PaaSage

CAMEL Model Creation

A Brief Tutorial

Tutorial version 1.0
5/4/2016

Created by: Manos Papoutsakis

Based on: CAMEL Documentation v2015.9 (contributors: Alessandro Rossini, Kiriakos Kritikos, Nikolay Nikolov, Jorg Domaschka, Frank Griesinger, Daniel Seybold, Daniel Romero)

1. Camel Model Creation

1.1. Overview

The Cloud Application Modeling and Execution Language (CAMEL) [D2.1.3] super-DSL has been formed by aggregating pre-existing domain-specific languages (DSLs) (e.g., CloudML [CloudML] and Saloon feature model [SALOON]) as well as new ones developed in the context of the PaaSage project (e.g., Scalability Rule Language (SRL) [SRL]). This super-DSL provides significant support to the model-driven engineering approach adopted by the PaaSage project in order to facilitate the whole multi-cloud application management lifecycle. To this end, CAMEL is able to capture various aspects in the latter lifecycle, including deployment and non-functional requirements as well as scalability rules. This documentfocuses mainly on the latter three aspects as they reflect the needs of application developers/owners, which guide the deployment plan derivation process, as well as the way the way this deployment plan can be evolved during application runtime. Apart from the aforementioned three main aspects, additional ones are also outlined by relying solely on the explication of the way respective aspect-specific modelling constructs can be specified such that are re-used to support the modelling of these three aspects.

While there are different ways via which CAMEL models can be specified, CAMEL Textual Editor is used here, which relies on the textual syntax of CAMEL. This is in accordance to the latest documentation in CAMEL (see [D2.1.3]) as well as to the fact that, as the main target users for this language are devops users, such type of users will benefit the most from textual rather than graphical-based editors. Instructions about how to install and use the CAMEL Textual Editor can be found in the CAMEL documentation at http://camel-dsl.org/documentation/.

In the following, we adopt the Scalarm use case as a running example to exemplify how to specify CAMEL models in textual syntax. The complete Scalarm CAMEL model in textual syntax can be downloaded at:
https://tuleap.ow2.org/plugins/git/paasage/camel?p=camel.git&a=blob&h=421be9e1caa955ee9d15725a6c26966cc5df9e9f&hb=f8905ee94cbef60fdf49a6aabe274e33b32ff022&f=examples/Scalarm.camel.
Scalarm stands for Massively Scalable Platform for Data Farming and intends to fulfil the following requirements:

- support all phases of a data farming experiment, starting from the experiment design phase, through simulation execution and progress monitoring, to statistical analysis of results,
- support different sizes of experiments from dozens to millions of simulations through massive scalability,
- support for heterogeneous computational infrastructure including private servers, computer clusters, grids and clouds.

Scalarm's architecture utilizes a service-oriented approach with an additional modification, which addresses the scalability requirement.

Pre-Requisites
In order to follow this tutorial, it is recommended that the CAMEL Textual Editor is installed and launched in your system. This will enable you to copy the models illustrated in this tutorial and play with them such that you get accustomed and learn CAMEL. You should also be aware of some of the main features of this editor (see last section in this document as well as https://eclipse.org/Xtext/#feature-overview), such as auto-completion, which can assist you in the rapid specification and updating of your models. The prospective reader does not need to have any knowledge of CAMEL to understand the content of this tutorial but in some cases it is recommended that the user reverts to the CAMEL documentation in order to fully comprehend some relevant notions or inspect further details, if needed – this might happen in some cases as the goal of this tutorial is not to cover CAMEL in its entirety.

Audience
We consider that the specification of a CAMEL application model involves input from three types of users. To this end, this tutorial targets all of them. These user types are the following:

Application Designer: This type of user is expected to have knowledge of the main deployment and non-functional requirements of the application at hand, such as the application topology and requirements on application *response time*.

Business User: This type of user will set the higher level business requirements, such as the *cost* of application execution, and specific business policies/restrictions, such as data processing only using EU hosts.

Systems Admin: This type of user will know the wider technical context from an organizational perspective that the application should execute within. Requirements such as the wider security policies and technical details (e.g., OS-specific component configuration demands) can be set by this user type.

The end user which monitors the application execution in the Cloud against the requirements posed could belong to either one of the user types above depending also on the level of detail and interference required. The end user choice also depends on the organization's characteristics. It may also be possible that the setting of requirements is delegated to one of the above user types, although, as already stated, requirements from each type of user is needed for the PaaSage platform to provide the optimum Cloud-based application management.

1.2. CAMEL creation

A *CamelModel* is a collection of sub-models mapping to the capturing of different information aspects, including deployment, requirement, measurement/metric, scalability and organization, relevant for the multi-cloud application management lifecycle. Each aspect is mapped to a respective sub-model. All relevant aspects and corresponding sub-models, associated to the different types of user requirements, policies and rules that can be specified, are discussed in detail in the sequel in different sub-sections.

1.2.1. Deployment Aspect – DeploymentModel

A *DeploymentModel* 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 deployment element's life cycle.

1.2.1.1. Components

A *Component* represents a reusable type of application component. A component can be an *InternalComponent* managed by the PaaSage platform, or a requirement for a *VM* (short for virtual machine) offering maintained by the cloud provider. A virtual machine or a deployment model can be associated to a *VMRequirementSet*, which refers to a set of requirements for a single virtual machine or for all virtual machines, respectively, such as hardware, operating system and location requirements. These requirements are specified via a CAMEL requirement model (see also Listing 6).

Assume that we have to specify the Experiment Manager component of the Scalarm use case. Listing 1 shows this specification in textual syntax where the corresponding component has been mapped to the definition of an `internal component` (i.e., an internal software component of the application) called *`ExperimentManager`*. `provided communication` *`ExpManPort`* represents that the Experiment Manager offers a communication port (443) via which its features can be exploited. `required communication` *`StoManPortReq`* and *`InfSerPortReq`* specify that the Experiment Manager requires features from the Information Service, which is another internal component, through port 11300 and from the Storage Manager through port 20001, respectively. The property `mandatory` of the latter signifies that the communication between the components should be obligatorily established, as the Execution Manager component needs to exploit the Storage Manager features from the very beginning of its initialization. As such, the Storage Manager will have to be started before the start of the Execution Manager.

Listing 1: Scalarm sample `internal component`

```
camel model ScalarmModel {

deployment model ScalarmDeployment {

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

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

`required host` *`CoreIntensiveUbuntuGermanyReq`* indicates that the Experiment Manager needs to be hosted on a specific VM that satisfies certain requirements indicated in the description of this *VM* in the model. `configuration` *`ExperimentManagerConfiguration`* specifies the commands to handle the life cycle of the Experiment Manager. `download`, `install`, and `start` specify the Unix shell scripts for downloading, installing, and starting the Experiment Manager, respectively.

Then, assume that we have to specify the virtual machine on which the Experiment Manager needs to be deployed (which can be used for other VMs, if this is necessary). Listing 2 shows this specification in textual syntax. `requirement set` *`CoreIntensiveUbuntuGermanyRS`* specifies a reusable set of requirements for the VM being modelled. `quantitative hardware`, `os`, and `location` refer to the requirements CoreIntensive, Ubuntu, and GermanyReq, respectively, from the requirement model ScalarmRequirement (cf. Listing 6), mapping to the specification of the hardware requirements.

In `vm` *`CoreIntensiveUbuntuGermany`* the previous *requirementSet* is connected to the specification of the *VM* on which the Experiment Manager will be hosted.

`provided host` *`CoreIntensiveUbuntuGermany`* is the hosting port of the *VM* via which a respective component can be connected to indicate to the system that it should be hosted on that VM.

Listing 2: Scalarm sample vm

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

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

1.2.1.2. Communications

A Communication represents a reusable type of communication binding between a required and a provided communication port.

Assume that we have to specify the communication binding between the Experiment Manager and the Storage Manager. Listing 3 shows this specification in textual syntax. `Communication` *`ExperimentManagerToStorageManager`* specifies that reusable type of communication binding between the two internal components in question. `from .. to .. block` specifies that the communication binding is from the required communication port StoManPortReq of the component ExperimentManager to the provided communication port StoManPort of the component StorageManager. `type: REMOTE` specifies that the Experiment

Manager and the Storage Manager is chosen to be deployed on separate virtual machine instances.

Listing 3: Scalarm sample communication

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

1.2.1.3. Hostings

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

Assume that we have to specify the hosting binding between the Experiment Manager and the virtual machine *`CoreIntensiveUbuntuGermany`*. Listing 4 shows this specification in textual syntax.

`hosting` *`ExperimentManagerToCoreIntensiveUbuntuGermany`* in Listing 4 below, specifies such a hosting binding. `from .. to .. block` specifies that the hosting binding is from the required hosting port `CoreIntensiveUbuntuGermanyPortReq` of the component `ExperimentManager` to the provided hosting port `CoreIntensiveUbuntuGermanyPortReq` of the virtual machine `CoreIntensiveUbuntuGermany`.

Listing 4: Scalarm sample hosting

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

1.2.2. Requirement Aspect – RequirementModel

A *RequirementModel* is a collection of *Requirement*s which can be associated to an application and/or its main components. A requirement can be a *HardRequirement*, such as a service level objective (SLO) (e.g., response time < 100ms), which the PaaSage platform must satisfy at all costs, or a *SoftRequirement*, such as an optimization objective (e.g., minimize cost), which the platform will attempt to satisfy in the best possible way with no precise guarantees.

A *RequirementGroup* represents a tree-based requirement structure which can contain simple requirements as well as requirement sub-structures (i.e., complex requirements / requirement groups). The property *requirementOperator* of *RequirementGroup* represents the logical operator that is used to connect these requirements and it can be assigned two different alternative values mapping to known logical operators (AND (logical conjunction) or OR (logical disjunction)). A requirement group refers to an *Application* for which all the requirements must be satisfied.

Different kinds of requirements are supported by CAMEL, each analysed in respective subsections.

1.2.2.1. Hard requirements

A hard requirement can be attached to the specification of the requirements for a VM, or to a whole deployment model. In the former case, it specifies that instances of the VM must conform to the requirement in question. In the latter case, it specifies that all VM instances should be constrained according to that requirement.

Hardware, OS & Image and Provider Requirements

Two types of a *HardwareRequirement* exist. On the one hand, a *QualitativeHardwareRequirement* represents benchmarking constraints / requirements with the intention to have a better classification and respective filtering of the VMs according to particular aspects like computation, memory, networking (e.g., computationally-large VMs vs memory-intensive VMs) or in an overall manner (by combining benchmark results over different aspects). As such, the respective properties *min-* and *maxBenchmark* of *QualitativeHardwareRequirement* of this class represent the range of benchmark results that a virtual machine instance must satisfy. On the other hand, a *QuantitativeHardwareRequirement* represents a set of constraints over the features of a VM (e.g., core number and RAM size) which can be used to perform typical filtering over the VM offerings across all cloud providers. For instance, in Listing 6, we can see that the user imposes for a respective hardware requirement that the number of cores provided should be from 8 to 32 while the size of main memory should range from 4096 to 8192 MB.

An *OsOrImageRequirement* can be specialized into an *OSRequirement* or an *ImageRequirement*. The former represents a requirement on the operating system run by a virtual machine, where the property *os* of *OSRequirement* represents the required operating system (e.g., “Ubuntu”, “Windows”, etc.), while the property is64os represents whether the operating system must conform to a 64bit architectures (e.g., x86-64). The latter represents a requirement on the image deployed on a virtual machine, where the property *imageId* of *ImageRequirement* represents the identifier of the required image.

A *ProviderRequirement* represents alternative cloud providers that could only be considered for the application deployment (e.g., Amazon and Rackspace only).

Location Requirements

A *LocationRequirement* refers to one or more *Locations*, which represent either geographical regions (e.g., a continent, a subcontinent, a country, or even a region) or cloud locations (i.e., regions and availability zones in Amazon cloud like us-east-1a).

Security Requirements

A *SecurityRequirement* refers to one or more *SecurityControls*, which represent the security controls that must be supported for a cloud provider in order to make it amenable for selection for application VM deployment (see also Section 1.2.7 to comprehend the way security controls can be specified). Moreover, it can refer to an *Application* or *InternalComponent*, which represent the application or component on which the security controls must be enforced. If the security requirement refers to an application, then all cloud providers' offerings and services, which are used by the application, must support the corresponding security controls. In case the security requirement refers to a single component, such as a virtual machine, then only offerings from cloud providers supporting the respective security controls can be selected for the particular component.

Scale Requirements

A *ScaleRequirement* can be referred to by a *ScalabilityRule* such that the way corresponding scaling actions can be performed is restrained. 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, respectively. 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, respectively.

Service Level Objectives

A *ServiceLevelObjective* represents an SLO. SLOs are used to specify measurable performance objectives (e.g., upper and/or lower thresholds regarding availability, response time, throughput, etc.) of a cloud service. In CAMEL, a ServiceLevelObjective refers to a *Condition*, such as a *MetricCondition*, which represents the metric condition that must be satisfied (i.e., the corresponding measurement values must not cross a particular threshold). Such a condition is specified via a metric model (see Section 1.2.4).

1.2.2.2. Soft requirements

Optimization Requirements

An *OptimisationRequirement* refers to a *Metric*, which represents the metric that should be optimized. Moreover, it refers to an *Application* or *InternalComponent*. The property *optimisationFunction* of *OptimisationRequirement* represents the optimization function applied to the metric and can be assigned the values of MINIMISE or MAXIMISE.

Listing 6: Scalarm requirement model

```
requirement model ScalarmRequirement {

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

  os Ubuntu {os: 'Ubuntu' 64os}

  location requirement GermanyReq {
    locations [ScalarmLocation.DE]
  }

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

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

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

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

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

Assume that we have to specify the requirements for the components of the Scalarm use case. Listing 6 show this specification in textual syntax. `quantitative hardware CoreIntensive` specifies that a *VM* must have from 8 to 32 CPU cores and from 4 to 8 GB of RAM. `os` Ubuntu specifies a quantitative hardware requirement prescribing that a *VM* must support the 64-bit edition of the Ubuntu operating system. `location requirement GermanyReq` specifies that a *VM* must be deployed in Germany. All three above requirements are referred to by the requirement set `CoreIntensiveUbuntuGermanyRS` in the deployment model *ScalarmDeployment* (cf. Listing 2). `locations` refers to the location DE, indicating the iso2 code for the country of Germany, in the location model *ScalarmLocation* (cf. Listing 7).

`horizontal scale requirement HorizontalScaleSimulationManager` specifies that the component

SimulationManager must scale horizontally between 1 and 5 instances. `component` refers to the internal component *SimulationManager* in the deployment model *ScalarmDeployment* (cf. Listing 2).

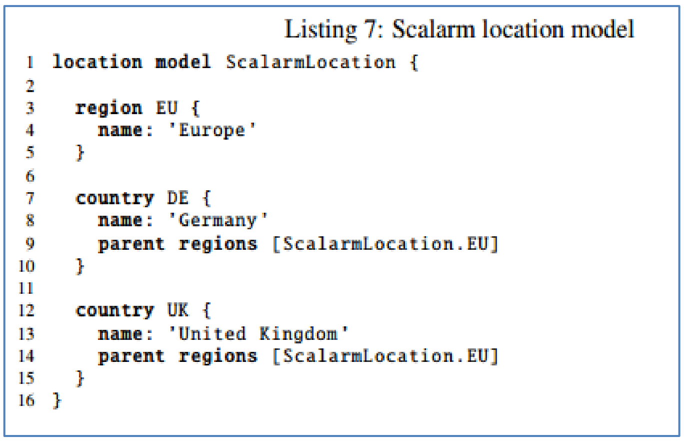
`slo CPUMetricSLO` is a specific SLO which is associated via the `service level` property to the metric condition *CPUMetricCondition* in the metric model *ScalarmModel* (cf. Listing 9). `optimization requirement MinimisePerformanceDegradationOfExperimentManager` specifies that the metric *MeanValueOfResponseTimeOfAllExperimentManagersMetric* of the component *ExperimentManager,* which is the average response time over all instances of the Experiment Manager application component*,* should be minimized and that this minimization has a priority of 0.8.

1.2.3. Location Aspect – Location Model

A *LocationModel* is a container for locations which can be mainly used to represent location requirements. Two kinds of locations can be captured. On the one hand, physical locations are represented by *GeographicalRegions*. The property *name* of such a location represents its name in English, while the property *alternativeNames* represents alternative names of this location in other natural languages. A geographical region can refer to a parent region, which allows creating hierarchies of geographical regions. A *GeographicalLocation* can be a Country, which represents a distinct entity in the political geography.

On the other hand, a *CloudLocation* represents a virtual location that is specific to a particular cloud (e.g., the eu-west-1 availability zone in Amazon EC2). Similar to the geographical region, a cloud location can refer to a parent location, which allows creating hierarchies of cloud-specific locations (e.g., regions and encompassing availability zones in Amazon EC2).

Assume that we have to specify the locations for the Scalarm use case. Listing 7 shows this specification in textual syntax. `region EU` specifies the region (continent) Europe. `country DE` specifies the country Germany. `parent regions` refers to the parent region of Europe for this country. Only the parents of a region need to be specified and not all possible ancestors. The ancestors of a country can be inferred in a recursive way by exploring the aforementioned parent-to-child relationship/property.


Listing 7: Scalarm location model

```
location model ScalarmLocation {

  region EU {
    name: 'Europe'
  }

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

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

1.2.4. Measurement/Metric Aspect – MetricModel

A metric model can be used to specific conditions over quality metrics or properties for applications and components (sw components and VMs), which can be associated to SLOs or (scalability rule) events, as well as all appropriate details to measure these metrics and properties. A condition can be specified by exploiting the following constructs analysed in the next sub-sections.

1.2.4.1. Metrics

A *Metric* is a standard of measurement which encapsulates all appropriate details for measuring non-functional properties. A *RawMetric* (e.g., raw response time) maps to the description of how raw measurements over a certain non-functional property (e.g., response time) can be produced. A *CompositeMetric*, in turn, represents an aggregated metric computed from other metrics. A metric refers to a *Unit* of measurement (e.g., the unit of SECONDS for the raw response time 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.

1.2.4.2. Metric Formulas

Each *CompositeMetric* refers to a *MetricFormula*, which explicates the computation formula used for deriving the composite metric measurements. For that purpose, a *MetricFormula* refers to one or more *MetricFormulaParameters*, which constitute its input, as well as to a pre-defined function to be applied on this input. There exist three kinds of parameters: *constants, Metrics, or MetricFormulas*. As such, a *MetricFormula* actually represents a measurement aggregation tree over particular metrics connecting different sub-formulas into a coherent whole.

1.2.4.3. Properties

Any *Metric* also refers to a measurable *Property*, i.e., the non-functional property of a component or an application that is measured by this metric. The attribute type represents the kind of property, where a value of MEASURABLE represents that the property can be measured, e.g., in the case of response time or CPU load, while a value of ABSTRACT represents that the property is not measurable. An abstract property that is not measurable can be realized by more concrete and possibly measurable properties. In this way, the construction of property hierarchies is supported.

1.2.4.4. Metric Conditions

A *MetricCondition* represents a constraint imposed on a metric. A constraint is violated when the respective condition threshold is not met by the produced measurements of this metric. The violation of a metric condition may lead to the triggering of a simple, non-functional event, which might be part of the overall event pattern of a scalability rule, and/or to the violation of an SLO.

1.2.4.5. Property Conditions

A *PropertyCondition* represents a condition on a non-functional property. This way, it is possible to specify, e.g., constraints on the cost for the whole application or one or more of its components. Then, it is up to the PaaSage platform to interpret these constraints appropriately in order to derive the required property values (e.g., based on a particular internal to the platform metric used for producing the respective property value).

1.2.4.6. Condition Contexts

A condition, either pertaining to a metric or to a property, refers to a particular *ConditionContext*, which represents the context under which it should hold. The context explicates whether the condition must be enforced on the whole application or a particular component/VM. It also indicates for how many instances of the application or component/VM the condition must be checked. Two different types of quantification are distinguished: relative, in the form of percentages over the number of instances for an application or a component, and absolute, in the form of the actual number of instances for these applications or components.

1.2.4.7. Metric Context

A *MetricContext* is a condition context that also refers to the metric to be used for evaluating a respective condition as well as to information regarding the measurement schedule and window for this metric. For a composite metric, a *CompositeMetricContext* includes a reference to the contexts of the composing metrics of this metric. For a raw metric, a *RawMetricContext* represents a reference to the sensor that produces the measurements of this metric. The PaaSage runtime generates contextual information whenever possible so that it is not necessary to create all composing contexts by hand. This is possible as some information is inherited from the composite metric’s context to its composing metrics’ contexts (actually scheduling and window of measurement information). Consequently, the definition of a context is only obligatory when information should not be inherited but differentiated for a specific composing context. For example, if we have specified the context of raw availability, the context of raw uptime (component of raw availability) does not need to include measurement scheduling and window information (e.g., measure the metric every 10 seconds) as this will be identical to the one encompassed in the availability's context.

1.2.5. Scalability Aspect – ScalabilityModel

A scalability model encompasses the specification of a set of scalability rules, regulating the adaptive runtime behaviour of particular application, along with the events used to trigger them as well as the scaling actions executed upon this triggering. These three latter constructs are analysed in more detail below in separate subsections.

1.2.5.1. Scalability Rules

A *ScalabilityRule* associates an *Event* and a set of *Actions*. The *Event* represents either a single event or an event pattern/aggregation that triggers the execution of the actions. The *Actions* either specify which components and virtual machines should be scaled (i.e., case of scaling actions) and how or just remark that a global deployment decision has to be made (i.e., for event creation actions) in case local adaptation fails or scalability limits based on given scaling requirements have been reached (that need to be associated to the respective *ScalabilityRule*). A scalability rule also refers to *Entities*, such as the user or the organization, which has specified it.

1.2.5.2. Actions

An *Action* can be specialized into a *ScalingAction* or an *EventCreationAction*. The *ScalingAction*, in turn, can be specialized 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 named by the **Update* pattern define the amount of virtual resources (e.g., CPU cores, RAM, etc.) to be added to or removed from the virtual machine instance. An *EventCreationAction* signifies via the creation of an event that the scaling actions are not sufficient to maintain the target service level of a multi-cloud application. For instance, a multi-cloud application may still violate the target response time defined in an SLO despite the scale-out or scale-up actions performed.

1.2.5.3. Events

Events can be simple or composite (i.e., event patterns). A *SimpleEvent* can be specialized into a *FunctionalEvent* or a *NonFunctionalEvent*. The *FunctionalEvent* represents a functional error (e.g., a virtual machine or a component has failed). A *NonFunctionalEvent* that refers to a metric or property condition is triggered when this condition is violated. (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. On the other hand, an event pattern is an aggregation of events based on logical or time-based operators (e.g., a logical conjunction of two other events via the AND logical operator).

Listing 8 shows the Scalarm's scalability model in textual syntax. This model encompasses one scalability rule that associates one binary event pattern with a scale-out action, while it is restricted by the scaling policy specified in Listing 6. The semantics of this rule specifies that we need to scale-out the *SimulationManager* component of Scalarm when particular bounds/conditions of two metrics are violated, mapping to respective events aggregated via a logical conjunction into the corresponding binary event pattern, provided that the number of instances of this component is less than 5. The scale-out action specification indicates important information about the scaling, such as the scale action type, which is the component to be scaled and on which VM type/offering it will be hosted.

Listing 8: Scalarm scalability model

```
scalability model ScalarmScalability {

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

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

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

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

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

requirement model ScalarmRequirement {

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

Listing 9 shows the Scalarm's metric model in textual syntax, which encloses the specification of the event conditions involved in the previously analysed scalability rules, and the corresponding metrics encompassed in these conditions along with their scheduling information. The two metrics map to common information for two families of metrics: (a) a raw (sensor) metric measuring CPU load and (b) an average CPU load metric; the latter metric will be instantiated with two different contexts, one with a window of five minutes, and another with a window of one minute. This is due to the semantics of the corresponding conditions mapping to these contexts which impose applying different bounds on the same composite metric with however different measurement scheduling and window directives. In particular, one condition (*CPUAvgMetricConditionAll*) will be violated when the average CPU, computed every 1 minute with a sliding window of 5 minute, for all instances of the *SimulationManager* component is greater than 50%, while the other condition (*CPUAvgMetricConditionAny*) will be violated when the average CPU, computed every 1 minute with a sliding window of 1 minute, for any instance of SimulationManager is greater than 80%.

Listing 9: Scalarm metric model

```
metric model ScalarmMetric {

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

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

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

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

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

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

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

  composite metric CPUAverage {
    description: "Average of the CPU"
    value direction: 1
    layer: PaaS
    property: ScalarmModel.ScalarmMetric.CPUProperty
    unit: ScalarmModel.ScalarmUnit.CPUUnit

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

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

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

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

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

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

  metric condition CPUAvgMetricConditionAny {
    context: ScalarmModel.ScalarmMetric.CPUAvgMetricContextAny
    threshold: 80.0
    comparison operator: >
  }
}
```

1.2.6. Security Aspect – *SecurityModel*

A *SecurityModel* is container of security-related constructs which can be exploited to specify security requirements and capabilities that can assist in the filtering of the cloud provider space during deployment plan reasoning. Such constructs are now analysed in detail.

A *SecurityControl* represents a technical or administrative countermeasure that aims at addressing security risks in a cloud-based application. Such a construct actually characterises high-level security requirements or capabilities that have to be satisfied or realised by the application owner or cloud provider, respectively. The *property* specification is used to specify textual descriptions of security controls in the CAMEL model. A security control can be linked to raw or composite security metrics which are specialisations of non-functional metrics (see Section 1.2.4). This kind of linkage enables connecting high-level requirements or capabilities expressed via security controls to more concrete requirements or capabilities expressed via conditions on security metrics. As such, we can evaluate whether a particular security control is satisfied via assessing the respective conditions on metrics associated to this control. A security control is also associated to a security domain and sub-domain. The latter constructs can be exploited to perform a partitioning of security-related building blocks in terms of security controls, metrics and properties.

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

```
security model ScalarmSecurity {

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

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

  domain IAM_UAR {
    name: "User Access Revocation"
  }

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

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

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

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

requirement model ScalarmExtendedReqModel {

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

organisation model AmazonExt {

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

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

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

A security property is a kind of a non-functional property. *Certifiable* security properties can actually be measured/certified and are thus connected to respective security metrics. A *SecuritySLO* is a kind of SLO which involves security metrics and properties in its conditions.

Assume that we have to specify a security model for the Scalarm use case. Listing 12 above shows this specification model in textual syntax. `domain IAM` specifies the security domain of Identity & Access Management (IAM). `domain IAM_CLCPM` and `IAM_UAR` specify two sub-domains of IAM, namely Credential Life Cycle/Provision Management (CLCPM) and User Access Revocation (UAR), respectively. `Property IdentityAssurance` specifies an abstract security property associated with the security domain IAM.

`security control IAM_02` (related to the establishment of user-control access and policies at the cloud provider side) specifies a security control associated with the security sub-domain (CLCPM) and the property `IdentityAssurance`. Similarly, `security control IAM_11` (related to the timely deprovisioning of user access) specifies a security control associated with the security sub-domain (UAR) and the property `IdentityAssurance`. Note that these security controls are part of the set of security controls of the Cloud Control Matrix[1] identified by the Cloud Security Alliance (CSA)[2]. `security capability SecCap` specifies a security capability associated with the security controls `IAM_02` and `IAM_11`. Finally, the `organisation model AmazonExt` refers to the security capability `SecCap`, which specifies that the Amazon provider supports this security capability.

1.2.7. Type Aspect - *TypeModel*

The type model includes the specification of values as well as of the types to which these values conform. Such types can be associated to metrics and feature attributes.

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*. A numeric value can also be specialized 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. On the other hand, the *EnumerateValue* represents an enumerated value. The property *name* represents the string associated with the value, while the property *value* represents the integer associated with the value (or position in the enumeration).

[1] https://cloudsecurityalliance.org/download/cloud-controls-matrix-v3-0-1/
[2] http://www.cloudsecurityalliance.org

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 having members which can be of a basic (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 referenced *type* represents the complex value type, and it has to be used in the second case.

A Range represents a range-based value type. It has two references *lowerLimit* and *upperLimit* to a Limit. A *limit* represents an actual bound, either upper or lower, of a range. The property *included* indicates whether the limit's value is included or not in the range. The *RangeUnion* represents a union of range-based value types. It refers to the contained range-based value types as well as to the primitive type that is common across all the contained value types.

Assume that we have to record the types of the Scalarm use case. Listing 14 shows this specification in textual syntax. The `range` statements specify two integer-based ranges and one double-based range. The first range is associated as a value type to the `CPUMetric` (cf. Listing 9 to represent that CPU metric values should be between 0 and 100, both included). The second range is associated as a value type to the `ResponseTimeMetric` to signify that the values of this metric should be between 0, not included (i.e., between 1), and 10000, included. The third range is associated to the `AvailabilityMetric` and signifies that the respective metric values should be between 0.0 and 100.0, where both bound/limit values are included.

Listing 14: Scalarm type model

```
type model ScalarmType {

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

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

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

1.2.8. Unit Aspect – *UnitModel*

A *UnitModel* is a collection of units that can be associated to metrics of a metric model or attributes of a provider model. A *Unit* represents an abstract unit. It can be specialised into the following classes:

- *CoreUnit*, which represents the unit of CPU cores
- *MonetaryUnit*, which represents a monetary unit (e.g., EUROS)
- *RequestUnit*, which represents the unit of number of 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).

Assume that we have to specify the units of the Scalarm use case. Listing 15 shows this specification in textual syntax. The unit model encompasses seven units that are used in the metric model. The specification of each unit follows the pattern: <unit_class> <unit_name>: <unit_type> (where the latter is an enumeration of all possible unit types). For instance, monetary unit {Euro: EUROS} specifies a monetary unit named "euros" and typed EUROS.

Listing 15: Scalarm unit model

```
unit model ScalarmUnit {

  monetary unit {Euro: EUROS}

  throughput unit {SimulationsPerSecondUnit: TRANSACTIONS_PER_SECOND}

  time interval unit {ResponseTimeUnit: MILLISECONDS}

  time interval unit {ExperimentMakespanInSecondsUnit: SECONDS}

  transaction unit {NumberOfSimulationsLeftInExperimentUnit:
    TRANSACTIONS}

  dimensionless {AvailabilityUnit: PERCENTAGE}

  dimensionless {CPUUnit: PERCENTAGE}
}
```

1.3. Conclusion

In this document, we have shortly analysed each aspect that can be captured by CAMEL, mostly related to the specification of non-functional and deployment requirements as well as scalability rules. For each aspect, we have described the main modelling concepts, their properties and relations, while we have provided concrete examples of the respective aspect-specific part of the CAMEL syntax by relying on the Scalarm use case.

Through the use of the CAMEL textual editor, we believe that the prospective user does not only have access to many interesting editing services but also have the capability to learn the CAMEL essentials without reverting to any extensive CAMEL documentation as well as produce in the end CAMEL models in a quite rapid manner.

The editor's services encompass capabilities for syntactic and semantic highlighting, domain validation reporting, auto-completion and clever suggestion (by also catering for user-intuitive cross-reference specification within or across CAMEL sub-models), while the automatic generation of the XMI CAMEL form is also supported. Such capabilities and editing mode cater mainly *devops* and *admin* types of users as they are more close to the way these user types work. If the remaining user type, i.e., a business user, is not comfortable with this editing mode, then he/she can revert to the alternative ways to specify CAMEL models which are graphics-based. These latter ways include the default graphical tree-based CAMEL editor offered by the Eclipse Environment which can operate over the file system or the MDDB CDO Repository [D4.1.2], or a web-based editor developed via Eclipse's RAP[3] technology which enables the on-line editing of CAMEL application (application + requirement) and organisation models over the MDDB CDO Repository.

References

[CloudML] N. Ferry, A. Rossini, F. Chauvel, B. Morin, A. Solberg, Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems, in: L. O'Conner (Ed.), Proceedings of CLOUD 2013: 6th IEEE International Conference on Cloud Computing, IEEE Computer Society, ISBN 978-0-7695-5028-2, 887–894, 2013.

[D2.1.3] A. Rossini, K. Kritikos, N. Nikolov, J. Domaschka, F. Griesinger, D. Seybold, D. Romero, D2.1.3 – CloudML Implementation Documentation (Final version), PaaSage project deliverable, 2015.

[D4.1.2] T. Kirkham, K. Kritikos, B. Kryza, K. Magoutis, P. Massonet, C. Papoulas, M. Korozi, A. Leonidis, S. Ntoa, C. Sheridan, A. Innes, Douglas A. Imrie, D4.1.2 – Product Database and Social Network System, PaaSage project deliverable, 2016.

[SALOON] C. Quinton, D. Romero, L. Duchien, Cardinality-based feature models with constraints: a pragmatic approach, in: T. Kishi, S. Jarzabek, S. Gnesi (Eds.), SPLC 2013: 17th International Software Product Line Conference, ACM, 162–166, 2013.

[SRL] K. Kritikos, J. Domaschka, A. Rossini, SRL: A Scalability Rule Language for Multi-cloud Environments, in: CloudCom, IEEE, 1–9, 2014.

[3] Eclipse.org/rap