PaaSage

Model Based Cloud Platform Upperware

Deliverable D2.1.3

CAMEL Documentation

Version: 1.1
Name, title and organisation of the scientific representative of the project’s coordinator:
Mr Philippe Rohou Tel: +33 4 9238 5306 Fax: +33 4 9238 7822 E-mail: philippe.rohou@ercim.eu
Project website address: http://www.paasage.eu

<table>
<thead>
<tr>
<th>Project</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Grant Agreement number</td>
<td>317715</td>
</tr>
<tr>
<td>Project acronym:</td>
<td>Paasage</td>
</tr>
<tr>
<td>Project title:</td>
<td>Model Based Cloud Platform Upperware</td>
</tr>
<tr>
<td>Funding Scheme:</td>
<td>Integrated Project</td>
</tr>
<tr>
<td>Date of latest version of Annex I against which the assessment will be made:</td>
<td>30th September 2015</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Document</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Period covered:</td>
<td></td>
</tr>
<tr>
<td>Deliverable number:</td>
<td>D2.1.3</td>
</tr>
<tr>
<td>Deliverable title</td>
<td>CAMEL Documentation</td>
</tr>
<tr>
<td>Contractual Date of Delivery:</td>
<td>30th September 2015 (M36)</td>
</tr>
<tr>
<td>Actual Date of Delivery:</td>
<td>16th October 2015</td>
</tr>
<tr>
<td>Editor (s):</td>
<td>Alessandro Rossini</td>
</tr>
<tr>
<td>Author (s):</td>
<td>Alessandro Rossini, Kiriakos Kritikos, Nikolay Nikolov, Jörg Domaschka, Frank Griesinger, Daniel Seybold, Daniel Romero</td>
</tr>
<tr>
<td>Reviewer (s):</td>
<td>Jörg Domaschka, Achilleas Achilleos</td>
</tr>
<tr>
<td>Participant(s):</td>
<td></td>
</tr>
<tr>
<td>Work package no.:</td>
<td>2</td>
</tr>
<tr>
<td>Work package title:</td>
<td>Languages</td>
</tr>
<tr>
<td>Work package leader:</td>
<td>Alessandro Rossini</td>
</tr>
<tr>
<td>Distribution:</td>
<td></td>
</tr>
<tr>
<td>Version/Revision:</td>
<td>1.1</td>
</tr>
<tr>
<td>Draft/Final:</td>
<td>Final</td>
</tr>
<tr>
<td>Total number of pages (including cover):</td>
<td>82</td>
</tr>
</tbody>
</table>
Disclaimer

This document contains description of the PaaSage project work and findings.

The authors of this document have taken any available measure in order for its content to be accurate, consistent and lawful. However, neither the project consortium as a whole nor the individual partners that implicitly or explicitly participated in the creation and publication of this document hold any responsibility for actions that might occur as a result of using its content.

This publication has been produced with the assistance of the European Union. The content of this publication is the sole responsibility of the PaaSage consortium and can in no way be taken to reflect the views of the European Union.

The European Union is established in accordance with the Treaty on European Union (Maastricht). There are currently 28 Member States of the Union. It is based on the European Communities and the member states cooperation in the fields of Common Foreign and Security Policy and Justice and Home Affairs. The five main institutions of the European Union are the European Parliament, the Council of Ministers, the European Commission, the Court of Justice and the Court of Auditors. ([http://europa.eu](http://europa.eu))

PaaSage is a project funded in part by the European Union.
Executive Summary

Cloud computing provides ubiquitous networked access to a shared and virtualised pool of computing capabilities that can be provisioned with minimal management effort [25]. Cloud applications are deployed on cloud infrastructures and delivered as services. The PaaSage project aims to facilitate the modelling and execution of cloud applications by leveraging model-driven engineering (MDE) and by exploiting multiple cloud infrastructures. 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. The Cloud Application Modelling and Execution Language (CAMEL) is the core modelling and execution language developed in the PaaSage project and enables the specification of multiple aspects of cross-cloud applications (i.e., applications deployed across multiple private, public, or hybrid cloud infrastructures). By exploiting models at both design- and run-time, and by allowing both direct and programmatic manipulation of models, CAMEL enables the management of self-adaptive cross-cloud applications (i.e., cross-cloud applications that autonomously adapt to changes in the environment, requirements, and usage). In this document, we describe the design and implementation of CAMEL. Moreover, we provide a real-world running example to illustrate how to specify models in a concrete textual syntax and how to programatically manipulate and persist them through Java APIs. Finally, we describe the abstract syntax of the language.
Contents

1 Introduction 7

2 CAMEL and the Self-Adaptation Workflow 8

3 Technologies 11

4 Installation and Usage of CAMEL Textual Editor 12
   4.1 Installation—Users 13
   4.2 Installation—Developers 13
   4.3 Usage 14

5 CAMEL Design and Syntax 14

6 Deployment 17
   6.1 Interplay with Executionware 21
      6.1.1 Life Cycle Scripts for Unix-based Applications 21
      6.1.2 Life Cycle Scripts for Windows-based Applications 24

7 Requirements 26

8 Metrics and Scalability Rules 28
   8.1 Interplay with Executionware 32

9 Organisations 32

10 Providers 33

11 Security 36

12 Execution 38

13 Locations 40

14 Units 40

15 Types 41

16 Java APIs and CDO 43
1 Introduction

MDE is a branch of software engineering that aims to improve the productivity, quality, and cost-effectiveness of software development by shifting the paradigm from code-centric to model-centric. MDE promotes the use of models and model transformations as the primary assets in software development, where they are used to specify, simulate, generate, and manage software systems. This approach is particularly relevant when it comes to the modelling and execution of cross-cloud applications (i.e., applications deployed across multiple private, public, or hybrid cloud infrastructures). This solution allows to exploit the peculiarities of each cloud service and hence to optimise the performance, availability, and cost of the applications.

Models can be specified using general-purpose languages like the Unified Modeling Language (UML) [29]. However, to fully unfold the potential of MDE, models are frequently specified using domain-specific languages (DSLs), which are tailored to a specific domain of concern. The PaaSage\(^1\) project exploits the latter approach and provides an integrated platform to support the modelling and execution of cross-cloud applications. To achieve this goal, PaaSage developed the Cloud Application Modelling and Execution Language (CAMEL). This DSL allows to specify multiple aspects of cross-cloud applications, such as provisioning, deployment, service level, monitoring, scalability, providers, organisations, users, roles, security, and execution.

CAMEL supports the models@run-time [5] approach, which provides an abstract representation of the underlying running system, whereby a modification to the model is enacted on-demand in the system, and a change in the system is automatically reflected in the model.

\(^1\)http://www.paasage.eu

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

CAMEL was designed and implemented to allow the design-time specification of models by users as well as their run-time manipulation by reasoners. By exploiting models at both design- and run-time, and by allowing both direct and programmatic manipulation of models, CAMEL enables the management of self-adaptive cross-cloud applications (i.e., cross-cloud applications that autonomously adapt to changes in the environment, requirements, and usage). This represents the main motivation for our research and contribution of our work, since, to the best of our knowledge, no other integrated language in the literature supports the management of self-adaptive cross-cloud applications (see Section 17).

Structure of the document: The remainder of the document is organised as follows. Section 2 describes the role of CAMEL models in a self-adaptation workflow. Section 3 presents some technologies used to design and implement CAMEL. Section 4 provide instructions for installing and using the CAMEL Textual Editor. Sections 5–15 present the various packages of the CAMEL metamodel along with corresponding sample models in concrete syntax. Section 16 exemplifies the usage of Java APIs to programmatically manipulate and persist models. Section 17 compares the proposed approach with related work, while Section 18 draws conclusions and outlines plans for future work. Finally, Appendix A presents the abstract syntax of CAMEL.

2 CAMEL and the Self-Adaptation Workflow

The components managing the life cycle of cross-cloud applications are integrated by leveraging CAMEL models. These models are progressively refined throughout the modelling, deployment, and execution phases of a self-adaptation workflow based on the models@run-time approach, as proposed in PaaSage [35].

Figure 2 shows the self-adaptation workflow. The white trapezes represent the activities performed by the user. The white rectangles represent the processes executed by the PaaSage platform. The coloured shapes represent the modelling artefacts, whereby the blue ones pertain to the modelling phase, the red ones to the deployment phase, and the green ones to the execution phase.
In the remainder of the document, we adopt the Scalarm\(^2\) [23] use case as a real-world running example to illustrate how to specify multiple aspects of cloud applications in CAMEL, and how these specifications facilitate the deployment of cloud applications across multiple clouds and their self-adaptation to changes in the environment, requirements, and usage. Scalarm is a massively self-scalable platform for data farming. Data farming experiments utilise high-performance and high-throughput computing to generate large amounts of data via simulations. These data are analysed to obtain new insights into the studied phenomena. The architecture of Scalarm is based on the principles of service-oriented architecture (SOA) and consists of the following services:

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

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

**Modelling phase.** The users design a **cloud provider-independent model (CPIM)**, which specifies the deployment of a cross-cloud application along with its re-

\(^2\)http://www.scalarm.com/
requirements and objectives (e.g., on virtual hardware, location, and service level) in a cloud provider-independent way.

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

![Diagram of CPIM and CPSM models](image)

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

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

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

---

\(^3\)https://www.sintef.no/en/
\(^4\)https://www.cloudservices.no/
Figure 3(b) shows the CPSM in graphical syntax. It consists of an Experiment Manager instance, which is hosted on a Ubuntu 14.04 instance at Amazon EC2 in the EU. Moreover, the Experiment Manager instance communicates with two Simulation Manager instances, which are hosted on two CentOS 7 virtual machine instances at SINTEF in Norway.

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

### 3 Technologies

In order to design and implement CAMEL, we adopted the Eclipse Modeling Framework (EMF)\(^5\) along with Object Constraint Language (OCL) [28], Xtext\(^6\), and Connected Data Objects (CDO).\(^7\) In the following, we outline these technologies and describe how they facilitate the implementation of the PaaSage platform described in Section 2.

**Eclipse Modeling Framework (EMF).** In MDE, the abstract syntax of a DSL is typically defined by its metamodel, which describes the set of concepts, their attributes, and their relations, as well as the rules for combining these concepts to specify valid models conforming to this metamodel [29]. EMF is a modelling framework that facilitates defining DSLs. It provides the Ecore metamodel, which is an Ecore model that conforms to itself (i.e., it is reflexive). The CAMEL metamodel is an Ecore model that conforms to the Ecore metamodel.

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

**Object Constraint Language (OCL).** EMF enables to check the cardinality constraints on properties and to validate models against their metamodels. However, it lacks the expressiveness required to capture all of the semantics of the domain. OCL is a declarative language for specifying expressions on metamodels.

---

\(^5\)https://www.eclipse.org/modeling/emf/
\(^6\)https://eclipse.org/Xtext/
\(^7\)https://www.eclipse.org/cdo/
that are evaluated on models conforming to these metamodels. Eclipse OCL\(^8\) is a tool-supported implementation of OCL that integrates with EMF. The CAMEL metamodel is annotated with OCL expressions to capture part of the semantics of cross-cloud applications and to guarantee the consistency, correctness, and integrity of CAMEL models at both design-time and run-time.

**Xtext.** In MDE, the concrete syntax describes the textual or graphical notation that renders the concepts, attributes, and relations in the abstract syntax. The concrete syntax may vary depending on the domain, e.g., a DSL could provide a textual notation as well as a graphical notation along with the corresponding serialisation in XML Metadata Interchange (XMI) [30]. Xtext is a language development framework that is based on- and integrates with EMF. It facilitates the implementation of Eclipse-based IDEs providing features, such as syntax highlighting, code completion, code formatting, static analysis, and serialisation. The concrete syntax of CAMEL is a textual syntax defined as an Xtext grammar. The textual syntax was preferred over the graphical syntax by the early adopters of CAMEL.

**Connected Data Objects (CDO).** CDO is semi-automated persistence framework that works natively with Ecore models and their instances. It provides a model repository where clients can persist, share, and query their models. It provides features that satisfy the design-time and run-time requirements of the self-adaptation workflow (see Section 2), such as abstraction from database management systems (DBMSs), validation, transactional processing, optimistic versioning [38], automatic notification, auditing, and role-based security [21].

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

### 4 Installation and Usage of CAMEL Textual Editor

In this section, we provide instructions for installing and using the CAMEL Textual Editor. These steps have been tested with the latest version of Eclipse, which

\(^8\)http://wiki.eclipse.org/OCL
at the time of writing is Eclipse Neon v4.6.0. We distinguish between the installation for users and for developers.

### 4.1 Installation—Users

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

### 4.2 Installation—Developers

- Clone the CAMEL Git repository from:  
  https://tuleap.ow2.org/plugins/git/paasage/camel
- Download and install “Eclipse IDE for Java and DSL Developers” from:  
  https://www.eclipse.org/downloads/
- Start Eclipse
- Select Help > Install New Software…
- Select Work with: Neon—http://download.eclipse.org/releases/neon
- Select Modeling > CDO Model Repository SDK
• Select Modeling > OCL Classic SDK: Ecore/UML Parsers,Evaluator,Edit
• Select Modeling > OCL Examples and Editors
• Install the three packages
• Restart Eclipse
• Select Import > Existing Projects into Workspace
• Select Browse...
• Select the folder where you cloned the CAMEL Git repository
• Select Finish
• Select eu.paasage.camel.dsl/src/eu.paasage.camel.dsl/GenerateCamelDsl.mwe2
• Select Run As > MWE2 Workflow...
• Select eu.paasage.camel.dsl
• Select Run > Run As > Eclipse Application...

4.3 Usage

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

5 CAMEL Design and Syntax

CAMEL has been designed based on the following requirements, among others:
• **Cloud provider-independence** (*R_1*): CAMEL should support a cloud provider-agnostic specification of multiple aspects of cross-cloud applications (i.e., provisioning, deployment, service level, monitoring, scalability, providers, organisations, users, roles, security, and execution). This will prevent vendor lock-in.

• **Separation of concerns** (*R_2*): CAMEL should support loosely-coupled packages (or modules) corresponding to multiple aspects of cross-cloud applications. This will facilitate the development of models.

• **Reusability** (*R_3*): CAMEL should support reusable types for multiple aspects of cross-cloud applications. This will ease the evolution of models.

• **Abstraction** (*R_4*): CAMEL should provide an up-to-date, abstract representation of the running system. This will enable the reasoning, simulation, and validation of the adaptation actions before their enactment.

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

In addition, CAMEL implements the *type-instance* pattern [1], which also facilitates reusability (*R_3*) and abstraction (*R_4*). This pattern exploits two flavours of typing, namely *ontological* and *linguistic* [24]. Figure 4 illustrates these two flavours of typing. SL (short for Small GNU/Linux) represents a reusable type of virtual machine. It is linguistically typed by the class **VM** (short for virtual machine). SL1 represents an instance of the virtual machine SL. It is ontologically typed by SL and linguistically typed by VMInstance.

![Figure 4: Linguistic and ontological typing](image)

CAMEL and the CAMEL Textual Editor are available under Mozilla Public License 2.0[^9] in the Git repository at OW2[^10]. The CAMEL metamodel is an

[^9]: https://www.mozilla.org/en-US/MPL/2.0/
[^10]: https://tuleap.ow2.org/plugins/git/paasage/camel
Ecore model organised into packages, whereby each package reflects an aspect (or domain).

As mentioned, in the remainder of the document, we adopt the Scalarm\textsuperscript{11} [23] use case as a real-world running example to illustrate how to specify CAMEL models in textual syntax. The complete Scalarm CAMEL model in textual syntax can be downloaded from the Git repository at OW2.\textsuperscript{12} This running example covers the most commonly used concepts in CAMEL. The interested reader may refer to [37] for a complete description of the abstract syntax of CAMEL.

Listing 5.1: Scalarm sample application

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

    organisation model AGHOrganisation {
        ...
    }

    user morzech {
        ...
    }

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

Listing 5.1 shows an excerpt of the CAMEL model of Scalarm in textual syntax. An element of a CAMEL model is specified by the name of the element followed by an identifier in CamelCase and a block delimited by curly brackets. camel model ScalarmModel {...} specifies the CAMEL model of Scalarm, where ScalarmModel represents the identifier of this model. application ScalarmApplication specifies the Scalarm application itself.

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

A reference to a single element is specified by the name of the reference followed by a colon and a fully qualified name conforming to the pattern: \(id_1.id_2.\ldots.id_n\), where \(id_i\) refers to element at the \(i\)th level of the containment path and \(id_n\) refers to the element at the leaf level, which is the element under discussion. owner: AGHOrganisation.morzech specifies that the application

\textsuperscript{11}http://www.scalarm.com/
\textsuperscript{12}https://tuleap.ow2.org/plugins/git/paasage/camel?p=camel.git&a=blob&f=examples/Scalarm.camel
is owned by the user Michal Orzechowski by referring to the user morzech in the organisation model AGHOrganisation (see Section 9, Listing 9.1).

Finally, a reference to a list of elements is specified by the name of the reference followed by a comma separated list of fully qualified names delimited by square brackets.\textsuperscript{13} deployment models [ScalarmModel.ScalarmDeployment] specifies that the Scalarm application has one deployment model by referring to the deployment model ScalarmDeployment (see Section 6, Listing 6.1).

\section{Deployment}

The deployment package of the CAMEL metamodel is based on CloudML\textsuperscript{14} [13, 11, 12], which was developed in collaboration with the MODAClouds project.\textsuperscript{15} CloudML consists of a tool-supported DSL for modelling and enacting the provisioning and deployment of cross-cloud applications, as well as for facilitating their dynamic adaptation, by leveraging MDE techniques and methods. In the following, we exemplify the main concepts in the deployment package.

Assume that we have to specify the Experiment Manager component of the Scalarm use case. Listing 6.1 shows this specification in textual syntax.

\begin{verbatim}
  internal component ExperimentManager
      specifies a reusable type of the component Experiment Manager.
      provided communication ExpManPort specifies that
      the Experiment Manager provides its features through port 443.
      required communication SimManPortReq specifies that the Experiment Manager requires features
      from the Simulation Manager through port 80 (cf. Listing 6.3 for the corresponding specification
      of the communication binding). The property mandatory of SimManPortReq specifies that the Experiment Manager depends on the features
      of the Simulation Manager, and hence, that the Simulation Manager has to be deployed
      and started before the Experiment Manager. required host CoreIntensiveUbuntuNorwayReq
      specifies that the Experiment Manager requires a virtual machine that provides a large number of CPU cores, runs the operating system
      Ubuntu, and is located in Norway (cf. Listing 6.2 for the specification of the VM
      and VM requirements, and Listing 6.4 for the specification of the corresponding
      hosting binding).

      configuration ExperimentManagerConfiguration
      specifies the commands to handle the life cycle of the Experiment Manager.
      download, install, and start specify the OS-specific shell scripts (in this case, Bash scripts)
      for downloading, installing, and starting the Experiment Manager, respectively. Note that,
\end{verbatim}

\textsuperscript{13}Note that the colon is not used in this case.
\textsuperscript{14}http://cloudml.org
\textsuperscript{15}http://www.modaclouds.eu
although not shown in this example, it is also possible to specify the configure and stop commands of a component.

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

Listing 6.1: Scalarm sample internal component

deployment model ScalarmDeployment {
  internal component ExperimentManager {
    provided communication ExpManPort {port: 443}
    required communication SimManPortReq {port: 80 mandatory}
    required host CoreIntensiveUbuntuNorwayHostReq
    configuration ExperimentManagerConfiguration {
      chmod +x scalarm_exp_man.sh'
      install: './scalarm_exp_man.sh install'
      start: './scalarm_exp_man.sh start'
    }
  }
  ...
}

Then, assume that we have to specify the virtual machine on which the Experiment Manager can be deployed. Listing 6.2 shows this specification in textual syntax.

vm CoreIntensiveUbuntuNorway specifies a reusable type for a virtual machine. requirement set refers to the aforementioned requirement set CoreIntensiveUbuntuNorwayRS. provided host CoreIntensiveUbuntuNorwayPort specifies that the virtual machine provides a large number of CPU cores, runs the operating system Ubuntu, and is located in Norway (cf. Listing 6.4 for the specification of the corresponding hosting binding).

requirement set CoreIntensiveUbuntuNorwayRS specifies a reusable set of requirements for a virtual machine. quantitative hardware, os, and location refer to the requirements CoreIntensive, Ubuntu, and NorwayReq, respectively, in the requirement model ScalarmRequirement (cf. Listing 7.1), which in turn specify the hardware requirements encompassing a large number of CPU cores, the operating system requirement of Ubuntu, and the location requirement of Norway, respectively.

Listing 6.2: Scalarm sample vm

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

requirement set CoreIntensiveUbuntuNorwayRS {
  quantitative hardware: ScalarmRequirement.CoreIntensive
}
Next, assume that we have to specify the communication binding between
the Experiment Manager and the Simulation Manager. Listing 6.3 shows this
specification in textual syntax.

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

Finally, assume that we have to specify the hosting binding between the Ex-
periment Manager and the virtual machine CoreIntensiveUbuntuNorway. List-
ing 6.4 shows this specification in textual syntax.

```plaintext
hosting ExperimentManagerToCoreIntensiveUbuntuNorway {
  from ExperimentManager.CoreIntensiveUbuntuNorwayPortReq to CoreIntensiveUbuntuNorway.CoreIntensiveUbuntuNorwayPort
}
```
The types presented above can be instantiated in order to form a CPSM. In PaaSage, the instances within the deployment model are automatically manipulated during the deployment phase (see Section 2). In the general case, the instances could also be manipulated manually. Note that different CPSMs can adopt different instantiation patterns for communications and hosting bindings, while still conforming to the same CPIM. The interested reader may refer to [7] for an extensive discussion on the subject.

Listing 6.5 shows the specification of instances of the components, virtual machines, communications, and hosting bindings from the previous examples (cf. Listings 6.1, 6.2, 6.3, and 6.4) in textual syntax.

```
vm instance CoreIntensiveUbuntuNorwayInst specifies an instance of a virtual machine. vm type and vm type value refer to the virtual machine flavour M1.LARGE in the provider model SINTEFProvider (cf. Listing 10.1), which is compatible with the requirement set of the virtual machine template CoreIntensive-UbuntuNorway (cf. Listing 7).

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

Listing 6.5: Scalarm sample instances of internal component, vm, communication, and hosting

```
connect ExperimentManagerInst.SimManPortReqInst to SimulationManagerInst.SimManPortInst typed ScalarmModel. ScalarmDeployment.ExperimentManagerToSimulationManager


...
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\(^\text{16}\) in order to enable a maximum of isolation between the instances. This has a consequence for 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 besides `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 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 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

\(^{16}\text{http://docker.io} \)
on the name of the provided communication from the deployment model. For instance, the provided port \texttt{ExpManPort} from Listing 6.1 is mapped to the following three environment variables:

- \texttt{CONTAINER\_EXP\_MAN\_PORT}: the port number as specified in the deployment model and as accessible from within the container. Should be used for binding.
- \texttt{CLOUD\_EXP\_MAN\_PORT}: the port number as accessible from within the cloud.
- \texttt{PUBLIC\_EXP\_MAN\_PORT}: the port number as accessible from the outside world (\textit{i.e.}, by using the public IP).

**Outgoing Connections in the Execution Environment.** Similar to the provided communications, there is a mapping for required communications. The main difference is that it uses sets of IP addresses in combination with ports. For instance, the required port \texttt{StoManPortReq} from Listing 6.1 is mapped to the following three environment variables; all consisting of a sequence of \texttt{ipv4:port} separated by a comma (,).

- \texttt{PUBLIC\_STOMAN\_PORT\_REQ}: provides access to the public IP addresses and public ports of all downstream component instances.
  \texttt{<stoman1publicip>:<public_port>,<stoman2publicip>:<public_port>}

- \texttt{CLOUD\_STOMAN\_PORT\_REQ}: provides access to the cloud-internal IP addresses and cloud-internal ports of all downstream component instances. Note that the addresses of component instances that are not hosted in the same cloud as the local component instances are still in the list, but it is very likely that traffic cannot be routed to them.
  \texttt{<stoman1cloudip>:<cloud_port>,<stoman2cloudip>:<cloud_port>}

- \texttt{CONTAINER\_STOMAN\_PORT\_REQ}: provides access to the container-internal IP addresses and container-internal ports of all downstream component instances. Note that the addresses of component instances that are not hosted in the same container as the local component instances are still in the list, but it is very likely that traffic cannot be routed to them.
  \texttt{<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 the 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, 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.

6.1.2 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 component 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 6.1 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 6.1.1) except there is no need to map the communication to a container-internal IP. Again, consider Listing 6.1 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.

## 7 Requirements

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

Assume that we have to specify the requirements for the components of the Scalarm use case. Listing 7.1 shows this specification in textual syntax.

- **quantitative hardware** CoreIntensive specifies that a virtual machine must have from 8 to 32 CPU cores and from 4 to 8 GB of RAM. os Ubuntu specifies that a virtual machine must run Ubuntu operating system 64-bit edition.
- **location requirement** NorwayReq specifies that a virtual machine must be deployed in Norway. locations refers to the location NO in the location model ScalarmLocation (cf. Listing 13.1). The three requirements above are referred to by the requirement set CoreIntensiveUbuntuNorwayRS in the deployment model ScalarmDeployment (cf. Listing 6.2).
- **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 8.2).
- **optimisation requirement** MinimisePerformanceDegradationOfExperimentManager specifies that the metric MeanValueOfResponseTimeOfAllExperimentManagers-Metric of the component ExperimentManager should be minimised and that this
minimisation has a priority 0.8. metric refers to the metric MeanValueOfResponse-
TimeOfAllExperimentManagersMetric in the metric model ScalarmModel (cf. Listing 8.2), while component refers to the internal component ExperimentManager in the deployment model ScalarmDeployment (cf. Listing 6.2). optimisation requirement MinimiseDataFarmingExperimentMakespan specifies a similar optimisation requirement with priority 0.2.

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

The horizontal scale requirement HorizontalScaleSimulationManager specifies that the component SimulationManager can scale horizontally between 1 and 5 instances. component refers to the internal component SimulationManager in the deployment model ScalarmDeployment (cf. Listing 6.2).

Listing 7.1: Scalarm requirement model

```plaintext
requirement model ScalarmRequirement {
  quantitative hardware CoreIntensive {
    core: 8..32
    ram: 4096..8192
  }
  os Ubuntu {
    os: 'Ubuntu' 64os
  }
  location requirement NorwayReq {
    locations [ScalarmLocation.NO]
  }
  slo CPUMetricSLO {
    service level: ScalarmModel.ScalarmMetric.CPUMetricCondition
  }
  optimisation requirement MinimisePerformanceDegradationOfExperimentManager {
    function: MIN
    metric: ScalarmModel.ScalarmMetric.
    MeanValueOfResponseTimeOfAllExperimentManagersMetric
    component: ScalarmModel.ScalarmDeployment.ExperimentManager
    priority: 0.8
  }
  optimisation requirement MinimiseDataFarmingExperimentMakespan {
    function: MIN
  }
}
```
8 Metrics and Scalability Rules

The scalability and metric packages of the CAMEL metamodel are based on the Scalability Rule Language (SRL) [20, 8]. SRL enables the specification of rules that support complex adaptation scenarios of cross-cloud applications. In particular, SRL provides mechanisms for specifying cross-cloud behaviour patterns, metric aggregations, and the scaling actions to be enacted in order to change the provisioning and deployment of an application. SRL is inspired by the Esper Processing Language (EPL)\(^\text{17}\) with respect to the specification of event patterns with formulas including logic operators and timing. SRL provides mechanisms for (a) specifying event patterns, (b) specifying scaling actions, and (c) associating these scaling actions with the corresponding event patterns. Moreover, in order to identify event patterns, the components of cross-cloud applications must be monitored. Therefore, SRL provides mechanisms for (d) expressing which components must be monitored by which metrics, and (e) associating event patterns with monitoring data. In the following, we exemplify the main concepts in the scalability and metric packages.

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

\[\text{cpu}_{i} \geq 0.5 \text{ and } \text{cpu}_{j} \geq 0.8\]

\(^{17}\)http://esper.codehaus.org/
∀i | cpu_i ≥ 50 ∧ ∃j | cpu_j ≥ 80

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

Listing 8.1 shows the scalability model in textual syntax. non-functional event CPUAvgMetricNFEAll specifies the violation of a metric condition. metric condition refers to the metric condition CPUAvgMetricConditionAll in the metric model ScalarmMetric (cf. Listing 8.2). non-functional event CPUAvgMetricNFEAny specifies a similar violation of a metric condition.

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

horizontal scaling action HorizontalScalingSimulationManager specifies a scale-out action. vm and internal component refer to the vm CPUIntensiveUbuntu-Norway and the internal component SimulationManager, respectively, in the deployment model ScalarmDeployment (cf. Listings 6.2 and 6.1).

scalability rule CPUScalabilityRule refers to the binary event pattern and the horizontal scaling action above, along with the scale requirement HorizontalScaleSimulationManager in the requirement model ScalarmRequirement (cf. Listing 7.1).

Listing 8.1: Scalarm scalability model

```plaintext
scalability model ScalarmScalability {
    non-functional event CPUAvgMetricNFEAll {
        metric condition: ScalarmModel.ScalarmMetric.CPUAvgMetricConditionAll violation
    }
    non-functional event CPUAvgMetricNFEAny {
        metric condition: ScalarmModel.ScalarmMetric.CPUAvgMetricConditionAny violation
    }
    binary event pattern CPUAvgMetricBEPAnd {
        left event: ScalarmModel.ScalarmScalability.CPUAvgMetricNFEAll
        right event: ScalarmModel.ScalarmScalability.CPUAvgMetricNFEAny
        operator: AND
    }
    horizontal scaling action HorizontalScalingSimulationManager {
        type: SCALE_OUT
        vm: ScalarmModel.ScalarmDeployment.CPUIntensiveUbuntuNorway
    }
}
```
Listing 8.2 shows the metric model in textual syntax. raw metric CPUMetric, along with the elements referred by it, specify a raw (sensor) metric measuring CPU load. composite metric CPUAverage, along with the elements referred by it, specify an average CPU load metric. composite metric context CPUAvgMetricContextAll and composite metric context CPUAvgMetricContextAny specify that the average CPU load metric is instantiated in two contexts, one with a window of five minutes and one with a window of one minute, respectively. The aggregated composite metrics are instantiated as metric instances twice per virtual machine, and once per metric context.

Listing 8.2: Scalarm metric model

```java
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
    }

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

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

raw metric CPUMetric {
value direction: 0
layer: IaaS
property: ScalarmModel.ScalarmMetric.CPUProperty
unit: ScalarmModel.ScalarmUnit.CPUUnit
value type: ScalarmModel.ScalarmType.Range0_100
}

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

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

composite metric CPUAverage {
description: "Average of the CPU"
value direction: 1
layer: PaaS
property: ScalarmModel.ScalarmMetric.CPUProperty
unit: ScalarmModel.ScalarmUnit.CPUUnit
metric formula FormulaAverage {
function arity: UNARY
function pattern: MAP
MEAN( ScalarmModel.ScalarmMetric.CPUMetric )
}
}

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

composite metric context CPUCoverageMetricContextAny {
metric: ScalarmModel.ScalarmMetric.CPUAverage
}
8.1 Interplay with Executionware

In order to enact the scalability rules, the Executionware provides a monitoring and adaptation engine for cross-cloud applications. In PaaSage, this monitoring and adaptation engine is Axe [9]. In particular, the Executionware configures the monitoring probes based on the specified metrics and evaluates the specified scalability rules. If a metric condition is violated, the Executionware enacts the specified scaling action (e.g., scale-out), which may include the provisioning of vm instances, the deployment of component instances, and the wiring of these (see Section 6.1). Life-cycle handlers attached to the specified communications can wire existing component instances by reconfiguring them. The interested reader may refer to [9] for a detailed description of how the Executionware enacts scaling actions.

9 Organisations

The organisation package of the CAMEL metamodel is based on the organisation subset of CERIF [18]. CERIF is an EU standard\(^\text{18}\) for research information.

\[^{18}\text{http://cordis.europa.eu/cerif/}\]
In particular, the organisation package of the CAMEL reuses the concepts from CERIF for specifying organisations, users, and roles. In the following, we exemplify the main concepts in the organisation package.

Assume that we have to specify the organisation model for the Scalarm use case. Listing 9.1 shows this specification in textual syntax.

organisation AGH specifies the organisation AGH (Akademia Górniczo-Hutnicza, i.e., AGH University of Science and Technology), while user Michal Orzechowski specifies the user Michal Orzechowski belonging to the organisation AGH and owning the application Scalarm (cf. Listing 5.1).

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

Listing 9.1: Scalarm organisation model

```plaintext
organisation model AGHOrganisation {
  organisation AGH {
    email: 'info@agh.edu.pl'
  }

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

  role devop

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

10 Providers

The provider package of the CAMEL metamodel is based on Saloon [32, 33, 34]. Saloon is a tool-supported DSL for specifying the features of cloud providers and matching them with requirements by leveraging feature models [3] and ontologies [17]. In the following, we exemplify the main concepts in the provider package.
Assume that we have to specify the provider model for the Scalarm use case. Listing 10.1 shows an excerpt of the provider model for a SINTEF private cloud specified using the CAMEL textual syntax.

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

feature VM is a sub-feature and specifies the attributes characterising the virtual machine flavours provided by SINTEF’s private cloud, such as type (attribute VMType), operating system (attribute VMOS), size of RAM (attribute VMMemory), size of storage (attribute VMStorage), and number of CPU cores (attribute VMcores). Each attribute has a value type, and a unit type. For instance, VMMemory has MemoryList, a list of integer values (256, 512, 2048, etc.), as value type, and MEGABYTES as unit type (cf. Listings 15.1 and 14.1). feature cardinality specifies that the feature has a cardinality between 1 and 8.

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

Listing 10.1: SINTEF provider model (excerpt)

```java
provider model SINTEFProvider {

  root feature SINTEF {

    attributes {

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

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

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

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

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

sub-features {
  feature VM {
    attributes {
      attribute VNodeType (value type: ScalarsModel.SINTEFType.VNodeType)
      attribute VMOS (value type: ScalarsModel.SINTEFType.VMOSEnum)
      attribute VMMemory (unit type: MEGABYTES value type: ScalarsModel.SINTEFType.MegabytesList)
      attribute VMStorage (unit type: GIGABYTES value type: ScalarsModel.SINTEFType.GigabytesList)
      attribute VMCPUs (value type: ScalarsModel.SINTEFType.CPUList)
    }
    feature cardinality {cardinality: 1 .. 8}
    ...
  }
  feature cardinality {cardinality: 1 .. 1}
}

cardinal 

constraints {
  implies M1LARGEMapping {
    from: ScalarsModel.SINTEFProvider.SINTEF.VM
    to: ScalarsModel.SINTEFProvider.SINTEF.VM
    attribute constraints {
      attribute constraint {
        from: ScalarsModel.SINTEFProvider.SINTEF.VM.VNodeType
        to: ScalarsModel.SINTEFProvider.SINTEF.VM.VMMemory
        from value: string value 'M1.LARGE'
        to value: int value 8192
      }
      attribute constraint {
        from: ScalarsModel.SINTEFProvider.SINTEF.VM.VNodeType
        to: ScalarsModel.SINTEFProvider.SINTEF.VM.VMCPUs
        from value: string value 'M1.LARGE'
        to value: int value 4
      }
    }
  }
}
11 Security

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

Assume that we have to specify the security model for the Scalarm use case. Listing 9.1 shows this specification in textual syntax.

```plaintext
security model ScalarmSecurity {

domain IAM {

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

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

Listing 11.1: Scalarm security model
```

```plaintext
security model ScalarmSecurity {

domain IAM {

    attribute constraint {
        from: ScalarmModel.SINTEFProvider.SINTEF.VM.VMType
        to: ScalarmModel.SINTEFProvider.SINTEF.VM.VMStorage
        from value: string value 'M1.LARGE'
        to value: int value 80
    }

...}

19https://cloudsecurityalliance.org/`
name: "Identity & Access Management"
sub-domains [ScalarmSecurity.IAM_CLCPM, ScalarmSecurity.IAM_CLCPM]

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

domain IAM_UAR {
    name: "User Access Revocation"
}

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

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

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

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

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

organisation model AmazonExt {

provider Amazon {
email: 'contact@amazon.com'
PaaS
IaaS

security capability [ScalarmModel.ScalarmSecurity.SecCap]
}

12 Execution

The execution package of the CAMEL metamodel is not based on existing DSLs and has been developed to enable the recording of historical data about the execution of cross-cloud applications. Historical data, such as metric measurements and SLO assessments, can be used for auditing purposes as well as for optimising the CAMEL model to better exploit the available cloud infrastructures [22]. In PaaSage, the execution model is automatically manipulated by the PaaSage platform during the execution phase (see Section 2), and so it should be in the general case too. In the following, we exemplify the main concepts in the execution package.

Assume that we have to record the execution of the Scalarm use case. Listing 12.1 shows this specification in textual syntax.

vm binding ScalarmVMBinding specifies that the virtual machine instance Core-IntensiveUbuntuNorwayInst (cf. Listing 6.5) is bound to the execution context EC1.

raw metric instance CPUMetricInstance specifies that the metric instance CPUMetricInstance is an instance of the metric CPUMetric and is bound to the virtual machine instance CoreIntensiveUbuntuNorwayInst and the execution context EC1 (cf. Listing 6.5).

execution context EC1 specifies the current execution context. It refers to the application being executed, the deployment model of the application, the requirement group that led to this deployment model, and an indication of the total cost of application execution along with a reference to the corresponding monetary unit (cf. Listing 14.1).

vm measurement VM1 specifies the virtual machine measurement for the CPU metric instance. It refers to the execution context, the metric instance, the virtual
machine instance (cf. Listing 6.5), 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 12.1: Scalarm execution model

```plaintext
metric model ScalarmMetric {
    vm binding ScalarmVMBinding {
        execution context: ScalarmExecution.EC1
        CoreIntensiveUbuntuNorwayInst
    }

    raw metric instance CPUMetricInstance {
        metric: ScalarmModel.SavelengthMetric.CPU
        sensor: ScalarmMetric.CPU.Sensor
        binding: ScalarmModel.SavelengthMetric.SavelengthVMBinding
    }

    execution model ScalarmExecution {
        execution context EC1 {
            application: ScalarmModel.ScrollerApplication
            deployment model: ScalarmModel.Savelength.Deployment
            requirement group: ScalarmRequirement.ScalarmRequirementGroup
            total cost: 100.0
            cost unit: ScalarmModel.ScalarmUnits.Euro
        }

        vm measurement VM1 {
            execution context: ScalarmExecution.EC1
            metric instance: ScalarmMetric.RawCPUMetricInstance
            vm instance: ScalarmModel.ScalarmDeployment.
            CoreIntensiveUbuntuNorwayInst
            value: 95.0
            time: 2016-10-31T22:50
        }

        assessment A1 {
            execution context: ScalarmExecution.EC1
            measurement: ScalarmExecution.VM1
            slo: ScalarmRequirement.CPU.ScalarmSLO
            violated time: 2016-10-31T22:50
        }
    }
}
```
13 Locations

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

Assume that we have to specify the locations for the Scalarm use case. Listing 13.1 shows this specification in textual syntax.

region EU specifies the region Europe. country NO and country DE specifies the countries Norway and Germany, respectively. parent regions refers to the parent region Europe. The location Norway is referred to by the location requirement NorwayReq in the requirement model ScalarmRequirement (cf. Listing 7.1).

Listing 13.1: Scalarm location model

```java
location model ScalarmLocation {
  region EU {
    name: 'Europe'
  }
  country NO {
    name: 'Norway'
    parent regions [ScalarmLocation.EU]
  }
  country DE {
    name: 'Germany'
    parent regions [ScalarmLocation.EU]
  }
}
```

14 Units

The unit package of the CAMEL metamodel is not based on existing DSLs and has been developed to enable the specification of units that are adopted by the following packages: (a) metric, where they are used to define the unit of measurement for a metric, (b) execution, where they are used to define the monetary unit for the cost of a particular application execution, and (c) the provider, where they are used to define the unit for a particular feature attribute. In the following, we exemplify the main concepts in the unit package.
Assume that we have to specify the units of the Scalarm use case. Listing 14.1 shows this specification in textual syntax.

The unit model ScalarmUnit encompasses seven units that are referred to by metrics in the metric model ScalarmMetric (cf. Listing 8.2). 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 14.1: Scalarm unit model

```java
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}
}
```

The complete list of available units is as follows:

- core unit, which represents the unit of CPU cores
- monetary unit, which represents a monetary unit (e.g., EUROS)
- request unit, which represents the unit of number requests
- storage unit, which represents the unit of storage (e.g., BYTES)
- throughput unit, which represents the unit of throughput (e.g., REQUESTS_PER_SECOND)
- time interval unit, which represents the unit of time interval (e.g., SECONDS)
- transaction unit, which represents the number of transactions
- dimensionless, which represents a unit without dimension (e.g., a unit of PERCENTAGE is dimensionless)

### 15 Types

The type package of the CAMEL metamodel is also based on Saloon [32, 33, 34]. It provides the concepts to specify value types and values used across CAMEL.
models (e.g., integer, string, or enumeration). In the following, we exemplify the main concepts in the type package.

Assume that we have to specify the types of the Scalarm use case. Listing 15.1 shows this specification in textual syntax.

- `range Range0_100` specifies an integer range between 0 and 99 (i.e., 100, not included). It is referred to by the metric `CPUMetric` (cf. Listing 8.2).
- `range Range0_1000` specifies an integer range between 1 (i.e., 0, not included) and 1000. It is referred to by the metric `ResponseTimeMetric` (cf. Listing 8.2).
- `range DoubleRange0_100` specifies a double range between 0.0 and 100.0. It is referred to by the metric `AvailabilityMetric` (cf. Listing 8.2).

The other value types, enumerations, and lists in the type model `SINTEFType` are referred to by the features in the provider model `SINTEFProvider AvailabilityMetric` (cf. Listing 10.1).

Listing 15.1: Scalarm type model

```plaintext
type model ScalarmType {
  range Range0_100 {
    primitive type: IntType
    lower limit {int value 0 included}
    upper limit {int value 100}
  }

  range Range0_10000 {
    primitive type: IntType
    lower limit {int value 0}
    upper limit {int value 10000 included}
  }

  range DoubleRange0_100 {
    primitive type: DoubleType
    lower limit {double value 0.0 included}
    upper limit {double value 100.0 included}
  }
}

type model SINTEFType {
  string value type StringValueType {
    primitive type: StringType
  } 

  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, ... ]
  }

  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 ]
}

16 Java APIs and CDO

As mentioned, CAMEL consists of an Ecore model (cf. Section 5). This enables to specify CAMEL models using the CAMEL Textual Editor as well as to programmatically manipulate and persist them through Java APIs.

Listing 16.1 shows the creation of a VM of Scalarm (cf. Section 2). The classes that are instantiated and initialised in the code have been automatically generated by the EMF generator model based on the deployment package. All class instances are obtained using the DeploymentFactory object specific for the deployment package. This object provides a set of methods that are used to make sure the model objects are appropriately instantiated.

Listing 16.1: A sample VM definition

```
//create a ML VM
VM coreIntensiveVM = DeploymentFactory.eINSTANCE.createVM();
//First create VM requirement set & add it to the deployment model
VMRequirementSet coreIntensiveReqs = DeploymentFactory.eINSTANCE.
createVMRequirementSet();
mlReqs.setName("CoreIntensiveReqs");
coreIntensiveVM.setVmRequirementSet(coreIntensiveReqs);
sensAppDeploymentModel.getVmRequirementSets().add(coreIntensiveReqs);
//Create a quantitative hardware requirement to include it in the
requirement set
QuantitativeHardwareRequirement coreIntensiveRequirement =
  RequirementFactory.eINSTANCE.createQuantitativeHardwareRequirement();
coreIntensiveRequirement.setName("CoreIntensive");
coreIntensiveRequirement.setMaxCores(32);
coreIntensiveRequirement.setMinCores(8);
coreIntensiveRequirement.setMaxRAM(8192);
coreIntensiveRequirement.setMinRAM(4096);
rm.getRequirements().add(coreIntensiveRequirement);
coreIntensiveReqs.setQuantitativeHardwareRequirement(
coreIntensiveRequirement);
//Create a Location requirement imposing that the VM should be located
in Scotland
```
Listing 16.2 shows the specification of Scalarm’s ExperimentManager Internal-Component. This internal component has an associated Configuration, which specifies the corresponding life cycle control scripts (e.g., download and install commands). It also specifies communications with other internal components by creating corresponding RequiredCommunication, ProvidedCommunication entities and a dedicated type of host RequiredHost that the ExperimentManager requires.

Listing 16.2: A sample InternalComponent definition

//create a Scalarm InternalComponent give it a name
InternalComponent experimentManagerIc = DeploymentFactory.eINSTANCE.createInternalComponent();

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

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

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

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

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

Listing 16.3 shows the creation of a Communication binding between the Scalarm’s ExperimentManager and the Scalarm’s StorageManager.

Listing 16.3: A sample Communication definition

Communication experimentManagerToStorageManager = DeploymentFactory.eINSTANCE.createCommunication();
experimentManagerToStorageManager.setName("experimentManagerToStorageManager");
experimentManagerToStorageManager.setProvidedCommunication(storageManagerProvidedCommunicationMongodNginx);
experimentManagerToStorageManager.setRequiredCommunication(experimentManagerReqStorageCommunication);

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

Listing 16.4 shows the specification of a Hosting binding between the ExperimentManager InternalComponent and the VM of coreIntensiveUbuntuGermanyHost type.

Listing 16.4: A sample Hosting definition

Hosting experimentManagerToCoreIntensiveUbuntuGermany = DeploymentFactory.eINSTANCE.createHosting();
experimentManagerToCoreIntensiveUbuntuGermany.setName("ExperimentManagerToCoreIntensiveUbuntuGermany");
experimentManagerToCoreIntensiveUbuntuGermany.setProvidedHost(coreIntensiveUbuntuGermanyHost);
experimentManagerToCoreIntensiveUbuntuGermany.setRequiredHost(coreIntensiveUbuntuGermanyHostReq);

//Add hosting to the deployment model
scalarmDeploymentModel.getHostings().add(experimentManagerToCoreIntensiveUbuntuGermany);
Listing 16.5 shows the process of saving a deployment model in a CDO repository. As mentioned, CDO uses a set of APIs that are designed after the JDBC APIs. In order to save a model, we first need to 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, a partner project used within CDO. Once the transaction is obtained, the deployment model refers to the CDOResource responsible for its persistence, and the transaction is committed.

Listing 16.5: Saving a deployment model in a CDO repository

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

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

// create the session configuration
CDONet4jSessionConfiguration config = CDONet4jUtil.
    createNet4jSessionConfiguration();
config.setConnector(connector);
config.setRepositoryName("repo1");

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

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

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

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

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

Listing 16.6 shows the process of loading and modifying a deployment model. In this example, the host VM for the ExperimentManager is changed. We change it by setting the ProvidedHost of the Hosting to point to a virtual machine definition suitable for single CPU intensive tasks (assuming it is initially one suited for multi-core processing).

Listing 16.6: Loading and modifying a deployment model in a CDO repository

```java
20https://www.eclipse.org/modeling/emf/?project=net4j
```
// open a new transaction
CDOTransaction transaction = cdoSession.openTransaction();

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

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

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

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

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

The example above show the Java code for programmatically saving, loading, and modifying part of a deployment model in a CDO repository. The Java code for programmatically saving, loading, and modifying models from other packages of the CAMEL metamodel is analogous. The full version of the Java code of the Scalarm example is available for reference in the Git repository at OW2.21

17 Related Work

In the following, we compare CAMEL with related work. We distinguish between tools (e.g., DevOps and cloud orchestration tools) to automate the deployment

of cloud applications and languages to model cloud aspects. For the latter category, we define six comparison criteria and evaluate the languages according to these criteria. This comparison will validate our claim that CAMEL advances the state-of-the-art in modelling and execution of cloud applications.

17.1 Tools

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

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

Cloud orchestration tools such as Cloudify\(^{26}\) or Apache Brooklyn\(^{27}\) rely on the TOSCA \[^{31}\] (see below) to specify the topologies of cloud applications along with the processes for their orchestration. These tools facilitate the provisioning, deployment, and monitoring of cloud applications across multiple cloud infrastructures \[^{2}\]. However, TOSCA does not provide an instance model and hence does not support models@run-time, which makes these tools unsuitable for reasoning on the models and hence enabling self-adaptive cross-cloud applications.

17.2 Languages

In the research community, within the Reservoir\(^{28}\) EU project, Galán et al. \[^{15}\] proposed a service specification language for cloud computing platforms, which extends the DMTF’s Open Virtualization Format (OVF) standard to address the specific requirements of these environments.

\[^{22}\]http://www.jclouds.org
\[^{23}\]http://deltacloud.apache.org/
\[^{24}\]https://puppetlabs.com/
\[^{25}\]http://www.opscode.com/chef/
\[^{26}\]http://getcloudify.org/
\[^{27}\]https://brooklyn.apache.org/
\[^{28}\]http://www.reservoir-fp7.eu/
Within the 4CaaSt\(^{29}\) EU project, Nguyen et al. [26] proposed a language to specify Blueprint Templates—a uniform abstract description for cloud service offerings that may cross different cloud computing layers, \textit{i.e.}, infrastructure and platform.

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

The ARTIST project\(^{31}\) provides the Cloud Application Modelling Language (CAML). CAML consists of an \textit{internal} DSL [14] realised as a UML library along with UML profiles [4] rather than an \textit{external} DSL such as CAMEL. The main rationale behind the latter stems from the goal of the ARTIST project to support the migration of existing applications to the cloud, whereby UML models are reverse-engineered and tailored to a selected cloud environment.

The ARCADIA project\(^{32}\) provides a methodology and a framework to support the development of highly-distributed applications (HDAs) that are reconfigurable by design. The ARCADIA Framework [36] relies on unikernel technology in order to bundle microservices, and leverages on an extensible \textit{context model} throughout the entire life cycle of HDAs. Similar to CAMEL, the ARCADIA Context Model [16] has multiple facets, such as the component model, the service graph model, the service deployment model, and the service run-time model.

In the standards community, the Topology and Orchestration Specification for Cloud Applications (TOSCA) [31] is a specification developed by the OASIS consortium, which provides a language for specifying the components comprising the topology of cloud applications along with the processes for their orchestration.

### 17.3 Comparison

In the following, we define six comparison criteria and evaluate the aforementioned languages according to these criteria. These criteria were selected to evaluate the usefulness, usability, and self-adaptation support of the languages. In

\(^{29}\)\url{http://www.4caast.eu}
\(^{30}\)\url{http://www.modaclouds.eu/}
\(^{31}\)\url{http://www.artist-project.eu/}
\(^{32}\)\url{http://www.arcadia-framework.eu}
particular, the abstract syntax and aspect coverage, delivery model support, and models@run-time support reflect the usefulness of the language; the concrete syntax and integration level reflect the usability; and models@run-time support also reflects the self-adaptation support.

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

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

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

**Integration level.** A language that covers multiple aspects may provide different levels of integration across these aspects, especially when these aspects include similar or equivalent concepts. The integration solution has to: (a) join equivalent concepts and separate similar concepts into respective sub-concepts; (b) homogenise the remaining concepts so that they are defined at the same level of granularity; (c) enforce a uniform formalism and notation for the abstract and

---

33 Abstract syntaxes defined in Ecore fall into this category.
concrete syntaxes; and (d) enforce the consistency, correctness, and integrity of the models. Each of these steps is a precondition to the following step and requires an increasing amount of effort. This criterion reflects how many of these steps have been adopted to integrate the sub-languages. The values for this criterion can be “low” if the sub-languages were integrated following only step (a), “medium” if they were integrated following steps (a) and (b), “high” if they were integrated following all steps, and “N/A” if they were not integrated. In the last case, each sub-language covers one aspect and is independent from the other sub-languages. This independence leads to the following disadvantages: (a) it raises the complexity of the language, since each sub-language has its own abstract and concrete syntax; (b) it steepens the learning curve and increases the modelling effort for the same reason; (d) it leads to the duplication of modelling for similar or equivalent concepts; (e) it leads to the manual validation of cross-aspect dependencies.

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

**Models@run-time support.** Models@run-time [5] provides an abstract representation of the underlying running system, whereby a modification to the model is enacted on-demand in the system, and a change in the system is automatically reflected in the model. Models@run-time can be implemented using the type-instance pattern [1], which facilitates reusability and abstraction. For instance, we implemented the type-instance pattern in two aspects, namely deployment and metric. In the case of deployment, this allows to automatically adapt the component- and virtual machine instances in the deployment model based on scalability rules (e.g., scale out a Scalarm service along with the underlying virtual machine). In the case of monitoring, this allows to automatically populate the metric model with metric instances (e.g., CPU load measurements of the virtual machine hosting the Scalarm service). This criterion reflects how many of these aspects implement the type-instance pattern. The values for this criterion can be “deployment” and “metric”.

### 17.4 Analysis

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

<table>
<thead>
<tr>
<th>Language</th>
<th>Abstract Syntax</th>
<th>Concrete Syntax</th>
<th>Aspect Coverage</th>
<th>Integration Level</th>
<th>Delivery Model Support</th>
<th>Models@run-time Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reservoir OVF Extension</td>
<td>XML Schema</td>
<td>XML</td>
<td>low</td>
<td>N/A</td>
<td>IaaS</td>
<td>N/A</td>
</tr>
<tr>
<td>4CaaS Blueprint Template</td>
<td>XML Schema</td>
<td>XML, gra, txt</td>
<td>low</td>
<td>N/A</td>
<td>IaaS, Paas</td>
<td>N/A</td>
</tr>
<tr>
<td>ModaCloudML</td>
<td>MOF</td>
<td>gra</td>
<td>medium</td>
<td>low</td>
<td>IaaS</td>
<td>deployment</td>
</tr>
<tr>
<td>CAML</td>
<td>XML Schema</td>
<td>XML</td>
<td>medium</td>
<td>medium</td>
<td>IaaS</td>
<td>deployment</td>
</tr>
<tr>
<td>ARCADIA Context Model</td>
<td>XML Shader</td>
<td>XML</td>
<td>high</td>
<td>medium</td>
<td>IaaS, Paas</td>
<td>deployment</td>
</tr>
<tr>
<td>TOSCA</td>
<td>XML Shader</td>
<td>XML, txt</td>
<td>medium</td>
<td>medium</td>
<td>IaaS, Paas</td>
<td>deployment</td>
</tr>
<tr>
<td>CAMEL</td>
<td>MOF</td>
<td>XML, gra, txt</td>
<td>high</td>
<td>high</td>
<td>IaaS</td>
<td>deployment, metric</td>
</tr>
</tbody>
</table>

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

MODACloudML and CAMEL achieve similar goals, but with different approaches: MODACloudML is a family of loosely coupled DSLs, while CAMEL is a standalone language. CAMEL has the advantage of providing a uniform abstract and concrete syntax, which resulted from a process of coupling and homogenising multiple DSLs [27]. However, ModaCloudML supports both IaaS and PaaS while CAMEL only supports the former. CAMEL will be extended to include support PaaS in future work.

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

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

TOSCA supports the specification of types and templates, but not instances, in deployment models. CAMEL, in contrast, supports the specification of types, templates, and instances. Note that CAMEL provides built-in types such as vm, internal component, communication, and hosting, while TOSCA offers similar types in a library of reusable types and allows to define arbitrary types. In its current form, TOSCA can only be used at design-time, while CAMEL can be used at both design-time and run-time.

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

18 Conclusion and Future Work

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

In this document, we described the design and implementation of CAMEL. Moreover, we provided a real-world running example to illustrate how to specify models in a concrete textual syntax and how to programmatically manipulate and persist them through Java APIs. Finally, we described the abstract syntax of the language.

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

In addition, CAMEL models that conform to an old version of CAMEL often have to be migrated to conform to its current version. In the future, we would like to integrate a solution for the challenge of maintaining multiple versions and automatically migrating CAMEL models [27] based on CDO and Edapt.

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

Acknowledgements. The research leading to these results was supported by the European Commission’s Seventh Framework Programme (FP7/2007-2013) under grant agreement number 317715 (PaaSage) and the European Commission’s Framework Programme Horizon 2020 (ICT-07-2014) under grant agreement numbers 644690 (CloudSocket) and 645372 (ARCADIA). The authors would like to thank the use case providers and component developers in the

\textsuperscript{34}http://www.cactosfp7.eu/
\textsuperscript{35}https://www.cloudsocket.eu
\textsuperscript{36}http://www.musa-project.eu/
above projects for the constructive feedback on CAMEL. Michal Orzechowski is also grateful to AGH University of Science and Technology for their support under grant number 15.11.230.212.

References


Appendix A  Abstract Syntax

As mentioned, the CAMEL metamodel is organised into packages, whereby each package reflects an aspect (or domain).

Figure 5 shows the top-level camel package of the CAMEL metamodel. A CamelModel consists of an Application along with a collection of sub-models, namely DeploymentModels (see Section A.1), RequirementModels (see Section A.2), MetricModels (see Section A.3), ScalabilityModels (see Section A.3), ProviderModels (see Section A.5), OrganisationModels (see Section A.4), SecurityModels (see Section A.6), ExecutionModels (see Section A.7), LocationModels (see Section A.8), UnitModels (see Section A.9), and TypeModels (see Section A.10).

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

## A.1 Deployment

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

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

A `VMRequirementSet` represents a set of requirements for virtual machines, *i.e.* hardware requirements, operating system requirements, location requirements, and image requirements (see Section A.2). It can be referred to by a virtual machine or by a deployment model. In the former case, the set of requirements applies to all instances of the virtual machine, whereas in the latter case, the set
A **CommunicationPort** (see Figure 6) represents a communication port of an application component. A communication port can be a **ProvidedCommunication**, meaning that it provides a feature to another component, or a **RequiredCommunication**, meaning that it consumes a feature from another component. The property **isMandatory** of **RequiredCommunication** represents that the component depends on the feature provided by another component.

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

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

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

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

## A.2 Requirements

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

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

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

A requirement group allows for the creation of 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) could in turn contain single requirements, such as SLOs, logically connected by the AND operator.

A.2.1 Hardware, OS & Image and Provider Requirements

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

![Class Diagram](image)

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

HardwareRequirement, OsOrImageRequirement, and ProviderRequirement are hard requirements. They can be referred to by a VMRequirementSet (see Section A.1).

A HardwareRequirement can be specialised into a QualitativeHardwareRequirement, which represents a requirement on the performance of a virtual machine instance, or a QuantitativeHardwareRequirement, which represents a requirement on the virtual hardware of a virtual machine. The properties min- and maxBenchmark of QualitativeHardwareRequirement represent the range of benchmark results that a virtual machine instance must satisfy. The properties of QuantitativeHardwareRequirement represent the ranges of: CPU frequency, number of CPU cores, size of RAM, and size of storage that a virtual machine instance must satisfy. Note that for at least one of these properties, at least one of the minimum and maximum bounds must be specified.
An OsOrImageRequirement can be specialised into an OSRequirement, which represents a requirement on the operating system run by a virtual machine, or a ImageRequirement, which represents a requirement on the image deployed on a virtual machine. The property os of OSRequirement represents the required operating system (e.g., “Ubuntu” or “Windows”), while the property is64os represents whether the operating system must be compiled for 64-bit architectures (e.g., x86-64). The property imageId of ImageRequirement represents the identifier of the required image.

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

A LocationRequirement refers to one or more Locations (see Section A.8), which represent either geographical regions (e.g., a continent, a country, or a region) or cloud locations (i.e., a location specific to a cloud provider) that must be considered for an application deployment (e.g., Germany only).

A.2.2 Service Level Objectives and Optimisation Requirements

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

![Class Diagram](image)

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

A ServiceLevelObjective is a hard requirement. It refers to a Condition, such as MetricCondition (see Section A.3), which represents the metric condition that must be satisfied (i.e., the corresponding measurement values must not cross a particular threshold).

An OptimisationRequirement is a soft requirement. It refers to a Metric or a Property (see Section A.3), which represents the metric or the property, respectively,
that should be optimised. Moreover, it refers to an Application or InternalComponent, which represents the application or component, respectively, for which the metric should be optimised. The property optimisationFunction of OptimisationRequirement represents the optimisation function applied to the metric and can be assigned values MINIMISE or MAXIMISE.

A.2.3 Scale Requirements

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

![Class Diagram for Scaling Requirements](image)

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

A ScaleRequirement is a hard requirement. It can be referred to by a ScalabilityRule (see Section A.3), which restrains which scaling actions are 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.

A.2.4 Security Requirements

Figure 11 shows the part of the class diagram of the requirement package related to location and security requirements.
A SecurityRequirement is a hard requirement. It refers to one or more SecurityControls (see Section A.6), which represent the security controls that must be enforced. Moreover, it can refer to an Application or InternalComponent, which represent the application or component, respectively, 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 within a deployment model.

A.3 Metrics and Scalability Rules

A.3.1 Metrics, Properties, Windows, and Schedule

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

Figure 12 shows the part of the class diagram of the metric package related to its main concepts. Figure 13 shows the part of the class diagram of the metric package related to enumeration types.

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

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

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

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

Any Metric also refers to a measurable Property, which represents the measured non-functional property of a component or virtual machine. The property type represents the kind of property and can be assigned values MEASURABLE (the property can be measured directly) and ABSTRACT (the property cannot be measured and is reified by its sub-properties).

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

Figure 14 shows the part of the class diagram of the metric package related to metric instances.

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

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

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

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

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

Metric instances may refer to Windows, which represent how multiple measurements will be temporarily 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 \texttt{FIXED} represents that the window is cleared.

Metric instances may also refer to a Schedule, which represents any aspect of the operations/measurements that must be executed on a regular, timely basis (e.g., when an operation must run and when the scheduling must end). For a composite metric instance, a schedule defines when and how often the instance will be evaluated by applying the indirectly associated MetricFormula. For a raw metric instance, a schedule defines how often its value is measured by the respective metric sensor. The property \texttt{type} represents whether successive runs happen at a fixed rate or with a fixed delay. Finally, the property \texttt{intervalUnit} represents the time unit used for the schedule interval.

A.3.2 Conditions and Contexts

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

![Part of the class diagram of the metric package related to conditions](image)

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

A MetricCondition represents a constraint imposed on a metric. A constraint is violated when the comparison of the values of the instances of this metric with the threshold of the condition using the metric’s operator is false. The violation of a metric condition will lead to the triggering of a simple, non-functional event and/or a violation of an SLO (which is reflected in the respective SLOAssessment—see Section A.7). Note that constraints are not expressed on metric instances but on metrics. This enables the re-use of metric conditions in different execution contexts and guides the production of the instances of the metrics involved in these conditions to enable the assessment of the conditions under the respective particular execution contexts.

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

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

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

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

Depending on which element is associated, i.e., metric or property, a ConditionContext is further specialised into a PropertyContext and MetricContext. A PropertyContext represents the context of the property to be measured and evaluated. A MetricContext represents the metric to be used in the evaluation of the condition. For composite metrics, CompositeMetricContext refers to the contexts of the compositing metrics of the current metric. For raw metrics, RawMetricContext refers to the sensor that produces the measurements of this metric.

### A.3.3 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., in the case of scaling actions) and how or just remark that a global deployment decision has to be made (i.e., in the case of event creation actions, see next sub-section). A ScalabilityRule refers to a set of ScaleRequirements that restrict how scaling actions are performed. It also refers to Entities such as the user or the organisation associated with the scalability rule (see Section A.4).

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

### A.3.4 Actions

Figure 16 shows the part of the class diagram of the scalability package related to actions.
An Action can be specialised into a ScalingAction. The ScalingAction, in turn, can be specialised into a HorizontalScalingAction or a VerticalScalingAction. The HorizontalScalingAction refers to a VM and an InternalComponent (both specified via the deployment package). In case such an action is executed, the specified component is scaled (out or in) along with the virtual machine hosting it. The property count defines the number of additional instances to create, or the number of existing instances to destroy. In contrast to horizontal scaling, the VerticalScalingAction refers to a concrete VMInstance. The properties *Update define the amount of virtual resources (e.g., CPU cores, RAM, or storage) to be added to or removed from the virtual machine instance.

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

### A.3.5 Events

Figure 17 shows the part of the class diagram of the scalability package related to events, while Figure 18 shows the portion related to enumeration types.

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

An EventInstance represents the actual (measurement) data associated with a particular event that occurred in the system (e.g., the actual measured value and the component producing the event). The property status represents the status...
figure 17: Part of the class diagram of the scalability package related to events

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

of the event, i.e., if it is fatal, critical, warning, or success. This property can provide useful insight (e.g., for performing an analysis on QoS) while also enabling the evaluation/assessment of the events.

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

A BinaryEventPattern uses a binary operator to associate either two Events with each other or one event with a Timer. The property operator can be set to one of the common, logical operators such as AND and OR, the order operator PRECEDES, and the occurrence operator REPEAT_UNTIL (see Figure 18). PRECEDES defines that an event has to occur prior to another one. REPEAT_UNTIL defines that an event has to occur multiple times until another event occurs. In this case, the CAMEL user should define the lower and/or upper bounds for the number of event occurrences.

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

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

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

## A.4 Organisations

Figure 19 shows the class diagram of the organisation package.

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

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

A **User** represents a person belonging to an organisation. It refers to **PaaSageCredentials**, which represent the credentials to authenticate a user in the PaaSage platform. It can also refer to **CloudCredentials**, which represent the credentials to authenticate and authorise the user in a cloud provider and hence enable per-
forming 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 RoleAssignment represents the assignment of a role. It refers to either a user or a user group. The property assignmentTime represents the timestamp of the role assignment, while the properties startTime and endTime represent the start and end timestamps of the validity of the role.

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

### A.5 Providers

Figure 20 shows the class diagram of the provider package.

A ProviderModel has a root Feature and a set of Constraints. A Feature has a Feature Cardinality. The properties min and max represent the lower and upper
bound of the cardinality, respectively, (where the upper bound can also take the value of -1 to represent infinity and the 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 can 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 also be different from its sub-features.

A Constraint represents a typical restriction in binary feature models [19]. 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 cannot be together in a valid configuration). A Constraint can also be a Requires constraint, enabling the specification of restrictions of the form:

\[ [x', x''] A \rightarrow [y', y''] B \]

with \( x', x'', y', y'' \in \mathbb{N} \), 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:

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

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 an Attribute Constraint, enabling the specification of restrictions of the form:

\[
(A).c = X \rightarrow (B).d = Y
\]

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

### A.6 Security

Figure 21 shows the class diagram of the security package.

![Class Diagram](image)

Figure 21: The class diagram of the security package
A SecurityControl represents a technical or administrative countermeasure that is aimed at addressing a security risk in a cloud-based application. The property specification is used to specify textual specifications of security controls provided by the different service providers specified in the CAMEL model.

A security control refers to the Raw- and CompositeSecurityMetrics, which specialise Raw- and CompositeMetrics (see Section A.3), respectively, and represent the raw and composite security metrics associated with the security control. It also has two references to SecurityDomain, which represents the security domain associated with the security control (e.g., “Identity & Access Management”) and allows the grouping of security controls, properties, and metrics.

A SecurityProperty specialises Property (see Section A.3) 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 (see Section A.2) and represents a security SLO.

### A.7 Execution

Figure 22 shows the class diagram of the execution package.

![Diagram](image-url)

**Figure 22: The class diagram of the execution package**

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

A Measurement represents a measurement produced during application execution. It refers to a MetricInstance (see Section A.3) 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 [10]). The property measurementTime represents the timestamp of the measurement. A Measurement can be an ApplicationMeasurement, an InternalComponentMeasurement, a CommunicationMeasurement, or a VMMeasurement, depending on which application or element within a deployment model it refers to.

An SLOAssessment represents an assessment of whether the metric condition in an SLO is violated or not. It refers to an execution context, a measurement, and an SLO (see Section A.2). 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 (see Section A.3). 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.

A.8 Locations

Figure 23 shows the class diagram of the location package.

![Diagram of the location package]

Figure 23: The class diagram of the location package
A Location represents a physical or virtual location. It can be specialised into a GeographicalRegion, which represents a geographical region, or a CloudLocation, which represents a virtual 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 for the creation of hierarchies of geographical regions (e.g., continent, subcontinent, and country). A GeographicalRegion can be a Country, which represents a distinct entity in political geography. Similar to the geographical region, a cloud location can refer to a parent location. Note that both the geographical region and the cloud location should not (recursively) refer to themselves.

A.9 Units

Figure 24 shows the class diagram of the unit package.

Figure 24: The class diagram of the unit package

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

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

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

Figure 25 shows the class diagram of the enumerations in the unit package.

![Figure 25](image)

Figure 25: The class diagram of the enumerations in the unit package

A.10 Types

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

Figure 26: The class diagram of the type package