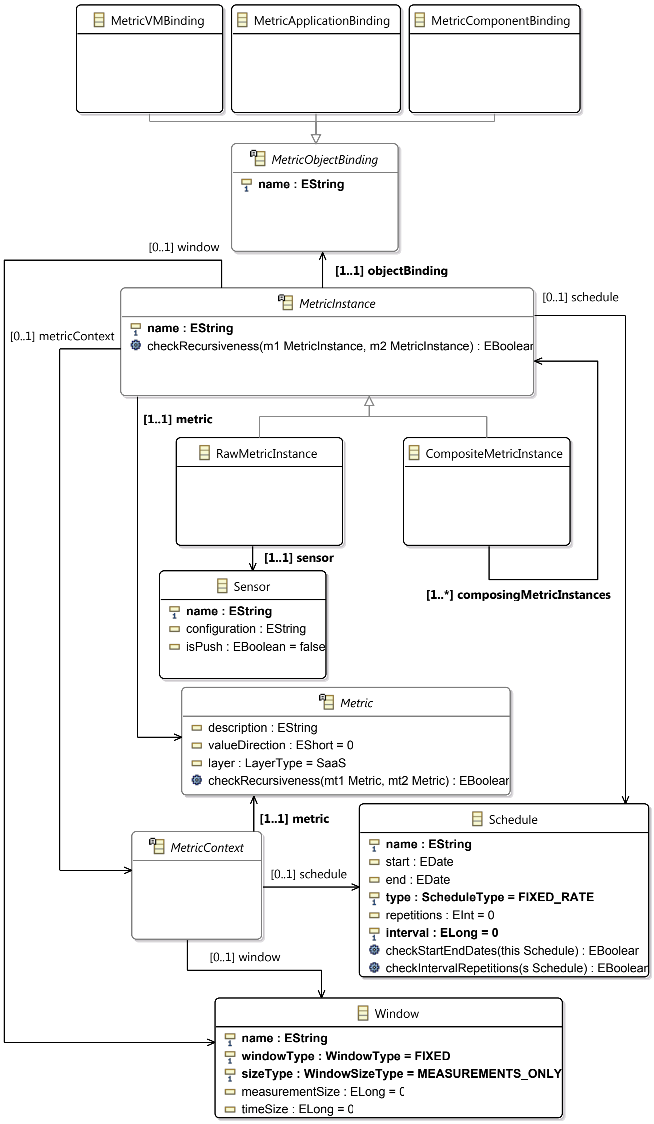

Figure 15: Portion of the class diagram of the metric package related to metric instances

from a single metric. In order to narrow down and the creation, the FunctionPatternType is used as demonstrated in the following example:

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

Metric Bindings

Each MetricInstance refers to a particular sub-class of MetricObjectBinding: MetricComponentBinding, MetricVMBinding, MetricApplicationBinding. Bindings are used by the runtime system to configure the monitoring system. 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.

10.2 Windows and Scheduling

A user has the possibility to further refine how computations of composite metrics shall be performed. This enabled by the Windows and Schedules.

Windows

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

Schedules

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.

10.3 Conditions

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

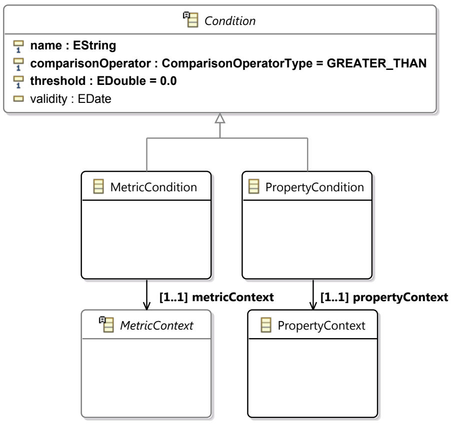

Figure 16: Portion of the class diagram of the metric package related to conditions

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

Metric Conditions

A MetricCondition represents a constraint imposed on a metric. A constraint is violated when the comparison of the values of the instances of this metric with the threshold of the condition using the metric's operator is false. The violation of a metric condition will lead to the triggering of a simple, non-functional event (*cf.* Section 11.3) and/or a violation of an SLO (which is reflected in the respective SLOAssessment – *cf.* Section 15). Note that constraints are not expressed on metric instances but on metrics. This enables the re-use of metric conditions in different execution contexts as well as guides the production of the instances of the metrics involved in these conditions to enable the assessment of the conditions under the respective particular execution contexts.

Property Conditions

A PropertyCondition represents a condition on a property. This way, it is possibly to specify, *e.g.*, constraints on cost for the whole application or one or all of its components. Then, it is up to the PaaSage 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, it refers to a monetary unit (*e.g.*, euros) and time interval unit (*e.g.*, seconds) to allow assessing the cost per time and not in absolute terms.

Condition Contexts

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

The qualifier relative represents the minimum and maximum percentage of application or component/VM instances on which the condition must hold. The qualifier absolute, in turn, represents the minimum and maximum number of instances of an application or component/VM on which the condition must hold. Four modelling constructs capture the two types of quantification: (a) quantifier: refers to a QuantifierType, which represents the set of instances to consider (all, any, some) in order to evaluate the condition – in case of *some*, the constructs (b)–(c) can be used to clarify the type and limits of quantification; (b) isRelative: defines whether a relative or absolute quantification is concerned; (c) minQuantity: represents the minimum relative or absolute value of instances; (d) maxQuantity: represents the maximum relative or absolute value of instances.

Note that the CAMEL users should be careful with providing correct min and max instance values in case the quantifier type is SOME: 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].

10.4 Metric Context

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

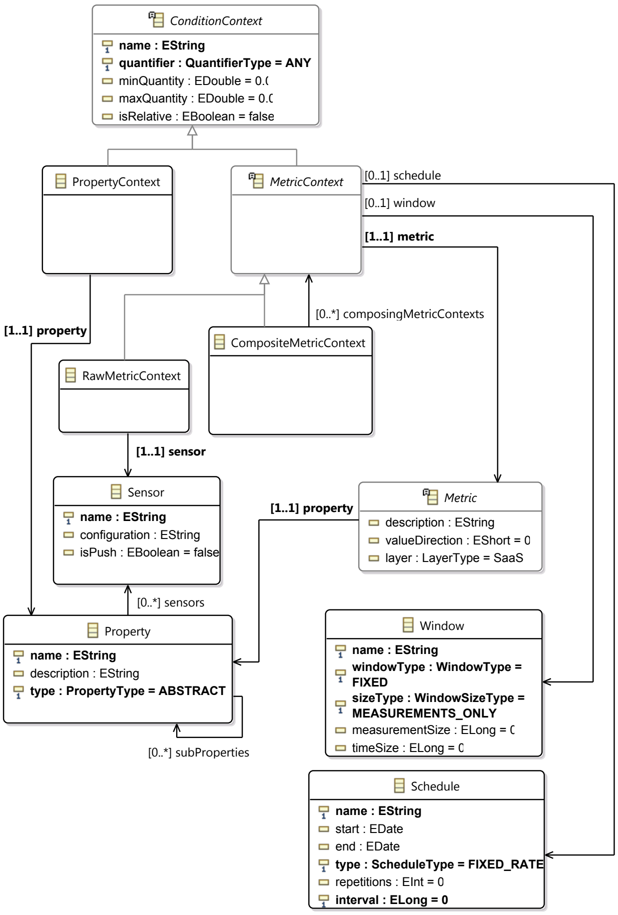

Figure 17: Portion of the class diagram of the metric package related to context

Depending on which element is associated, *i.e.*, metric or property, a ConditionContext is further specialised into a PropertyContext and MetricContext. A PropertyContext represents the property to be measured and evaluated. A MetricContext represents the metric to be used in the evaluation of the condition[12]. For composed metrics, CompositeMetricContext represents a reference to the contexts of the

[12]In the scope of PaaSage, it is the responsibility of the runtime system to generate the instances of the metrics 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.

composing metrics of the current metric. For raw metrics, RawMetricContext represents a reference to the sensor that produces the measurements of this metric.

The PaaSage runtime generates context whenever possible so that it is not necessary to create all composing contexts by hand. This is possible as some information is inherited from the composite metric contexts to its composing contexts. Consequently, the definition of a context is only obligatory when information should not be inherited but differentiated for a specific composing context. For instance, we have the metric of *response time* which is calculated by the addition of the metrics of *execution time* and *network latency*. Then, in the 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.

11 Scalability Rules

The scalability package of the CAMEL metamodel is also based on SRL [12, 3]. SRL is inspired by the Esper Processing Language (EPL)[13] 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). A ScalabilityRule refers to a set of ScaleRequirements that restrict how scaling actions are performed. It also refers to Entities such as the user or the organisation (*cf.* Section 13).

Note that the CAMEL user should be careful with modelling 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 with the scale requirements posed. A scale action conflicts with a scale requirement in the following two cases: (a) it is a HorizontalScalingAction, the requirement is of the respective type HorizontalScaleRequirement, and the amount of instances to scale-in or out does not conform to the range limit dictated by the requirement (*i.e.*, amount is greater than the difference between the upper and lower values in the range 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 upper and lower values in the range limit dictated by the requirement for this VM characteristic.

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

11.2 Actions

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

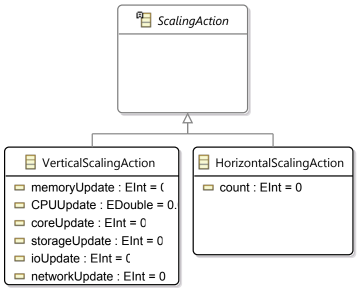

Figure 18: Portion of the class diagram of the scalability package related to actions

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

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

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.

An Event can be specialised into a SimpleEvent or an EventPattern. The SimpleEvent, in turn, can be specialised into a FunctionalEvent or a NonFunctionalEvent.

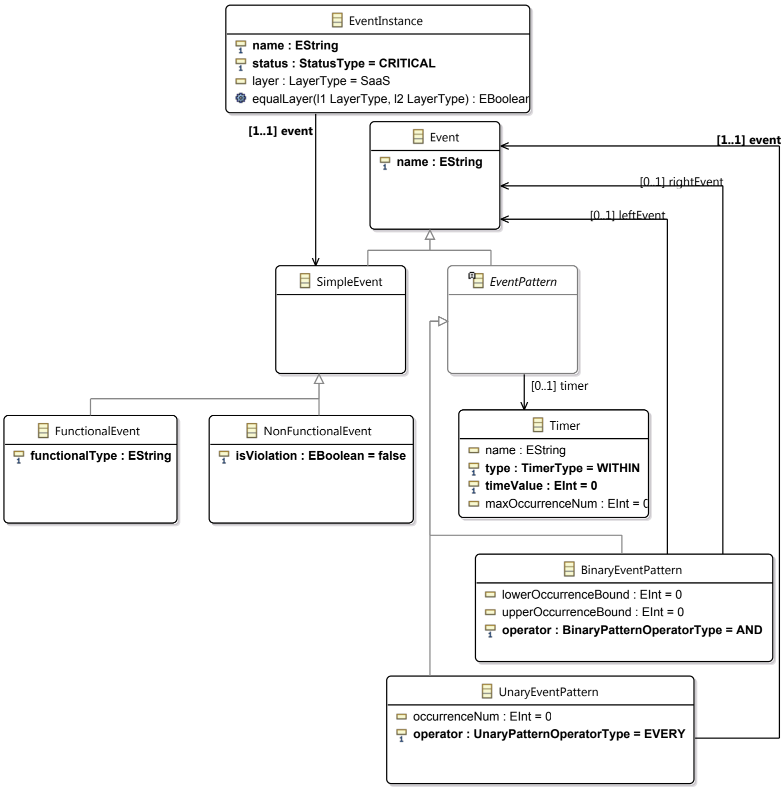

Figure 19: Portion of the class diagram of the scalability package related to events

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, which defines the threshold for the metric.

An EventInstance represents the actual (measurement) data associated with a particular event that occurred in the system (*e.g.*, the actual measured value, 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 enabling the evaluation/assessment of the events.

Events are grouped by EventPatterns, which can be specialised into a BinaryEventPattern or a UnaryEventPattern.

Binary Event Patterns

A BinaryEventPattern uses a binary operator to associate either two Events with each other or one event with a Timer. The property operator can be set to one of the common, logical operators such as AND and OR, the order operator PRECEDES, and the occurrence operator REPEAT_UNTIL (see Figure 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 this case, the CAMEL user should define the lower and/or upper bounds for the number of event occurrences.

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

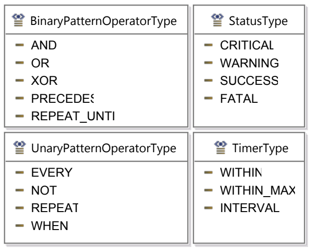

Figure 20: Portion of the class diagram of the scalability package related to enumeration types

Unary Event Patterns

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

Timers

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

Pattern Example

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.

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. In order to trigger the scalability rule, the following composite condition must be assessed:

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

Based on the above analysis, we created a scalability and metric model which specify respectively: (a) the scalability rule along with the events used to trigger it, and (b) the metrics and conditions that, when evaluated, trigger the action of the scalability rule.

Listing 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 8: Scalarm scalability model

```
scalability model ScalarmScalability {

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

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

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

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

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

requirement model ScalarmRequirement {

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

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 Simulation Manager 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 event creation action that would lead to changing the deployment model of the cluster.

Listing 9: Scalarm metric model

```
metric model ScalarmMetric {

  window Win5Min {
    window type: SLIDING
    size type: TIME_ONLY
    time size: 5
    unit: ScalarmModel.ScalarmUnit.minutes
  }

  window Win1Min {
    window type: SLIDING
    size type: TIME_ONLY
    time size: 1
    unit: ScalarmModel.ScalarmUnit.minutes
  }

  schedule Schedule1Min {
    type: FIXED_RATE
    interval: 1
    unit: ScalarmModel.ScalarmUnit.minutes
  }

  schedule Schedule1Sec {
    type: FIXED_RATE
    interval: 1
    unit: ScalarmModel.ScalarmUnit.seconds
  }

  property CPUProperty {
    type: MEASURABLE
    sensors [ScalarmMetric.CPUSensor]
  }

  sensor CPUSensor {
    configuration: 'cpu_usage;de.uniulm.omi.cloudiator.visor.sensors.
    CpuUsageSensor'
    push
  }

  raw metric CPUMetric {
    value direction: 0
    layer: IaaS
    property: ScalarmModel.ScalarmMetric.CPUProperty
    unit: ScalarmModel.ScalarmUnit.CPUUnit
    value type: ScalarmModel.ScalarmType.Range_0_100
  }

  composite metric CPUAverage {
    description: "Average of the CPU"
    value direction: 1
```

```
    layer: PaaS
    property: ScalarmModel.ScalarmMetric.CPUProperty
    unit: ScalarmModel.ScalarmUnit.CPUUnit

    metric formula Formula_Average {
      function arity: UNARY
      function pattern: MAP
      MEAN( ScalarmModel.ScalarmMetric.CPUMetric )
    }
  }

  raw metric context CPUMetricConditionContext {
    metric: ScalarmModel.ScalarmMetric.CPUMetric
    sensor: ScalarmMetric.CPUSensor
    component: ScalarmModel.ScalarmDeployment.SimulationManager
    quantifier: ANY
  }

  raw metric context CPURawMetricContext {
    metric: ScalarmModel.ScalarmMetric.CPUMetric
    sensor: ScalarmMetric.CPUSensor
    component: ScalarmModel.ScalarmDeployment.SimulationManager
    schedule: ScalarmModel.ScalarmMetric.Schedule1Sec
    quantifier: ALL
  }

  composite metric context CPUAvgMetricContextAll {
    metric: ScalarmModel.ScalarmMetric.CPUAverage
    component: ScalarmModel.ScalarmDeployment.SimulationManager
    window: ScalarmModel.ScalarmMetric.Win5Min
    schedule: ScalarmModel.ScalarmMetric.Schedule1Min
    composing metric contexts [ScalarmModel.ScalarmMetric.
    CPURawMetricContext]
    quantifier: ALL
  }

  composite metric context CPUAvgMetricContextAny {
    metric: ScalarmModel.ScalarmMetric.CPUAverage
    component: ScalarmModel.ScalarmDeployment.SimulationManager
    window: ScalarmModel.ScalarmMetric.Win1Min
    schedule: ScalarmModel.ScalarmMetric.Schedule1Min
    composing metric contexts [ScalarmModel.ScalarmMetric.
    CPURawMetricContext]
    quantifier: ANY
  }

  metric condition CPUMetricCondition {
    context: ScalarmModel.ScalarmMetric.CPUMetricConditionContext
    threshold: 80.0
    comparison operator: >
  }
  metric condition CPUAvgMetricConditionAll {
    context: ScalarmModel.ScalarmMetric.CPUAvgMetricContextAll
    threshold: 50.0
    comparison operator: >
  }

  metric condition CPUAvgMetricConditionAny {
    context: ScalarmModel.ScalarmMetric.CPUAvgMetricContextAny
```

```
      threshold: 80.0
      comparison operator: >
    }
}
```

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 cloud applications and selecting compatible cloud providers by leveraging upon feature models [2] and ontologies [8]. In the following, we describe and exemplify the main concepts in the provider package.

Figure 21 shows the class diagram of the provider package.

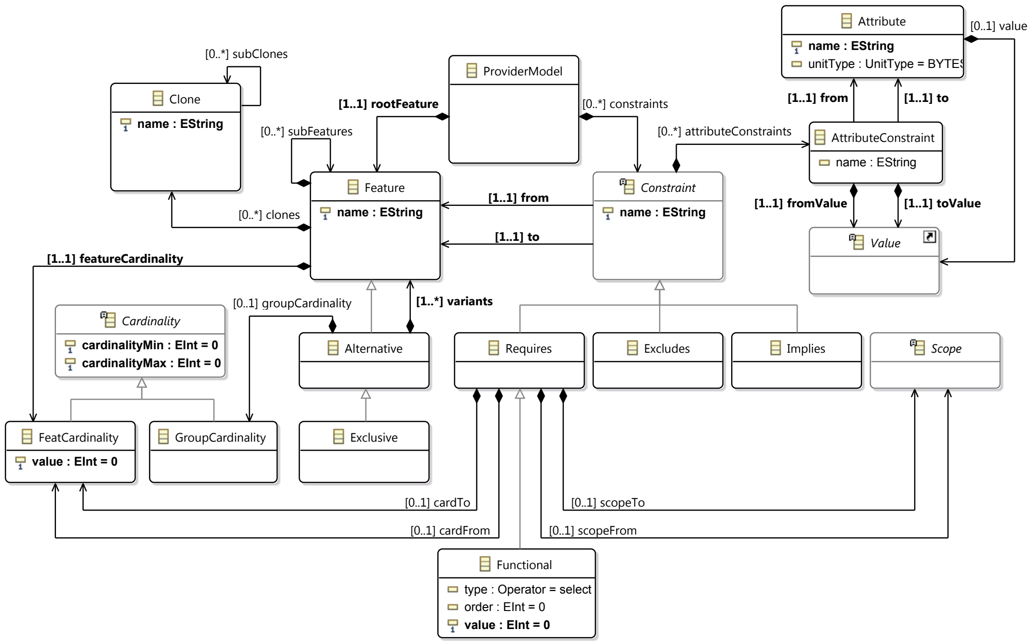

Figure 21: Class diagram of the provider package

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

A Constraint represents a typical restriction in binary feature models [11]. A Constraint can be an Implies constraint, meaning that a given feature requires another feature when selected (*i.e.*, both features have to be together in a valid configuration), or an Excludes constraint, meaning that one feature excludes another one when selected (*i.e.*, both features can not be together in a valid configuration). A Constraint can also be a Requires constraint, enabling the specification of restrictions of the form:

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

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

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

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

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

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

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

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

Note that the CAMEL user should be careful with modelling attributes: (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[14] specified using the CAMEL textual syntax, while Figure 22 depicts the same model using the FODA notation [11].

`root feature GWDG` specifies the attributes characterising the GWDG provider, such as the deployment model (not to be confused with the CAMEL deployment model), service model and availability. `attribute DeploymentModel` specifies that GWDG is a private cloud. `attribute ServiceModel` specifies that

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

GWDG is a IaaS. `attribute Availability` specifies that the guaranteed availability of GWDG is 95%.

`feature VM` specifies the attributes characterising the virtual machine flavours offered by the GWDG provider, such as type (`attribute VMType`), operating system (`attribute VMOS`), size of RAM (`attribute VMMemory`), size of storage (`attribute VMStorage`), and number of CPU cores (`attribute VMCores`). Each attribute has a value type, and a unit type. For instance, `VMMemory` has `MemoryList`, a list of integer values (256, 512, 2048, etc.), as value type, and `MEGABYTES` as unit type.

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

Listing 10: GWDG provider model (excerpt)

```
provider model GWDGProvider {

  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
          }

          attribute VMOS {
            value type: ScalarmModel.GWDGType.VMOsEnum
          }

          attribute VMMemory {
            unit type: MEGABYTES value type: ScalarmModel.GWDGType.
    MemoryList
          }

          attribute VMStorage {
            unit type: GIGABYTES value type: ScalarmModel.GWDGType.
    StorageList
          }

          attribute VMCores {
            value type: ScalarmModel.GWDGType.CoresList
          }
        }

        feature cardinality {
          cardinality: 1 .. 8
        }
      }

      feature Location {

        sub-features {

          feature Germany {

            feature cardinality {
              cardinality: 1 .. 1
            }
          }
        }

        feature cardinality {
          cardinality: 1 .. 1
        }
      }
    }

    feature cardinality {
      cardinality: 1 .. 1
    }
  }

  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
        }
      }
    }
...
  }
}

type model GWDGType {

  enumeration VMTypeEnum {
    values [ 'M1.MICRO' : 0,
...
    'M1.LARGE' : 4,
...
    'C1.XXLARGE' : 15 ]
  }

  enumeration VMOsEnum {
    values [ 'Fedora 20 server x86_64' : 0,
    'Ubuntu 14.04 LTS Server x86_64' : 1,
...
    ]
  }

  range MemoryRange {
    primitive type: IntType
    lower limit {
      int value 256 included
    }
    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 ]
  }

}
```

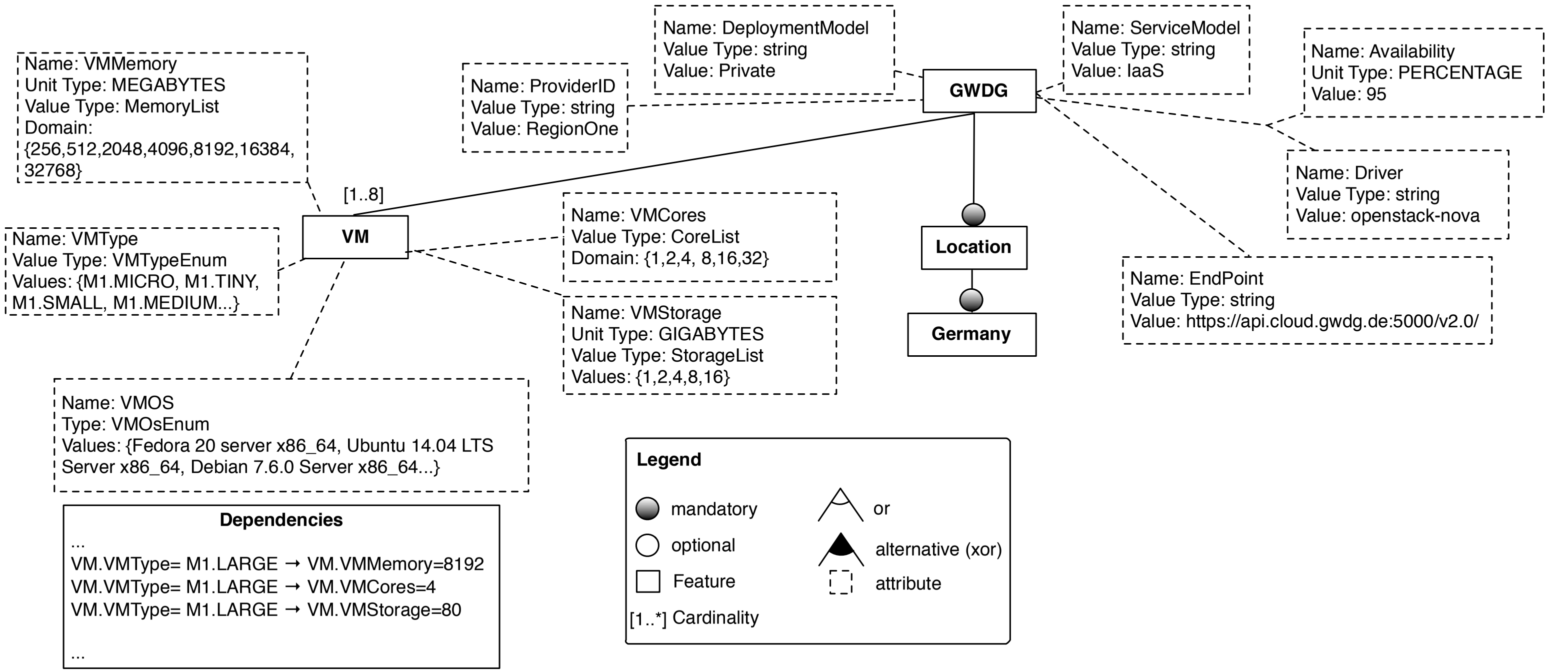

Figure 22: GWDG provider model diagram (Excerpt)

13 Organisations

The organisation package of the CAMEL metamodel is based on the organisation subset of CERIF [9]. CERIF is a modelling framework for specifying organisations, users and other entities in the research domain. It is an EU recommendation[15] for information systems related to research databases used for standardising research information and fostering research information exchange. In the following, we describe and exemplify the main concepts in the organisation package.

Figure 23 shows the class diagram of the organisation package.

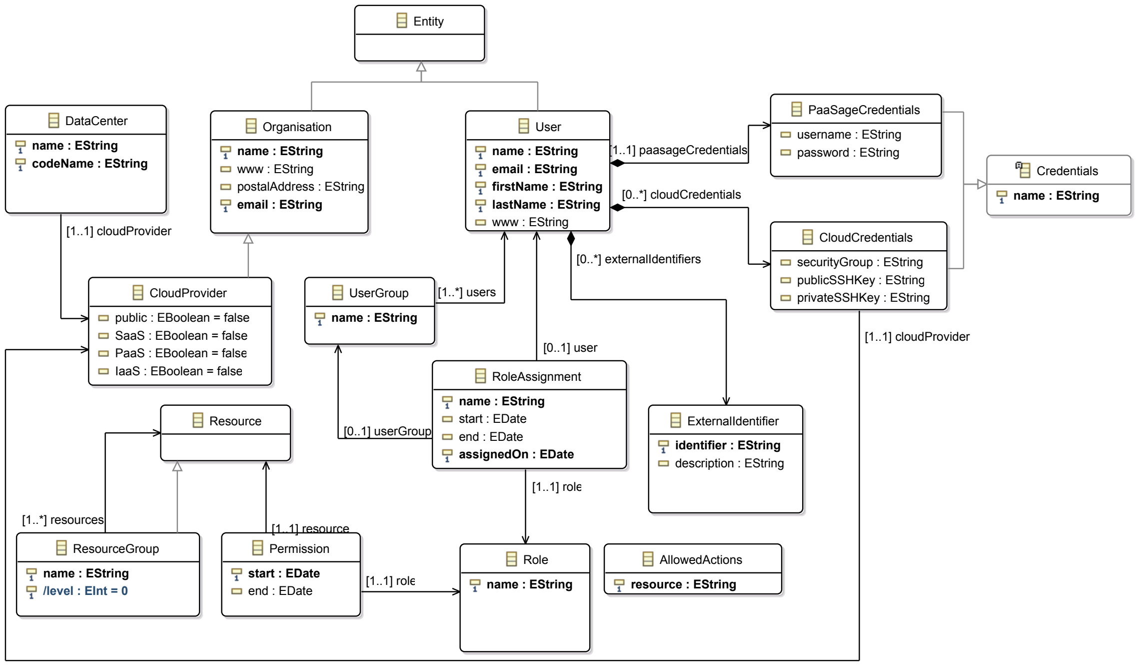

Figure 23: Class diagram of the organisation package

An Entity is a generic entity from CERIF. An entity can be an Organisation, which represents an organisation using the PaaSage platform. An organisation, in turn, can be a CloudProvider, which represents a cloud provider used by the PaaSage platform. The property public represents whether the cloud provider is public or not, while the properties PaaS, IaaS and SaaS represent whether the cloud provider supports the corresponding service models or not. A DataCenter represents a data centre of a cloud provider. The property codeName represents the internal code name used to identify the data centre at the cloud provider (*e.g.*, Amazon EC2 `eu-west-1`).

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

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

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

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

AllowedActions represents the set of all possible actions allowed to be performed on a resource. This way, when a permission is specified, the actions of the permission are checked against the allowed actions.

13.1 Example

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

`organisation AGH` specifies the AGH (Akademia Górniczo-Hutnicza, *i.e.*, AGH University of Science and Technology in Poland) organisation and the `user MichalOrzechowski` specifies a user (Michal Orzechowski). `paasage credentials MorzechCredentials` specifies the username and password for the user.

Listing 11: Scalarm organisation model

```
organisation model AGHOrganisation {

  organisation AGH {
    www: 'http://www.agh.edu.pl/en/'
    postal address: 'al. Mickiewicza 30, 30-059 Krakow, Poland'
    email: 'morzech@agh.edu.pl'
  }

  user MichalOrzechowski {
    first name: Michal
```

```
    last name: Orzechowski
    email: 'morzech@agh.edu.pl'

    paasage credentials MorzechCredentials {
      username: morzech
      password: '************'
    }
  }
}
```

14 Security

Note: Currently, the `security` package is not supported by the PaaSage platform. The security features of the PaaSage platform will be investigated during the last year of the project.

The security package of the CAMEL metamodel, which is based on the initial MDDB schema designed in WP4 (*cf.* D4.1.1 [13] and D4.1.2 [14]), provides the concepts to specify security aspects. It aims at enabling: (i) the specification of security requirements that can be used for filtering filter cloud providers; and (ii) the specification of security levels in SLOs, metrics, and scalability rules that can be used for adapting a cloud-based application in case of violations of the specified security levels.

Figure 24 shows the class diagram of the security package.

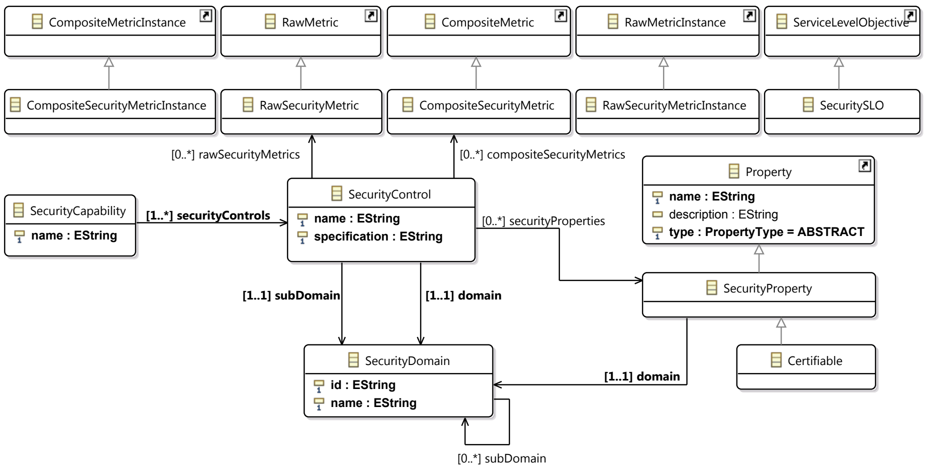

Figure 24: Class diagram of the security package

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

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

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

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

14.1 Example

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

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

`property IdentityAssurance` specifies an abstract security property associated with the security domain IAM. `security control IAM_02` specifies a security control associated with the security sub-domain (CLCPM) and the property `IdentityAssurance`. Similarly, `security control IAM_11` specifies a security control associated with the security sub-domain (UAR) and the property `IdentityAssurance`. Note that these security controls are part of the set of security controls identified by the Cloud Security Alliance (CSA)[16].

`security capability SecCap` specifies a security capability associated with the security controls `IAM_02` and `IAM_11`.

Finally, the `organisation model AmazonExt` refers to the security capability `SecCap`, which specifies that the Amazon provider supports this security capability.

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

```
security model ScalarmSecurity {

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

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

  domain IAM_UAR {
    name: "User Access Revocation"
  }

```

[16] https://cloudsecurityalliance.org/

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

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

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

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

requirement model ScalarmExtendedReqModel {

  security requirement AllIAMsSupported {
    controls [ScalarmSecurity.IAM_02, ScalarmSecurity.IAM_11]
  }
}

organisation model AmazonExt {

  provider Amazon {
    www: "www.amazon.com"
    email: "contact@amazon.com"
    PaaS
    IaaS
```

```
    security capability [ScalarmModel.ScalarmSecurity.SecCap]
  }
}

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

15 Execution

Note: This section is for PaaSage developers only. The execution model is automatically manipulated by the PaaSage platform during the execution phase (*cf.* Section 2).

The execution package of the CAMEL metamodel, which is based on the initial MDDB schema designed in WP4 (*cf.* D4.1.1 [13] and D4.1.2 [14]), provides the concepts to record historical data about the application execution, such as the measurements produced and the SLO assessments performed. This historical data allows the PaaSage platform to continuously optimise the CAMEL model to better exploit the cloud infrastructures. In the following, we describe and exemplify the main concepts in the execution package.

Figure 25 shows the class diagram of the execution package.

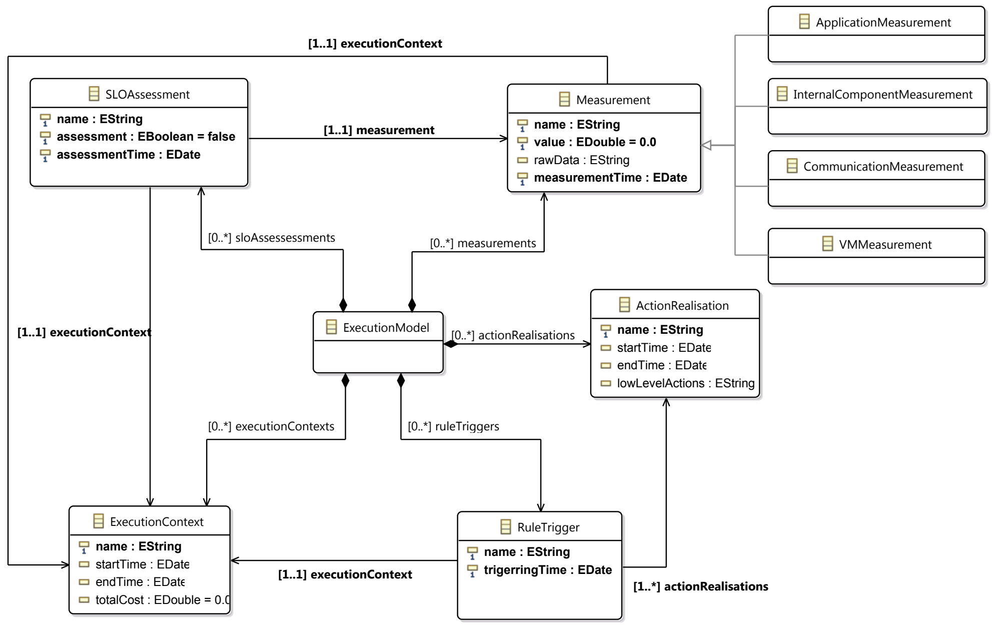

Figure 25: Class diagram of the execution package

An ExecutionContext represents the execution context for a particular deployment model. The properties startTime and endTime represent the date when the application execution started and ended, respectively. The property totalCost represents the total cost for the application execution, calculated when the application execution ended.

A Measurement represents a measurement produced during application execution. It refers to a `Metric` (*cf.* Section 10) and an execution context. The property value represents the value of the measurement. The property rawData represents the raw data in the time series database (TSDB) (*cf.* D5.1.2 [4]). The property measurementTime represents the timestamp of the measurement. A Measurement can be an ApplicationMeasurement, an InternalComponentMeasurement, a CommunciationMeasurement, or a VMMeasurement, depending on which element of a deployment model it refers to.

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

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

15.1 Example

Assume that we have to record the execution of the Scalarm use case. Listing 13 shows this specification in textual syntax for illustrative purposes. Note that the execution model must otherwise be manipulated through the Java API (*cf.* Section 18).

`vm binding ScalarmVMBinding` specifies the context for metric instances. It comprises a reference to the execution context and the virtual machine instance for which metric instances are produced (*cf.* Listing 5).

`execution context EC1` specifies the current execution context. It comprises a reference to the application being executed, a reference to the deployment model of the application, a reference to the requirement group that lead to this deployment model, and an indication of the total cost of application execution along with a reference to the corresponding monetary unit (*cf.* Listing 15)

`vm measurement VM1` specifies the virtual machine measurement for the CPU metric instance. It comprises a reference to the execution context, a reference to the metric instance, a reference to the virtual machine instance (*cf.* Listing 5), the measured value (`95.0`), and the timestamp of the measurement.

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

Listing 13: Sclarm execution model

```
metric model ScalarmMetric {

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

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

execution model ScalarmExecution {

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

  vm measurement VM1 {
    execution context: ScalarmExecution.EC1
    metric instance: ScalarmMetric.RawCPUMetricInstance
    vm instance: ScalarmModel.ScalarmDeployment.CoreIntensiveUbuntuGermanyInst
    value: 95.0
    time: 2014-12-10
  }

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

16 Types

The type package of the CAMEL metamodel is also based on Saloon [22, 23, 24]. It provides the concepts to specify value types and values used across CAMEL models (*e.g.*, integer, string, enumeration, etc.). In the following, we describe and exemplify the main concepts in the type package.

Figure 26 shows the class diagram of the type package.

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

ValueType represents a generic value type. It can be specialised into a StringValueType, BooleanValueType, Enumeration, List, Range and RangeUnion. StringValueType and BooleanValueType represent string and boolean value types, respectively. Enumeration represents an enumeration type that can take EnumerateValues. List represents a list type that can take either basic value type (*i.e.*, a numeric, string, or boolean value) or complex value type (*e.g.*, an enumeration or a range). The property `primitiveType` represents the basic value type, and it has to be used in the first case. The reference `type` represents the complex value type, and it has to be used in the second case. A Range represents a range-based value type. It has two references `lowerLimit` and `upperLimit` to Limit. A limit represent the lower and upper limits for the value type of the range. The property `included` represents whether the lower and upper limits are included in the range or not. The RangeUnion represents a union of range-based value types. It refers to the contained range-based value types as well as to the primitive type that is common across all the contained range-based value types (*e.g.*, all range-based value types are integer-based).

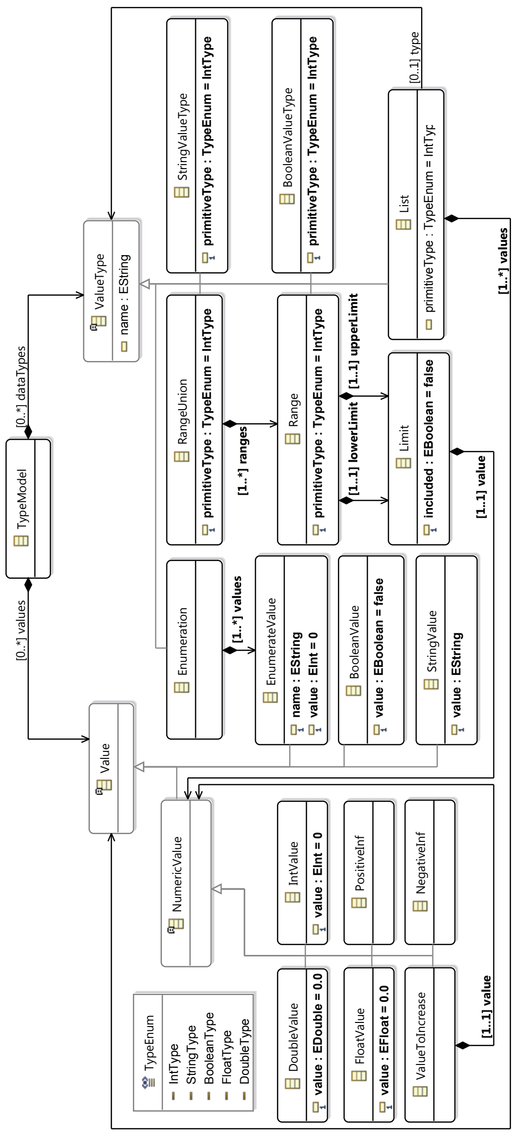

Figure 26: Class diagram of the type package

16.1 Example

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

The `range` statements specify two integer-based ranges and one double-based range. The first range is used to specify `CPUMetric` (*cf.* Listing 9 to represent that CPU metric values should be between 0 and 100, both included). The second range is used to specify the `ResponseTimeMetric` to represent that the response time values should be between 0, not included (*i.e.*, between 1), and 10000, included. The third range is used to specify the `AvailabilityMetric` to represent that the availability should be between 0.0 and 100.0, both included.

Listing 14: Scalarm type model

```
type model ScalarmType {

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

  range Range_0_10000 {
    primitive type: IntType
    lower limit {int value 0}
    upper limit {int value 1000 included}
  }

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

17 Units

The unit package of the CAMEL metamodel provides the concepts to specify units used across CAMEL models. These concepts are adopted by the following packages: (a) metric, where they are used to define the unit of measurement for a metric, (b) execution, where they are used to define the monetary unit for the cost of a particular application execution, and (c) the provider, where they are used to define the unit for a particular feature attribute. In the following, we describe and exemplify the main concepts in the unit package.

Figure 27 shows the class diagram of the unit package.

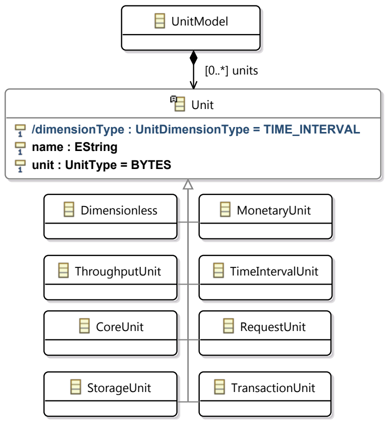

Figure 27: Class diagram of the unit package

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

- CoreUnit, which represents the unit of CPU cores
- MonetaryUnit, which represents a monetary unit (*e.g.*, EUROS)
- RequestUnit, which represents the unit of number requests
- StorageUnit, which represents the unit of storage (*e.g.*, BYTES)
- ThroughputUnit, which represents the unit of throughput (*e.g.*, REQUESTS_PER_SECOND)
- TimeIntervalUnit, which represents the unit of time interval (*e.g.*, SECONDS)
- TransactionUnit, which represents the number of transactions

- Dimensionless, which represents a unit without dimension (*e.g.*, a unit of PERCENTAGE is dimensionless)

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

Figure 28 shows the class diagram of the enumerations in the unit package.

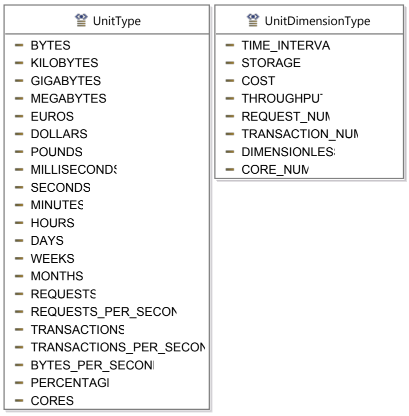

Figure 28: The class diagram of the enumerations in the unit package

17.1 Example

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

The unit model encompasses seven units that are used in the metric model (*cf.* Listing 9. The specification of each unit follows the pattern: <unit_class> <unit_name>: <unit_type>. For instance, monetary unit {Euro: EUROS} specifies a monetary unit named "euros" and typed EUROS).

Listing 15: Scalarm unit model

```
unit model ScalarmUnit {

  monetary unit {Euro: EUROS}

  throughput unit {SimulationsPerSecondUnit: TRANSACTIONS_PER_SECOND}

  time interval unit {ResponseTimeUnit: MILLISECONDS}

  time interval unit {ExperimentMakespanInSecondsUnit: SECONDS}

  transaction unit {NumberOfSimulationsLeftInExperimentUnit:
    TRANSACTIONS}
```

```

  dimensionless {AvailabilityUnit: PERCENTAGE}

  dimensionless {CPUUnit: PERCENTAGE}
}
```

18 Java APIs and CDO

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

Listing 16 shows the creation of a `VM` of SENSAPP (*cf.* Section 2). The classes that are instantiated and initialised in the code have been automatically generated by the EMF generator model based on the deployment package. All class instances are obtained using the `DeploymentFactory` object specific for the deployment package. This object provides a set of methods that are used to make sure that the model objects are appropriately instantiated.

Listing 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 of a SensApp `InternalComponent`. It has an associated `Configuration`, which specifies the life cycle control scripts (*e.g.*, download and install commands) for the internal component. The SensApp application must be hosted on a servlet container, which is specified through the `RequiredHostingPort`. Furthermore, it will need to communicate with another component (MongoDB), and will provide a web interface on port 8080. This information is expressed in the `RequiredCommunication` and `ProvidedCommunication` objects specification respectively.

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 how to establish a `Communication` binding between the SensApp and the SensApp Admin `InternalComponents`.

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 19 shows the specification of a `Hosting` binding between the Jetty `InternalComponent` and the SL `VM`.

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 that are designed to resemble 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[17], a partner project used within CDO. Once the transaction is obtained, the deployment model can be stored in a `CDOResource` and then persisted as the transaction is committed.

Listing 20: Saving a deployment model in a CDO repository

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

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

```
Net4jUtil.prepareContainer(container);
TCPUtil.prepareContainer(container);
CDONet4jUtil.prepareContainer(container);
container.activate();

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

// create the session configuration
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 we change where one of the internal components is hosted (*i.e.* the Jetty servlet contained). We move the hosting from an ML to an SL virtual machine.

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 vmJettySCReq = null;
for (InternalComponent ic: model.getInternalComponents()){
  if (ic.getName().equals("JettySC")){
      vmJettySCReq = ic.getRequiredHost();
        break;
    }
}
ProvidedHost vmSLProv = null;
for(VM vm: model.getVMs()){
```

```
  if (ic.getName().equals("SL")){
      vmSLProv = vm.getProvidedHost();
        break;
    }
}

//find previous hosting and remove it from deployment model
Hosting oldHosting = null;
for (Hosting h2: model.getHostings()){
  if (h2.getName().equals("JettySCToVMML")){
      oldHosting = h2;
        break;
    }
}
model.getHostings().remove(oldHosting);

//create new hosting and add it to the deployment model
Hosting jettySCToVMSL = DeploymentFactory.eINSTANCE.createHosting();
jettySCToVMSL.setName("JettySCToVMSL");
jettySCToVMSL.setRequiredHost(vmJettySCReq);
jettySCToVMSL.setProvidedHost(vmSLProv);
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. The full version of the Java code of the SensApp example is available for reference at: `https://tuleap.ow2.org/plugins/git/paasage/cdo_client?p=cdo_client.git&a=blob&f=src/eu/paasage/camel/examples/SensAppCDO.java`.

19 Related Work

In the cloud community, libraries such as jclouds[18] or DeltaCloud[19] 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[20], Puppet[21], or Chef[22] 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 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, SINTEF presented the models@run-time approach to the TOSCA technical committee (TC) and proposed to form an ad hoc group to investigate how TOSCA could be extended to support this approach. The TC welcomed this proposal and, 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.

[18] `http://www.jclouds.org`
[19] `http://deltacloud.apache.org/`
[20] `http://www.cloudifysource.org/`
[21] `https://puppetlabs.com/`
[22] `http://www.opscode.com/chef/`

20 Conclusion and Future Work

In this document, we have provided the final version of the CAMEL documentation. 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 will continue developing CAMEL iteratively. In particular, we will adapt and extend the capabilities of CAMEL to the changing requirements. In this respect, the developers will provide feedback on whether the concepts in CAMEL are adequate to design and implement their components (either within or outside the PaaSage platform). Similarly, the users will provide feedback on whether the concepts in CAMEL are satisfactory for modelling the use cases.

In addition, CAMEL models that conform to an old version of CAMEL often have to be migrated to conform to its current version. In the future, we would like to integrate a solution to the challenge of maintaining multiple versions and automatically migrating CAMEL models [17] based on CDO and Edapt.

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

Acknowledgements The authors would like to thank the use case providers and component developers for the constructive feedback on CAMEL.

References

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

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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`.

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

[11] 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`.

[12] 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`.

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

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

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

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

[17] Nikolay Nikolov, Alessandro Rossini and Kyriakos Kritikos. "Integration of DSLs and Migration of Models: A Case Study in the Cloud Computing Domain". In: *Cloud Forward 2015 Conference*. Elsevier, 2015 (To Appear).

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

[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.