Journal of Cloud Computing: Advances, Systems and Applications

RESEARCH **Open Access**

The cloud application modelling and execution language

Achilleos P. Achilleos[1,4*], Kyriakos Kritikos[2], Alessandro Rossini[3], Georgia M. Kapitsaki[4], Jörg Domaschka[5], Michal Orzechowski[6], Daniel Seybold[5], Frank Griesinger[5], Nikolay Nikolov[7], Daniel Romero[8] and George A. Papadopoulos[4]

Abstract

Cloud computing offers a flexible pay-as-you-go model for provisioning application resources, which enables applications to scale on-demand based on the current workload. In many cases, though, users face the single vendor lock-in effect, missing opportunities for optimal and adaptive application deployment across multiple clouds. Several cloud modelling languages have been developed to support multi-cloud resource management, but still they lack holistic cloud management of all aspects and phases. This work defines the Cloud Application Modelling and Execution Language (CAMEL), which (i) allows users to specify the full set of design time aspects for multi-cloud applications, and (ii) supports the models@runtime paradigm that enables capturing an application's current state facilitating its adaptive provisioning. CAMEL has been already used in many projects, domains and use cases due to its wide coverage of cloud management features. Finally, CAMEL has been positively evaluated in this work in terms of its usability and applicability in several domains (e.g., data farming, flight scheduling, financial services) based on the technology acceptance model (TAM).

Keywords: Cloud computing, Domain-specific language, Model-driven engineering, Models@run-time

Introduction

Cloud computing enables organisations to use (virtualised) resources in a pay-as-you-go model. By adopting this computing paradigm, organisations can reduce costs and outsource infrastructure management for their applications. Also, they can support flexible application provisioning by acquiring additional resources on-demand based on the current workload. Based on these benefits, many organisations have decided to move their applications in the Cloud.

Motivation

To support this migration, various frameworks have been developed enabling automated user application deployment and scaling. In some cases, the ability to use vendor specific tools (e.g., AWS CodeDeploy, Azure Kubernetes Service (AKS), Amazon Elastic Container Service for Kubernetes (Amazon EKS)) to manually deploy application components, observe the deployment progress and monitor the application performance is offered. Also, there are languages that support the definition of platform specific models (i.e., they are directly bound to a cloud environment such as Amazon's CloudFormation and OpenStack's HOT). However, such frameworks do not enable users to move to another Cloud provider (lock-in effect) when a respective need arises (e.g., better offerings, bad application performance, costs).

To address the vendor lock-in effects [34], multi-cloud resource management (MCRM) has been proposed [31], which offers organisations several capabilities including [2]: (a) optimal use of best possible cloud services from a variety of offerings supplied by a multitude of cloud providers; (b) ability to sustain an optimal quality level via the application dynamic reconfiguration; (c) ability to achieve a better security level by exploiting suitable security services; (d) ability to move applications near the client location to improve application performance; (e) ability to conform to national and international regulations.

*Correspondence: com.aa@frederick.ac.cy; achilleas@cs.ucy.ac.cy
[1]Frederick University, Nicosia, Cyprus
[4]University of Cyprus, Nicosia, Cyprus
Full list of author information is available at the end of the article

To support MCRM and exhibit a suitable automation level, different Cloud Modelling Languages (CMLs) have been defined in many research projects and prototypes [8]. These CMLs “focus mainly on design-time aspects, come from disjoint research activities and lack convergence with proposed standards. They also lack the right expressiveness level, while commonly cover one service type (IaaS) in the cloud stack” [8]. On the other hand, widely used and powerful container orchestrators such as Kubernetes [1] and Docker Swarm[2] suffer from limitations, such as multi-cloud support and support for basic scalability rules. For instance, for multi-cloud deployment, a Kubernetes cluster needs to be deployed manually in each cloud provider or Pipeline[3] can be used to deploy Kubernetes clusters on major cloud providers via a unified interface prior to deploying the application.

Contributions

To address the aforementioned challenges, the Cloud Application Modelling and Execution Language (CAMEL) has been devised. CAMEL is a multi-domain-specific language (multi-DSL) covering all aspects necessary for cloud application management at both design time and runtime. CAMEL has been developed mainly by appropriately integrating existing cloud-specific DSLs, such as CloudML [15] and by also defining additional ones like the Scalability Rule Language (SRL) [22]. In addition, CAMEL comes with a textual syntax, which enables the rapid specification of multi-cloud models by DevOps users.

In relevance to previous approaches, the contribution of this work lies in the innovative aspects of CAMEL that are not present in the existing literature: First, by developing a single, unified and integrated megaDSL, as recommended in [4], the user avoids having to use a set of heterogeneous DSLs and editors. This can reduce the learning curve, while it caters for better maintainability as it is easier to control the development of a unified, single DSL. Second, CAMEL supports the type-instance pattern, well suited to support the models@runtime approach [9], to enable users to provide models that abstract away from technical details, in contrast to other CMLs. In the models@runtime approach (see Fig. 1), the application state is monitored and reflected on a certain model that abstracts from quite technical details, while any changes on this model are reflected directly on the application and its provisioning.

Third, the identification of all MCRM needed information, based on the experience of CAMEL developers in implementing other CMLs, enables automated, adaptive cross-cloud application provisioning. As CAMEL targets

[1]Kubernetes - https://kubernetes.io/
[2]Docker Swarm - https://docs.docker.com/engine/swarm/
[3]Pipeline - https://github.com/banzaicloud/pipeline

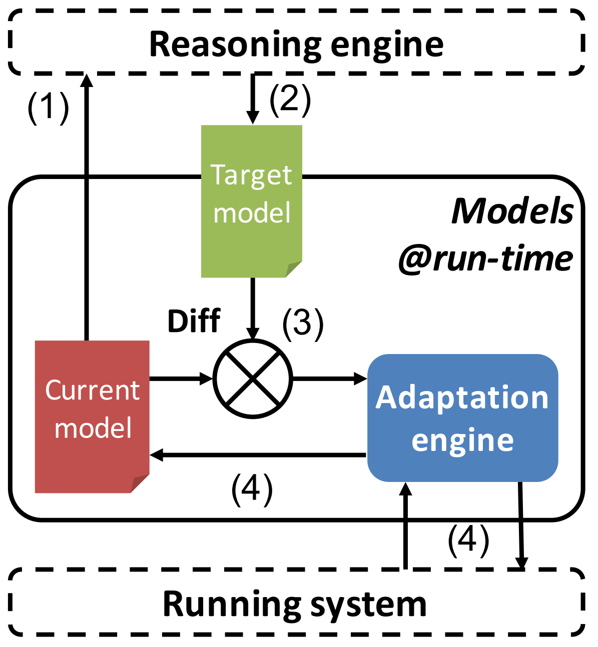


Fig. 1 Models@run-time architecture

DevOps, a user study was conducted in this work, in terms of adaptive provisioning of applications in the Cloud for various domains (e.g., data farming, flight scheduling). It shows the unique CAMEL benefits, i.e., a good level of usability, comprehensiveness and suitability. Fourth, to address heterogeneity and interoperability, CAMEL has been also aligned with TOSCA. As expressed in [8]: “Having the TOSCA standard, it is desirable to align existing and potential new CMLs for providing continuous modeling support, for example, by achieving interoperability among the languages”.

Background

CAMEL has been developed in the framework of the PaaSage EU project[4] [38]. PaaSage’s goal is to provide an aPaaS-like abstraction to its users enabling a vendor-neutral application specification mappable to different IaaS cloud providers. Hence, PaaSage offers an environment, where application developers and operators can easily develop and deploy applications on multiple cloud infrastructures, taking advantage of flexibility, adaptivity and scalability, without having to consider the specifics of different infrastructure requirements and APIs. In that context, CAMEL is an important part of the PaaSage development and deployment platform. Its eco-system supports a dedicated social network, where the users can share their CAMEL models [30]. Based on the above, the aim of the current paper is to present the CAMEL language and how it addresses the issues required for

[4]PaaSage EU FP7 Project - https://paasage.ercim.eu/

successful multi-cloud application design, whereas the actual model execution, management and adaptation is performed by other components of the PaaSage platform. Their presentation is outside the scope of the current paper. High-level information on how CAMEL is integrated in the PaaSage platform and its workflow are provided in "CAMEL in the PaaSage workflow" section, whereas dedicated papers cover specific aspects of the platform, such as security enforcement [23].

CAMEL has already been adopted, extended and used in several EU research projects (PaaSage, CloudSocket[5], CACTOS[6]) to support the modeling and execution of applications distributed over multiple cloud environments. Within these projects, CAMEL has also been extended to support PaaS and SaaS cloud services [27] and has been established as a baseline for the provisioning of Business Process as a Service [18]. It currently continues to evolve in the H2020 Melodic project[7], to address the challenges of multi-cloud management of large-scale optimised data-intensive computing applications [20].

Structure of this document

The rest of the article is structured as follows. The next section presents the key step of the requirements analysis and the subsequent steps that demonstrate the rationale behind how CAMEL has been defined, designed and developed. "The CAMEL language" section provides an overview of CAMEL, presents the key role of CAMEL in the workflow of the PaaSage platform and defines the CAMEL metamodels. "CAMEL application: the data farming use case" section explicates how a certain use case from PaaSage can benefit from its modelling via CAMEL and its subsequent evolution via the application of PaaSage's model-based MCRM framework. "Evaluation" section introduces the user study performed in this work and discusses its main results. The related work is reviewed in "Related work" section and a criteria-based comparative study of the CAMEL language with other CMLs is also presented in this section. Finally, "Conclusions & future work" concludes the article and draws directions for further research.

CAMEL specification and implementation

This section presents the steps for the specification and implementation of the CAMEL. Initially the analysis and extraction of the CAMEL requirements is presented. These form the basis for subsequent steps defined and presented as follows: (i) the definition of a suitable design and development approach, (ii) the identification of the complete set of MCRM aspects to be covered by the CAMEL language, (iii) the selection, adaptation and extension of existing CMLs and DSLs to cover the MCRM aspects, (iv) defining the method for integrating these diverse languages and (v) finally the use of suitable technologies to drive the integration method for the implementation of CAMEL.

Requirements

To create CAMEL, the following requirements were derived based on the challenges presented in "Introduction" section, summarized as: 1) support design-time and models@runtime approaches, 2) unify CMLs (aspects) created in disjoint activities and prototypes and 3) achieve convergence with relevant standards.

- *models@runtime* (R_1): CAMEL must support both type and instance level, enabling to specify both provider-independent and provider-specific models. The first will drive the deployment reasoning phase, thus enabling users to define non-functional and deployment requirements in a cloud-provider-agnostic way. The second will enable to maintain a cloud-provider-specific model of both the application and monitoring topology.
- *multiple aspects coverage* (R_2): CAMEL should enable the coverage of multiple aspects, to support all phases of the MCRM lifecycle.
- *high expressiveness level* (R_3): A suitable expressiveness level should be employed to capture accordingly required aspects of the respective domain. This enables both the users to specify the needed application information and the system to maintain and derive such information at a detailed level, so as to support all application lifecycle management phases.
- *Separation of concerns (R_4)*: CAMEL should support loosely-coupled packages, each covering an aspect of MCRM. This will facilitate a faster and more focused specification of models at each phase.
- *Reusability (R_5)*: CAMEL should support reusable types for multiple aspects of cross-cloud applications. This will ease the evolution of models.
- *Suitable integration level (R_6)*: All CAMEL sub-DSLs should be mapped to an appropriate integration level that can support the consistency of the information provided and minimise overlap across sub-DSLs.
- *Textual syntax support* (R_7): CAMEL targets DevOps that deal with cloud management and are akin to textual/code editing. Thus, the need to support CAMEL textual syntax arises for editing textual models.
- *Re-use of DSLs* (R_8): Existing DSLs from disjoint research activities should be reused and integrated (R_6), as attested also in [8]. This is because they provide valuable experience and information on

[5]CloudSocket EU H2020 Project - https://site.cloudsocket.eu/
[6]CACTOS EU FP7 Project - http://cactos-cloud.eu/
[7]Melodic EU H2020 Project - http://melodic.cloud

MCRM aspects. This also enables involving different DSLs communities in CAMEL evolution, while it reduces the learning curve for DevOps already familiar with them.

Design and development

CAMEL design is inspired by component-based approaches, which support the requirements of separation of concerns (R_4) and reusability (R_5). As such, deployment models can be regarded as assemblies of components exposing ports, and bindings between these ports. Furthermore, CAMEL developers have defined a design and development approach that satisfies the rest of the requirements and its composed by the following steps: (a) *Aspect/Domain Identification* [R_2]; (b) *Selection of Languages* [R_2, R_3 and R_8]; (c) *Integration* [mainly R_6 but also R_1, R_4 and R_5]; (d) *Implementation* [R_7].

More to the point, this approach is based on the rationale of heterogeneous CMLs convergence, extension and optimization to produce one complete CML that takes benefit on the knowledge already captured in these languages [8]. Also, such an approach makes CML maintainability, evolution and alignment with the standards (i.e., TOSCA) more feasible, as attested also in the CMLs survey in [8]. Finally, organisations, apart from involving these experts in CAMEL development, have their own communities, which could enable CAMEL to keep up with changes made to those individual CMLs.

Aspect Identification

Based on the knowledge and expertise of modelling experts in PaaSage, each action involved in MCRM was mapped to specific information requirements to address a certain domain/aspect. Table 1 presents the identified aspects for fully supporting the multi-cloud application lifecycle management actions.

Language Selection

The aspects identification for MCRM, was then followed by a careful examination of existing CMLs and DSLs covering additional aspects (e.g., organisational). PaaSage experts knowledge and involvement in implementation of existing CMLs, supported greatly and assisted in selecting the following CMLs:

– Cloud Modelling Language (CloudML) [15–17] enabling to specify deployment topology models
– Saloon [35–37] covering the modelling of cloud providers and value types
– CERIF's [21] organisation part enabling to model organisations and their access control policies
– OWL-Q [25] covering the modelling of: (a) non-functional terms (metrics and attributes), (b)

Table 1 The relevant aspects for multi-cloud application management

Aspect	Phase	Rationale
Deployment	All	The PITMs and PSTMs models drive both application reasoning and deployment, while execution-related activities should be reflected in PSTM models
Requirement	Reasoning Execution	The user requirements drive application deployment reasoning, while they are also used to restrain the way local scalability can be performed at runtime
Provider	Reasoning,	Provider models enable to matchmake and select suitable cloud offerings
Security	Reasoning	High- and low-level security requirements can drive the offering space filtering, as well as the application deployment optimisation according to security criteria apart from the quality ones and cost
Metric	Reasoning, Execution	Metrics are used as optimisation criteria for deployment reasoning, while they also explicate how application monitoring can be performed during the execution phase
Scalability	Execution	Scalability rules drive the local application reconfiguration during execution
Organisation	Reasoning, Deployment	An organisation can have accounts on certain providers which reduces the offering space only to them. The credentials to these providers enable the platform to act on user behalf for deploying application components to suitable VMs
Location	Reasoning	Location requirements can be used to filter the offering space during deployment reasoning
Execution	Reasoning,	Previous execution history knowledge can be used to improve application deployment
Unit	All	Auxiliary aspect enabling to associate units of measurement to metrics and thus, indirectly, to the conditions (i.e., SLOs) posed on them
Type	All	Auxiliary aspect enabling to provide types to language elements like metrics, as well as to define different kinds of values that can be assigned to element properties

respective requirements or capabilities imposed on them in the form of constraints, and (c) units.

These CMLs and relevant DSLs served as the starting point covering many aspects for MCRM. Nevertheless, additional information was necessary and thus the focus was reverted on the coverage of missing aspects. In specific, the information coverage for the *location* aspect was minimal and thus a relevant metamodel was incorporated in CAMEL. Furthermore, for the aspects of *requirement*, *scalability*, *execution* and *security*, none of the existing DSLs had sufficient information coverage. Hence, additional aspect-specific DSLs were developed in CAMEL. In the end, six aspects were covered by existing partner-owned DSLs, while five were developed from scratch by considering the requirements posed on the domain by the MCRM process.

Integration

In addition to the DSLs selection, some well-known challenges in DSL integration and evolution [32] had to be addressed, involving the following: (a) each DSL comes with its own abstract and concrete syntax, which makes it then difficult to join two or more DSLs, especially if they adopt different formalisms to define their syntax, (b) the DSLs to be integrated can have equivalent or overlapping concepts, which can lead to information repetition and misconceptions at the modeller side, (c) different modelling styles can be adopted leading to completely heterogeneous DSLs resulting in lack of uniformity, and (d) different DSLs might exhibit a different description granularity level, which makes it difficult to find the most appropriate detail level for integration.

To resolve these challenges, a detailed integration approach was followed that combines all DSLs to the same modelling (technical) space, description level and style by also addressing the equivalence and overlapping concepts issue. This was done by adopting the Eclipse Modeling Framework (EMF) that provides: (i) tranformation tools from various syntaxes (e.g., XML Schema) to the Ecore meta-language, (ii) semantic intra- and inter- domain validation of models using tools that enable the definition of Object Constraint Language (OCL) [33] constraints, and (iii) the production of a uniform, homogeneous concrete syntax of the CAMEL multi-DSL, using the Ecore metamodel, which follows the same modelling patterns and style. This enables modellers to rapidly specify in a similar and logical manner elements of heterogeneous DSLs. This reduces the learning curve and promotes the CAMEL usage.

The above description provides a high-level overview of the integration approach. Interested readers can find further details on the integration procedures for accomplishing a unified CAMEL language, as defined and explained in [38] and also documented in the CAMEL Technical Documentation [8].

Implementation

In addition to the rich expressiveness in defining a DSL's abstract syntax using EMF, as well as both the syntactic and semantic model validation using OCL, Eclipse offers also programmatic tools enabling the DSL developer to: (a) produce domain code out of an Ecore model, (b) produce a graphical editor for this DSL, (c) programmatically validate the DSL's models and (d) produce the DSL concrete syntax. Although the Eclipse tools allow generating a graphical tree-based editor, the feedback received from the use cases partners in PaaSage while using this editor, resulted in the conclusion that DevOps (i.e., CAMEL's main target group) are more accustomed to code-based textual editors. Hence, the Eclipse's XText language framework was used to define the CAMEL textual syntax. XText supports the automatic generation of textual editors out of the textual syntax definitions with user-friendly features, such as error highlighting, autocompletion and validation. CAMEL and its textual editor are available in PaaSage's repository [9] under the Mozilla Public License version 2.0.

Apart from the modelling adjustments in CAMEL's textual syntax, the CAMEL model importing feature was implemented. This feature enables users to exchange and re-use CAMEL models to have a better support in their modelling tasks. For example, suppose that a user needs to specify location requirements for the VM nodes of an application topology model. If no location model is reused, the user will need to manually develop a location hierarchy to model the desired locations of such VMs. However, by relying on a standardised location model that can be imported in a currently edited CAMEL model, the user can reduce the modelling effort by just selecting from the imported model the desired locations. In fact, this location model is already available and can be generated by exploiting the model importer tool available in PaaSage's repository. The model is constructed by transforming the United Nation's FAO geopolitical ontology[10] to a model conforming to the CAMEL's location sub-DSL. This model covers a location hierarchy involving the levels of continents, sub-continents and countries. Thus, it is quite sufficient to support specifying physical location requirements.

Requirements fulfillment

The design, integration and implementation steps were performed by following a process that guarantees that the

[8]CAMEL Technical Documentation—http://camel-dsl.org/documentation/
[9]PaaSage's Git Repository - https://gitlab.ow2.org/paasage/
[10]UN FAO geopolitical ontology -http://www.fao.org/countryprofiles/geoinfo/modulemaker/index.html

eight requirements described in "Requirements" section are satisfied. First, the CAMEL language follows the *type-instance* pattern [3], facilitating reusability (R_3) and the models@runtime approach (R_1). This pattern exploits two flavours of typing, namely *ontological* and *linguistic* [29], as depicted in Fig. 2. In this figure, SL (short for Small GNU/Linux) represents a reusable type of VM. 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.

Second, CAMEL follows the models@runtime approach, mapping to the R_1 requirement, as it has been designed to utilise the abstraction of provider-independent models, which are then transformed into provider-specific ones based on matching cloud capabilities with the respective requirements posed. The provider-specific models can then be evolved by the system through the adaptive provisioning of the user application by still satisfying the requirements given at the provider-independent level.

The coverage of multiple aspects, i.e., requirement R_2, is one of the cornerstones of the DSL design approach. The determination of relevant aspects enabled to produce an all-inclusive but focused DSL, which attempts to address the MCRM problem by covering only the most suitable information pieces. This enabled to discover suitable DSLs that were integrated into a coherent super-DSL, i.e., the CAMEL, to reduce its development effort and time.

Requirement R_3 is guaranteed at two levels: (a) by selecting and extending (when needed) a suitable DSL to ascertain the optimal coverage of each aspect; (b) by adopting a formalism (EMF Ecore + OCL), which enables to also cover, in an expressive manner, the semantics of the respective domain.

Separation of concerns (requirement R_4) is achieved by separating the information aspects to be covered in different CAMEL packages enabling their individual evolution. The approach to integration between DSLs enabled us to move generic or domain-specific concepts to suitable packages in the CAMEL metamodel. This allows each DSL to focus on a specific domain, thus avoiding semantic overlaps across domains.

Requirement R_5 is satisfied via the design of CAMEL and the aforementioned DSL integration process. In particular, CAMEL is designed for re-usability by separating between generic and aspect-specific concepts that can be re-used across different CAMEL sub-DSLs. For instance, a Metric (part of metric DSL) is associated with a respective Measurement (part of execution DSL) incorporated in an application execution context (i.e., deployment episode). In fact, the latter is a form of cross-referencing, also enabling the inter-domain CAMEL model validation. Apart from this, the CAMEL tools allow importing other CAMEL models. For example, standardised location models can be re-used for specifying location requirements in multiple CAMEL models.

A suitable integration level (requirement R_6) is achieved by using the right modelling technologies and employing the aforementioned DSL integration process. The followed procedure enabled to bring all DSLs into the same modelling space and integrate them into a unified DSL. The DSL exhibits the same modelling styles/patterns, while also caters for providing the same detail level, which is sufficient enough for capturing a specific domain by also keeping the respective modelling effort at an appropriate level.

The support for a textual syntax (requirement R_7) is provided by the CAMEL textual editor, which was implemented using XText and enables users to operate with CAMEL. A good effort has been spent in homogenising this syntax across different DSLs, by adopting the same modelling patterns and differentiating with respect to the default patterns automatically generated via XText. By providing user-friendly features, such as syntax highlighting and auto-completion, combined with the capability to import existing CAMEL models, the CAMEL editor

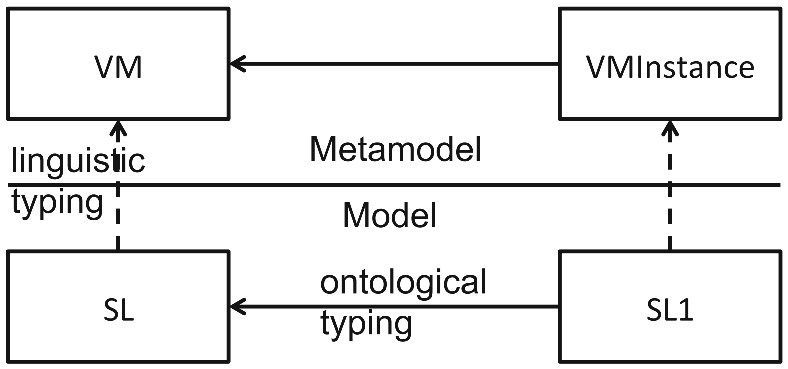


Fig. 2 Linguistic and ontological typing

enhances the user experience, exhibits a suitable usability level, and enables rapid development of CAMEL models. This has been validated in "Evaluation" section.

Finally, the re-use of DSLs (requirement R_8) was one of the design cornerstones of CAMEL. It enabled to reduce CAMEL's development effort, to cover well the respective domains in many cases, while also guaranteed the participation in this development of language engineers that have a special interest in maintaining the up-to-date versions of their DSLs within CAMEL.

The CAMEL language

In this section, an overview of CAMEL is presented first, with respect to its constituent sub-DSLs. Next, the analysis will focus, also for brevity reasons, on some core sub-DSLs, i.e., those involved in the modelling of application topologies, requirements and scalability rules, thus targeting the *DevOps* users.

In this respect, the CAMEL sub-DSLs covered in the following sub-sections include: the deployment, requirement, metric, and scalability ones. More details on other CAMEL sub-DSLs can be found in CAMEL's documentation. Also, an analysis over CAMEL's security sub-DSL can be inspected in [24].

CAMEL overview

Based on its previously analysed design method, CAMEL was realised as a super-DSL integrating multiple sub-DSLs/metamodels. Table 2 provides an overview of CAMEL's content. It explicates which are the DSLs included, supplies a list of the core domain concepts covered by these DSLs, as well as the newly added concepts, and indicates the roles of users that can be responsible to provide information for these domains.

The following user roles are expected to be involved in CAMEL model specification: (a) *DevOps*: represent users responsible for defining the application non-functional and deployment requirements along with scalability rules; (b) *Admin*: responsible for specifying: (1) the organisation model covering information about the organisation running the platform and the access control policies pertaining to that platform's usage; (2) provider models covering the offerings from both public and private cloud providers. Thus, there is a separation of concerns as *DevOps* users work at a higher abstraction level (provider-independent level), while *Admins* at a lower, more cloud provider-dependent level; (c) *System*: it maps to the platform supporting the multi-cloud application deployment, responsible for specifying and evolving provider-dependent models, as well as enriching the execution history of the application(s).

The separation of concerns between roles also defines when certain CAMEL model parts should be modelled or modified. In particular, *DevOps* and *Admins* are usually involved in the modelling phase as they provide information used mainly for supporting the subsequent phases. One exception concerns the provider models that can be updated by the *Admin* whenever changes in the offerings of respective cloud provider(s) are detected. As this change can occur at any time, this modification can span all application management phases. On the other hand, the *System* role takes care of updating the initial CAMEL model provided by the other roles during the subsequent phases of application reasoning, deployment and execution.

Some patterns can be derived from Table 2. First, the *DevOps* role is responsible to provide most of the domain-specific models in CAMEL. This is obvious as CAMEL targets mainly this role. However, while it can be argued that a lot of modelling effort will be contributed by this role, this is not necessarily the case. In particular, only two core models need always to be specified, i.e., the

Table 2 The DSLs comprising CAMEL, the core concepts they cover and the roles responsible for providing these DSLs' models

DSL	Core concepts covered	Role
Core (Top-Level)	Top model, Container of other Models, Applications	DevOps, System
Deployment	Application topology (Internal Components, VMs, Hostings, Communications)	DevOps, System
Requirement	Hardware, Security, Location, OS, Provider, QoS and Optimisation Requirements	DevOps
Provider	Provider offerings (in form of a feature-attribute model)	Admin
Security	Security controls, Attributes and mMtrics	DevOps
Metric	Metrics, Sensors, Attributes, Schedules, (measurement) Windows, Conditions	DevOps, System
Scalability	Scalability Rules, Event (Patterns), Horizontal and Vertical Scaling Actions	DevOps
Location	Physical and Cloud-specific Locations	DevOps
Organisation	Organisations, Users, Roles, Policies, Cloud/platform credentials	Admin
Execution	Execution contexts, measurements, SLO assessments, adaptation history	System
Unit	Units of measurement	DevOps
Type	Value types and Values	DevOps

deployment and requirement ones. The specification of the rest of the models depends on the application requirements. For instance, scalability rules are not needed for an application facing constant load, while security requirements do not need to be modelled when the application does not access critical organisational assets. Further, template models are already offered for basic cloud providers, metrics, units and locations which could be re-used.

Second, it is evident that there are two aspects, which concern two roles, mapping to the deployment and metric DSLs. This implements CAMEL's support for the models@runtime approach. Hence, the *DevOps* role provides the provider-independent topology and metric models, while the *System* role transforms them into provider-specific models that evolve at user application provisioning.

CAMEL in the PaaSage workflow

CAMEL per se is a modelling language and framework for cloud applications and their execution status. This modelling itself can be generic and on a level that is independent from cloud providers, e.g., describing requirements for an application to be run; on the other hand, the modelling can also be specific and describe very concretely which application components shall be run on which virtual machines on what cloud provider. Being a modelling language, CAMEL provides the means to express these scenarios, but itself does not come with any tools for manipulating the models or moving from provider-agnostic models to provider-specific models. Initially, such tools have been developed and evaluated in the PaaSage project and been enhanced in work since then. Even though this paper is about CAMEL as a language, this section describes PaaSage's MCRM framework with CAMEL at its core. We hope that this illustrates the usage of CAMEL in a larger context and helps the reader to better understand.

In the following, we focus on the application deployment and reconfiguration flow supported by the PaaSage framework. It is important to note that PaaSage has not been designed to be a cloud broker. Instead, its operation is similar to configuration management tools such as ansible and chef and its view is application-centric. In consequence, the storage of cloud credentials required for accessing cloud services is not overly critical, as the entire toolchain runs locally. Despite that, PaaSage uses encryption to store password and credentials. The use of CAMEL in cloud-broker scenarios has been investigated by the CloudSocket project [13, 18, 26], but it is out of the scope of this document.

Figure 3 illustrates the use of CAMEL in the PaaSage workflow. In this figure, *white trapezes* represent activities performed by the user, while *white rectangles* represent processes executed by the PaaSage framework. The *coloured shapes* represent modelling artifacts: the blue shapes pertain to the modelling phase, the red ones to the deployment and the green ones to the execution phase.

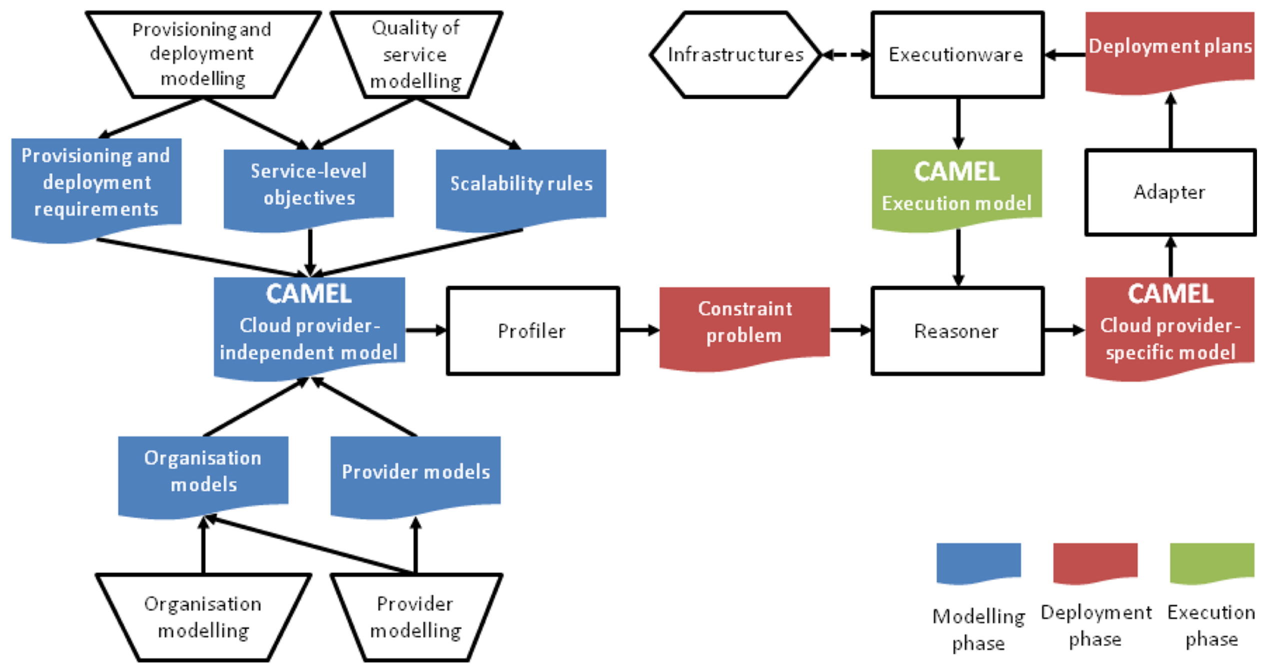


Fig. 3 CAMEL models in the self-adaptation workflow

Modelling phase

During the modelling phase, the users develop a CAMEL application model that includes three pieces of information: (a) the provider-independent topology model (PITM) specifying the types of virtual machine (VM) nodes on which the application components should be hosted; (b) the application requirements that include Service Level Objectives (SLOs) and optimization goals over quality, cost and security terms; (c) scalability rules that drive the local adaptation behaviour of the application. Apart from the CAMEL application model, users develop (i.e., organization's private cloud) or reuse CAMEL cloud provider models (e.g., Amazon, Azure, Organization's private cloud), which specify the offerings supplied by these Clouds. The provider models also cover the pricing information of the Cloud provider as well as the relative performance of its offerings.

Deployment phase

The design-time CAMEL application and provider models are then used by a reasoner to produce an application deployment plan solving a constraint problem. Application requirements are exploited to filter out cloud providers per application component, thus relying on component-specific requirements (e.g., # of cores - hardware requirements), as well as on constraints imposed at the application level (e.g., deployment cost $\leq$ € 20). The filtering dynamically generates a constraint optimization model that aims at the best VM offering per application component, by considering global optimization goals defined for the whole application (e.g., minimize application *cost* and maximize *availability*).

This optimisation model is in the CAMEL model leading to a provider-specific topology model (PSTM), covering the instance level. It defines how many instances of an application component are deployed to respective VM instances, which map to a certain VM offering in the solution. The PSTM is then exploited by the *Adapter* to create a deployment plan, which defines the acquisition of resources across different Clouds, e.g. virtual machines, and the application deployment flow, i.e., deployment of application components on these virtual machines. It is the *Executionware* that orchestrates these actions and invokes provider-specific deployment actions and creates an execution model.

Execution phase

Once the application deployment finishes, the *execution* phase starts. Initially, an execution sub-model is injected at runtime in the CAMEL model, which maintains execution-related information about the current deployment. It includes the measurements produced by the *Executionware* for the running application, plus SLO violations occurred that occurred at runtime. This model not only allows to keep track of the running application, but also to exploit its execution history to improve its deployment using the Profiler and Reasoner.

The Executionware itself is realised by the Cloudiator toolkit [6], a cross-cloud orchestration toolkit that handles the acquisition of virtual resources, deployment of application artifacts, wiring of application component instances, and monitoring of both applications and virtual resources. Cloudiator makes use of a multitude of technologies to fulfill its functionality. Yet, for the sake of acquiring virtual resources, i.e., virtual machines, it relies on the jclouds[11] library where possible [5, 12]. Other cloud platforms, e.g., Microsoft Azure, are supported through dedicated drivers.

Reconfiguration and adaptations

Both Executionware as well as Reasoner and Profiler may trigger actions that lead to changes: The Executionware monitors the quality of the application execution and compares live monitoring data against SLO thresholds set in the CAMEL model. Violations of these may lead to the executing local scaling rules whose execution leads to scale out/in of application components and hence to a change of the CAMEL execution model. On the other hand, Reasoner and Profiler continuously observe the application's execution history and current state and continuously produces new PSTMs, which are better than the currently applied one. If such a new configuration is found, the adapter generates a new deployment plan containing the difference between the current and the desired deployment that is passed on to the Executionware and enacted there. As such, a global reconfiguration loop is supported enabling to converge to an optimal application deployment, adaptable according to the current situation. Similarly, the entire process shown in Fig. 3 is triggered when the user changes the cloud provider model. This may be due to a new cloud provider being added to the model or changes in existing cloud provider models, for instance when the pricing of a provider changes, new virtual machine flavours are introduced, or the relative performance changes due to new hardware at provider side.

Both local and global reconfiguration actions are reflected in the currently applied PSTM runtime model, which enables to support the models@runtime approach, as opposed to other CMLs. In fact, the dynamic modification of the CAMEL models is performed by the system at runtime. This enables self-adaptation, i.e., the CAMEL model is "live", in contrast to other systems where such modification is manually performed at design time by the user. This is an aspect that is missing from current proprietary cloud application management systems and CMLs, that manage even single Clouds.

[11] http://jclouds.apache.org/

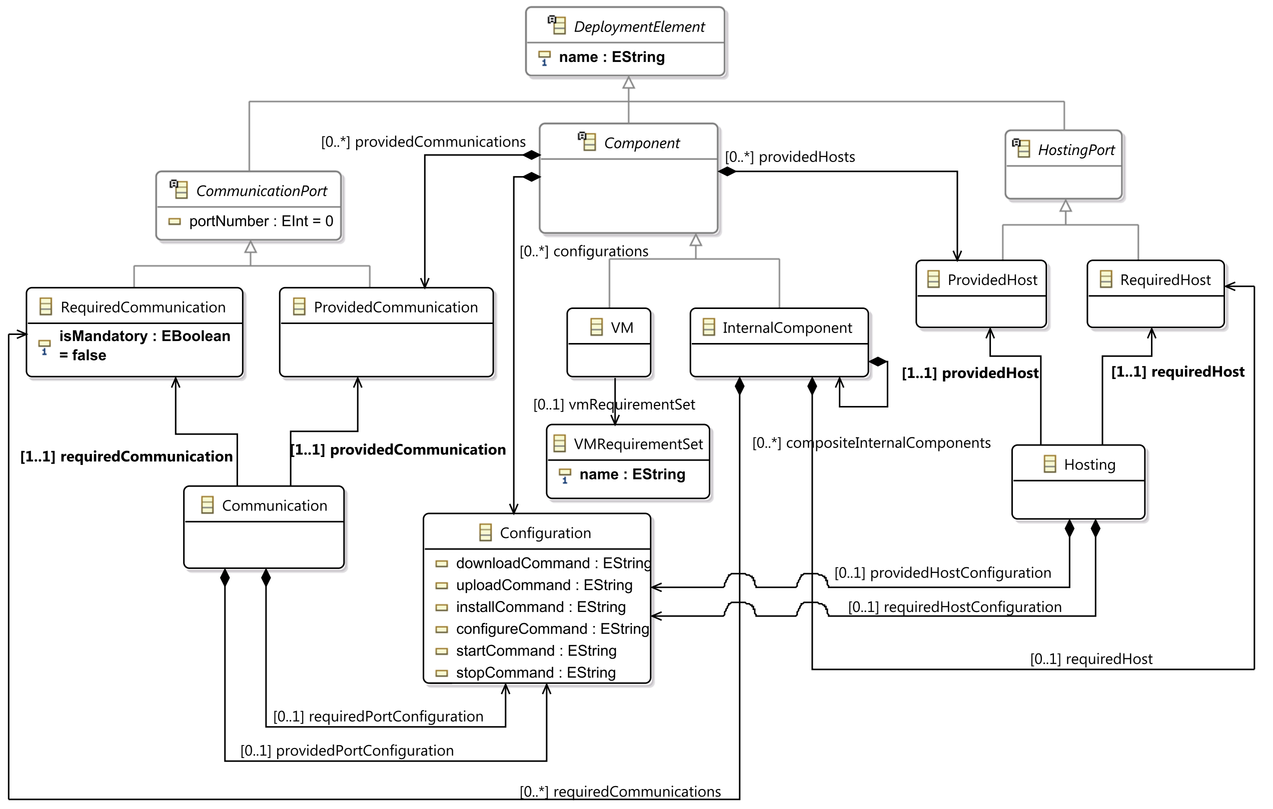


Fig. 4 The type part of the deployment metamodel

CAMEL metamodel

The CAMEL core metamodel is technically represented as an Ecore model and organised into eleven metamodels/packages. Each metamodel/package reflects a certain domain. The core package includes generic concepts, reused across different domains, as well as the CamelModel acting as a top-level container. For brevity and to limit the technical details, only the deployment, requirement, metric and scalability metamodels are introduced fully. The rest of the metamodels are briefly introduced. Readers can refer to the CAMEL Technical Documentation and CAMEL Semantics [12] for more details on the individual metamodels.

Deployment Metamodel

The deployment metamodel follows the type-instance pattern where the type part specifies a PITM while the instance part a PSTM. Figure 4 depicts the type part. The instance part is not shown as it is identical to the type part with the exception that instances (e.g., VMInstance) of type-based concepts (e.g., VM) are modelled, always pointing to their type.

The top-level entity in the deployment metamodel is DeploymentModel, i.e., a container of provider-independent deployment elements. At the type level, the basic but abstract entity is Component. Following a component-based modelling approach, this entity has a set of provided communication and required communication ports. The former enable it to communicate with other components, while the latter to host other components. It includes also a set of Configuration elements, in the form of OS-specific commands, for lifecycle management, i.e., to download, install, configure, run and stop this component.

A Component entity subsumes two component types: (1) the InternalComponent represents a software component to be deployed in the Cloud, requiring to be hosted by another Component (either InternalComponent or VM) via a HostingPort (for instance, a servlet container can host a servlet, where both are InternalComponents) and (2) the VM which acts as a host for internal components.

A Communication is established by connecting the provided and required communication ports of two components. This communication's lifecycle can also be managed via two Configuration elements. The first focuses on managing the provided, while the second the required communication port. Furthermore, a Communication has a type that

[12]CAMEL Semantics - http://camel-dsl.org/documentation/

draws the following values from the CommunicationType enumeration: (a) LOCAL: denoting that the internal components connected need to be hosted on the same VM node; (b) REMOTE: signifying that the two components should be hosted on different VM nodes; (c) ANY: denotes that the management platform is allowed to decide about the related placement of these two components, i.e., whether to co-locate them or not.

The second connector type maps to the Hosting concept, representing a hosting relation between two components: the hosted internal component and a hosting internal component or VM. Similarly to a Communication, a Hosting connects the provided and required hosting ports of the two components, while it includes two Configuration elements, each devoted to the management of one of the two hosting ports.

The VMRequirementSet includes a set of references to specific kinds of requirements that can be modelled in a requirement model, such as quantitative hardware, location or OS requirements (see Listing 2). A VMRequirementSet can be associated to a VM or to the whole DeploymentModel. In the latter case, it represents global VM requirements that must hold for the whole application topology. In the former case, it represents local VM requirements that must hold for a certain VM only, which take priority over global requirements.

Requirement metamodel

CAMEL's requirement metamodel, depicted in Fig. 5, can capture the user non-functional requirements, including hardware, quality, cost, location and security ones. It has been inspired by the WS-Agreement [1] and OWL-Q [25] languages. This metamodel includes the top-level RequirementModel concept, which can contain zero or more Requirements. Any Requirement can be either hard (see HardRequirement concept) or soft (see SoftRequirement concept). Hard requirements should be satisfied at all costs by the respective platform, while soft requirements should be satisfied on a best-fit basis.

Requirements can be grouped by using the RequirementGroup sub-concept of Requirement. A certain logical operator (AND, OR or XOR) is applied over the requirements grouped to formulate goal models, inspired by goal modelling approaches like i-star [41]. The requirement grouping enables to specify alternative service levels (SLs), defined as requirement conjunctions. This caters for a more flexible filtering of the provider space, increasing the possibility that a solution to the deployment reasoning problem can be reached.

Metric metamodel

CAMEL's scalability and metric packages rely on the SRL DSL [14, 22], enabling to specify rules supporting complex adaptation scenarios of cross-cloud applications. The metric package captures the way application monitoring can be performed and the main monitoring conditions to be evaluated. The former is specified via the Metric abstraction, while the latter by the Condition concept.

The metric metamodel (see Fig. 6) follows the type-instance pattern, an essential feature that distinguishes it

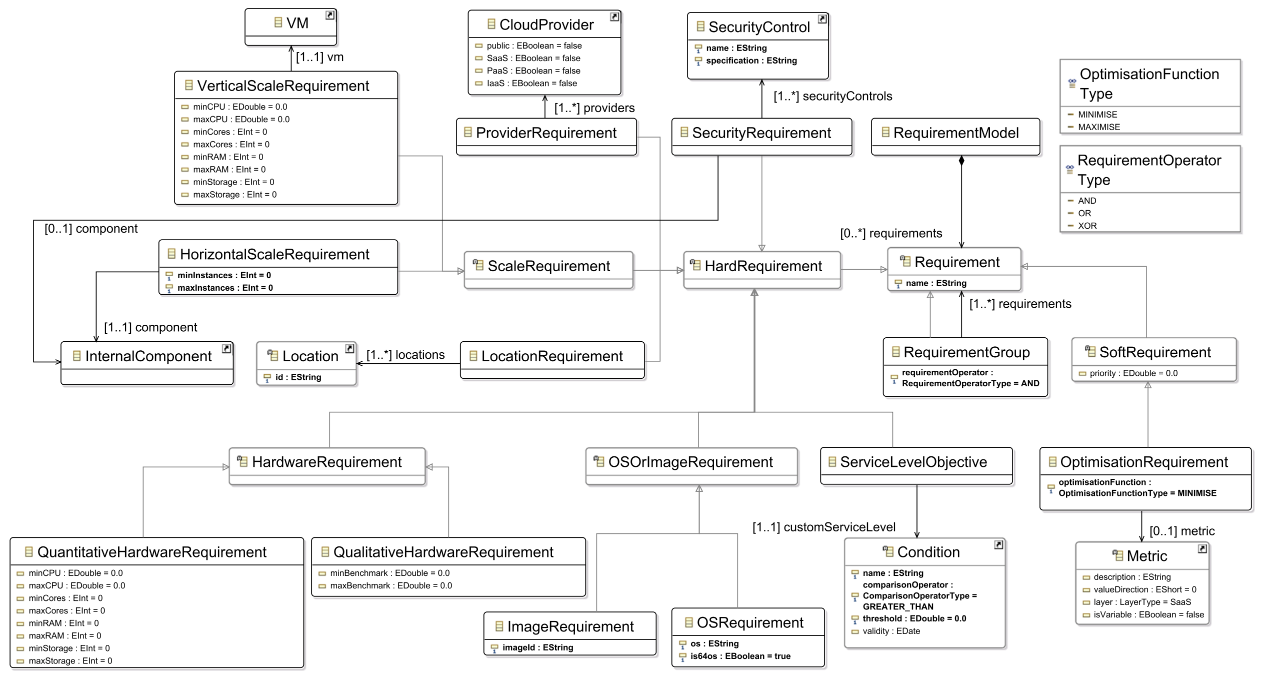


Fig. 5 The requirement metamodel

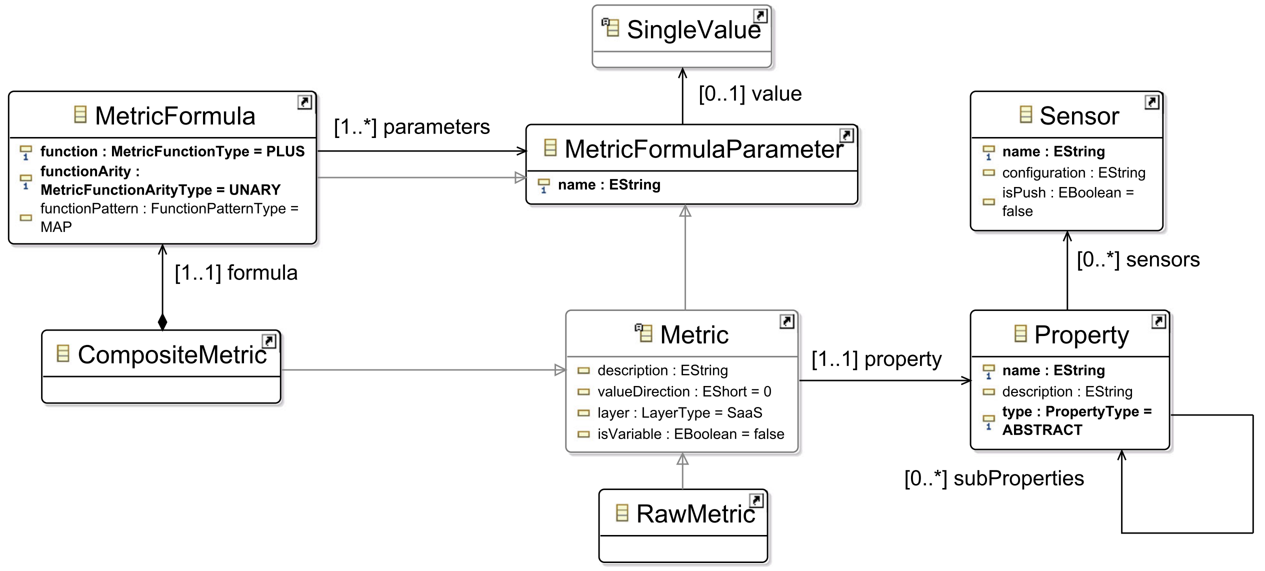


Fig. 6 The Metric concept and its hierarchy

from the state-of-the-art. This feature enables the respective (multi-cloud) application management framework to maintain and evolve the application monitoring infrastructure by following the models@runtime approach. This infrastructure should be synchronised with the changes performed on the application's PSTM model.

Scalability metamodel

SRL, apart from measurement constructs, also enables the modelling of scalability rules by including a scalability metamodel (Fig. 7). SRL is inspired by the Esper Processing Language (EPL)[13] with respect to the specification of event patterns with formulas including logic and timing operators. SRL offers mechanisms to (a) specify event patterns and associate them with monitoring data, (b) specify scaling actions, and (c) associate these scaling actions with event patterns. In the following, the main concepts defined in the scalability package are presented and analysed.

ScalabilityModel acts as a container for other scalability concepts, from which the most central is ScalabilityRule. This rule is mainly a mapping from an event to one or more scaling actions. It also specifies additional details, such as which is its developer (an Entity) and which scaling requirements (see ScaleRequirement in Section 8) should limit its triggering. Any ScalingAction is associated with a certain VM and it can be either horizontal or vertical.

Other metamodels

Provider Metamodel: The provider package of the CAMEL metamodel is based on Saloon [35–37]. Saloon is a tool-supported DSL for specifying the features of cloud providers and matching them with requirements by leveraging feature models [7] and ontologies [19]. It provides the capability to define the attributes and sub-features characterising a private or public cloud provider, e.g., the attributes characterising the virtual machine flavours provided by a private or public cloud. It also covers the costs and relative performance of individual offerings of a provider. The provider models enable matchmaking and selecting suitable cloud provider offerings, while they also unveil details specific to the application deployment.

Execution Metamodel: The execution metamodel in CAMEL has been developed from scratch with the main goal to cover the modelling of whole execution histories of multi-cloud applications. Such information can then be exploited by the management platform in order to optimise the deployment of a multi-cloud application, whether it is a new or an existing one. In this respect, an execution model is a container of different deployment episodes and enables the analysis on them to derive the added-value deployment-reasoning-targeting knowledge. Such a model not only allows to keep track of the running application but also to exploit its execution history to improve its deployment.

Security Metamodel: 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. Furthermore, an analysis over CAMEL's security DSL can be inspected in [24].

[13]https://www.espertech.com/esper/

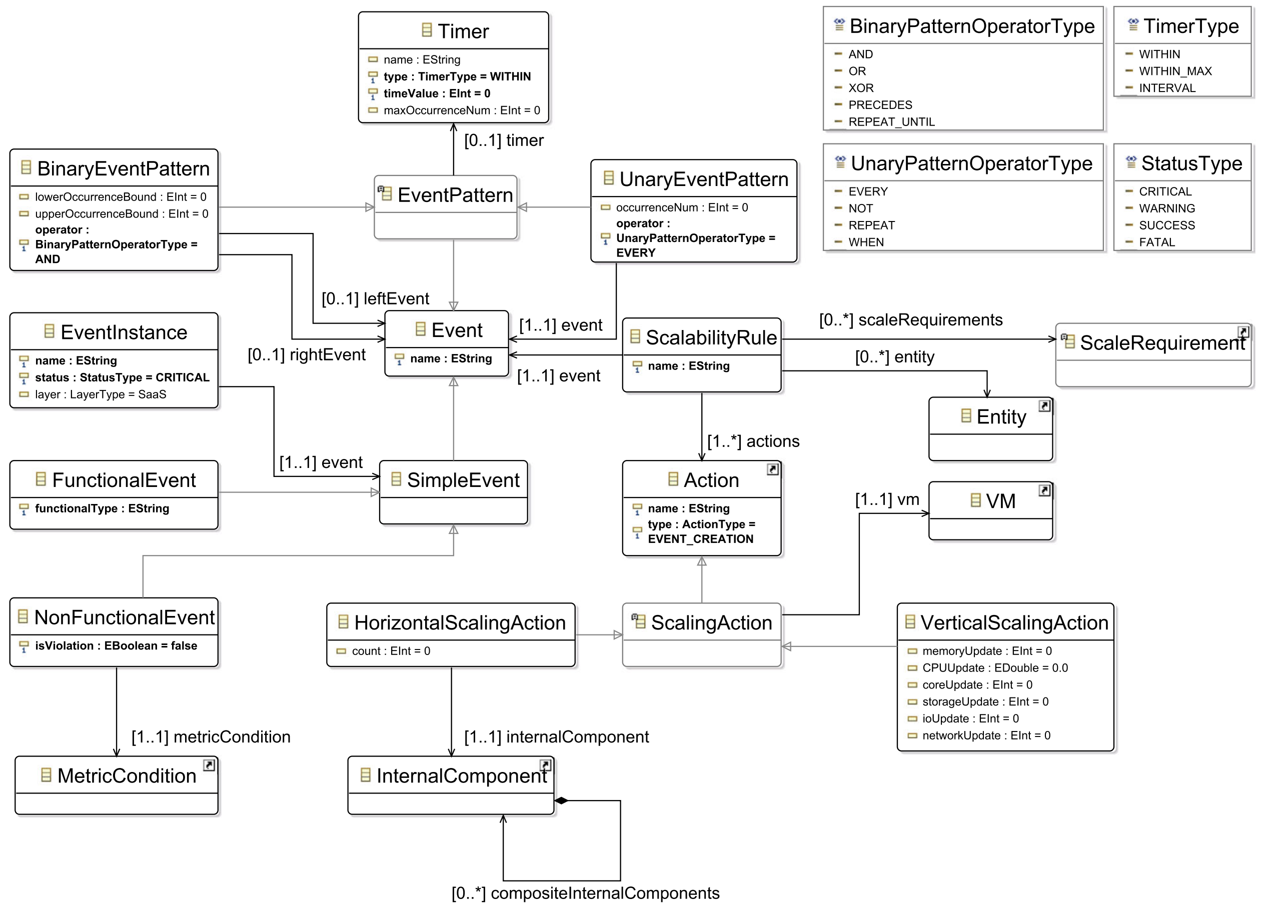


Fig. 7 The scalability metamodel

Location Metamodel: The location metamodel captures the modelling of hierarchical physical and cloud-based locations. This modelling enables specifying location requirements that can drive the filtering of the VM offering space in deployment reasoning, while also ensuring the compliance to regional or continental regulatory requirements. For example, as part of the Location an identifier is defined (e.g., ISO code for physical locations) and can be further distinguished into a GeographicalRegion and a CloudLocation.

Organisation Metamodel: The organisation package of the CAMEL metamodel is based on the organisation subset of CERIF [21]. CERIF is an EU standard for research information. In particular, the organisation package of CAMEL reuses the concepts from CERIF for specifying organisations, users, and roles. As a central part of the organisation model, the specific organisation details are defined, such as its name, contact email address, web URL.

Type Metamodel: The type metamodel is also based on Saloon [35–37]. It provides the concepts to specify value types and values used across the rest of the CAMEL models (e.g., integer, string, or enumeration).

CAMEL application: the data farming use case

The Scalarm platform's[14] [28] data farming use case allows illustrating how to specify CAMEL models conforming to CAMEL's textual syntax. We limit the presentation to those specific CAMEL sub-models presented in "The CAMEL language" section to illustrate the definition of essential properties for the use case. Readers interested in the complete concrete syntax of CAMEL should refer to [39]. The complete Scalarm CAMEL model can be downloaded from PaaSage's Git repository at OW2 [15].

Scalarm overview

Scalarm is a complete platform for conducting data farming experiments across heterogeneous computing infrastructures. It has been developed by the Akademia Grniczo-Hutnicza (AGH) University of Science and Technology. Data farming represents a methodology via which a simulation model is repeatedly executed according to an extensive parameter space such that sufficient data can be

[14]Scalarm - http://www.scalarm.com/

[15]Scalarm Model - https://gitlab.ow2.org/paasage/camel/blob/master/examples/

collected with the goal to provide an insight over the correlation between the model properties and behaviour, as well as the simulation's input parameters. Thus, Scalarm supplies to the user a set of well-known experiment design methods to generate the experiment parameter space.

Via Scalarm, each data farming experiment can be monitored, while the initial parameter space can be extended at runtime. Further, the amount of computational resources dedicated to the experiment execution can be increased such that Scalarm can scale itself based on the experiment size.

Scalarm architecture

The Scalarm architecture follows the master-worker design pattern and is depicted in Fig. 8. In this architecture, the worker part executes the simulation, while the master part coordinates the execution of the data farming experiments. Each part from the two is realized by using loosely coupled services.

In terms of the worker, the main component is the *Simulation Manager*, an intelligent wrapper for simulations capable to be deployed on different infrastructures. It implements the Pilot job concept [10] by being a specialized application that acquires computations resources to run actual simulations.

In terms of the master, (3) components are relevant: the *Experiment Manager*, *Information Service* and *Storage Manager*. The *Experiment Manager* supplies an overview about both running and completed data farming experiments, while it enables analysts to create new experiments or conduct statistical analysis on existing experiments. It is also responsible for scheduling simulations to *Simulation Managers*. The *Storage Manager* constitutes a persistence layer in the form of a service enabling other components or services to store different types of information, which include structural information about executed simulations and experiments, as well as actual simulation results, either in the form of binary or text data. Finally, the *Information Service* realizes the service locator pattern, constituting a registry of other services and components in the Scalarm system enabling the retrieval of their location.

Due to the master-worker architecture there is no immediate communication between the workers. Due to the fact that workers pull their upcoming experiments from the master, but the compute time per experiment is significantly longer than this communication (order of hours compared to orders of seconds), the application is particularly well suited for multi-cloud deployments, as there is no dependency on bandwidth and latency.

As-is and to-Be situation

Before employing the PaaSage platform, the user needs to manage the worker's resources by manually scheduling extra workers to different infrastructures. Moreover, the administrator needs to manually define scaling rules to specify scaling conditions and actions for each internal service for the master. On another note, the multi-cloud aspect and the complex scaling requirements

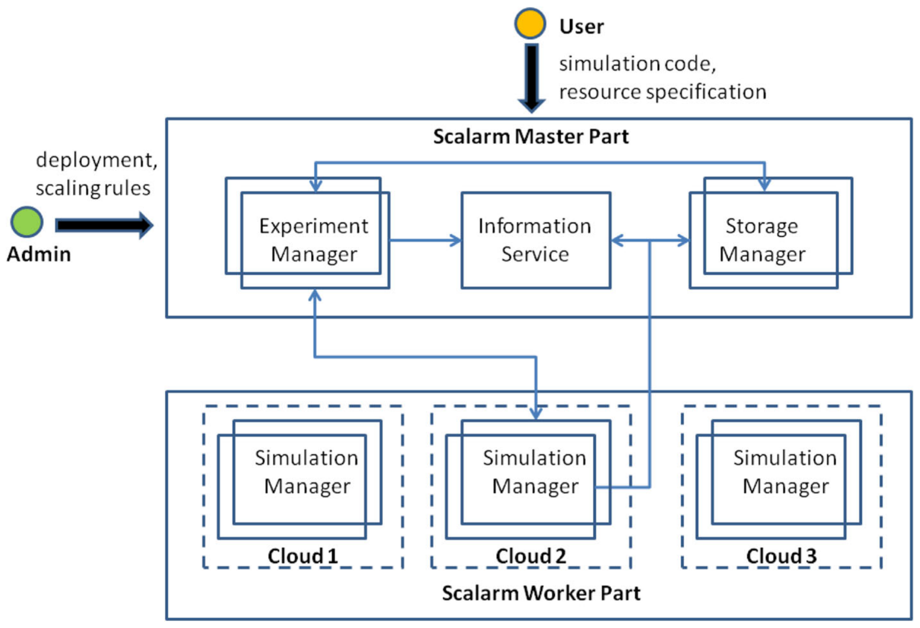


Fig. 8 Scalarm as-is architecture

of Scalarm disallow the use of widely used container orchestrators, such as Kubernetes and Docker Swarm, since they only support the definition of basic scalability rules and do not support multi-cloud deployment. As mentioned in Section 8, a Kubernetes cluster needs to be deployed manually in each cloud provider or Pipeline can be used to deploy Kubernetes clusters on major cloud providers through a unified interface before an actual application and its workload can be deployed.

By using the PaaSage platform and CAMEL, Scalarm became a fully autonomous data farming platform. This was achieved by using suitable scalability rules that enabled the automatic scaling of Scalarm components when certain conditions are met. These rules are derived by the *Reasoner* component in the PaaSage platform by considering the user's non-functional requirements. Furthermore, Scalarm initial deployment is handled by PaaSage itself so that there is no need to involve a system administrator or a user to perform scaling/deployment actions, as the PaaSage platform automatically handles all Scalarm services. Moreover, via PaaSage and CAMEL, Scalarm managed to be executed in multi-cloud environments. Multi-Cloud deployments free Scalarm from vendor lock-in and allows for fine-grained optimization of computation cost by selecting the cheapest possible cloud providers for executing large scale data experiments. The master-worker architecture of Scalarm makes it mostly insusceptible to network latency problems (which may result from highly geographically distributed deployments), and data farming does usually only requires to distribute the simulation binary - the input and output data remain reasonably small to avoid high of data transfers. Finally, by exploiting the Scalarm CAMEL model, which is publicly available, and modifying it according to specific deployments, PaaSage users can conduct data farming experiments without any prior investment in software infrastructure or the development of the right coordination software.

The scalarm cAMEL model

The key requirements for the Scalarm use case are the ability to define and modify the deployment model, as well as to specify both appropriate requirements and rules for autonomously conducting different data farming experiments. For these reasons and the need to showcase the Scalarm model definition in a clear and neat way, we present the deployment, requirement, metric and scalability models. All other models are accessible through the PaaSage repository[16].

[16]Scalarm Model - https://gitlab.ow2.org/paasage/camel/blob/master/examples/

The scalarm deployment model.

The main concepts in the deployment DSL are now exemplified via the Scalarm use case. As such, part of the deployment model is defined in Listing 1 to reduce the model length and complexity. The "..." denotes additional CAMEL elements omitted from readability.

Listing 1 Scalarm Deployment model (excerpt)

```
deployment model ScalarmDeployment {
 requirement set
     CoreIntensiveUbuntuGermanyRS {
     os: ScalarmRequirement.Ubuntu
     quantitative hardware:
         ScalarmRequirement.CoreIntensive
     location: ScalarmRequirement.
         GermanyReq
 }
 vm CoreIntensiveUbuntuGermany {
     requirement set
         CoreIntensiveUbuntuGermanyRS
     provided host
         CoreIntensiveUbuntuGermanyHost
 }
 requirement set
     CPUIntensiveUbuntuGermanyRS {
     os: ScalarmRequirement.Ubuntu
     quantitative hardware:
         ScalarmRequirement.CPUIntensive
     location: ScalarmRequirement.
         GermanyReq
 }
 vm CPUIntensiveUbuntuGermany {
     requirement set
         CPUIntensiveUbuntuGermanyRS
     provided host
         CPUIntensiveUbuntuGermanyHost
 }
 ...
 internal component ExperimentManager {
     provided communication ExpManPort {
         port: 443}
     required communication StoManPortReq
         {port: 20001 mandatory}
     required communication InfSerPortReq
         {port: 11300}
     required host
         CoreIntensiveUbuntuGermanyHostReq
     ...
 }
 internal component SimulationManager {
     required communication InfSerPortReq
         {port: 11300}
     required communication StoManPortReq
         {port: 20001}
     required communication ExpManPortReq
         {port: 443}
     required host
         CPUIntensiveUbuntuGermanyHostReq
     ....
 }
 ...
 communication
     SimulationManagerToExperimentManager
     {
     from SimulationManager.ExpManPortReq
         to ExperimentManager.ExpManPort
 }
 ...
```

```
hosting
    ExperimentManagerToCoreIntensiveUbuntu
     Germany {
    from ExperimentManager.
        CoreIntensiveUbuntuGermanyHostReq
         to
CoreIntensiveUbuntuGermany.
    CoreIntensiveUbuntuGermanyHost
}
hosting
    SimulationManagerToCPUIntensiveUbuntu
     Germany {
    from SimulationManager.
        CPUIntensiveUbuntuGermanyHostReq
        to
CPUIntensiveUbuntuGermany.
    CPUIntensiveUbuntuGermanyHost
}
...
```

As dictated by its architecture (see Fig. 8), Scalarm comprises four internal components, from which two are presented here along with their respective deployment requirements. The `ExperimentManager` has one provided communication port (443) and two required communication ports (20001 & 11300). It also requires to be hosted on a core intensive VM (i.e., hosting port). `SimulationManager` has three required communication ports (11300 & 20001 & 443) and requires to be hosted on a CPU intensive VM (i.e., hosting port). The two internal components define required hosting ports that need different VM nodes. In particular, VM nodes must be associated with a 64bit Ubuntu OS and be located in Germany, i.e., the nearest place to Poland where major cloud providers have data centres (see requirement model in Listing 2).

The scalarm requirement model.

In the above deployment model definition, the quantitative hardware requirements that must be respected by the corresponding VMs are referenced. The core intensive VM, defined in the model as `CoreIntensiveUbuntuGermany`, is associated with a quantitative requirement to incorporate 8 to 32 cores and have a memory size from 4096 to 8192 MB, while the CPU intensive VM, named as `CPUIntensiveUbuntuGermany`, must support a memory size between 8192 and 16384 MB. These requirements are actually specified (along with others) in the requirement model presented in Listing 2.

Listing 2 Scalarm Requirement model (excerpt)

```
requirement model ScalarmRequirement {
    quantitative hardware CoreIntensive {
        core: 8..32
        ram: 4096..8192
    }

    quantitative hardware CPUIntensive {
        core: 1..
        ram: 4096..8192
        cpu: 1.0..
    }
    ...
    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
    }
    ...
```

The scalarm scalability model.

Listing 3 showcases the sole scalability rule of the Scalarm application, which attempts to increase the number of instances of the *SimulationManager* component by one when the mean CPU utilisation in its corresponding VM is equal or goes above 80%.

Listing 3 Scalarm Scalability model (excerpt)

```
scalability model ScalarmScalability {
    horizontal scaling action
    HorizScaleSimulationManager {
        type: SCALE OUT
        vm: ScalarmModel.ScalarmDeployment.
    CPUIntensiveUbuntuGermany
        internal component: ScalarmModel.
    ScalarmDeployment.SimulationManager
    }

    non-functional event CPUAvgMetricNFEAny
     {
        metric condition: ScalarmModel.
    ScalarmMetric.CPUAvgMetricConditionAny
        violation
    }
    ...
    scalability rule CPUScalabilityRule {
        event: ScalarmModel.
    ScalarmScalability.CPUAvgMetricNFEAny
        actions [ScalarmModel.
    ScalarmScalability.
    HorizScaleSimulationManager]
        scale requirements [
    ScalarmRequirement.
    HorizontalScaleSimulationManager]
    }
}
```

This scalability rule, named as `CPUScalabilityRule`, maps the CPU specific event `CPUAvgMetricNFEAny` to the `HorizontalScalingSimulationManager` scaling action. It is also associated to the `HorizontalScaleSimulationManager` scale requirement (see Listing 2) denoting that the number of instances of *SimulationManager* should be at most 5, thus representing the actual upper scalability limit to hold for the scalability rule. The `HorizontalScalingSimulationManager` action indicates that the *SimulationManager* component should scale out, as hosted by the CPUIntensiveUbuntuGermany VM node, with an additional instance. On the other hand, the CPUAvgMetricNFEAny is a single non-functional event directly mapping to the violation of the CPUMetricCondition condition, as indicated in Listing 4.

The scalarm metric model

For brevity, the analysis focuses only on how the CPUAverage composite metric and its condition can be specified in CAMEL (see Listing 4. This metric condition participates in the CPUMetricSLO as indicated in Listing 2 and the CPUAvgMetricNFEAny non-functional event in Listing 3.

Listing 4 Scalarm Metric model (excerpt)

```

metric model ScalarmMetric {
    ...
    property CPUProperty {
        type: MEASURABLE
        sensors [ScalarmMetric.CPUSensor]
    }
    ...
    sensor CPUSensor {
        configuration: 'de.uniulm.omi.
cloudiator.visor.sensors.
SystemCpuUsageSensor'
        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: REDUCE
            MEAN( ScalarmModel.
ScalarmMetric.CPUMetric )
        }
    }
    ...
    raw metric context CPURawMetricContext
    {
        metric: ScalarmModel.ScalarmMetric.
CPUMetric
        sensor: ScalarmMetric.CPUSensor
        component: ScalarmModel.
ScalarmDeployment.SimulationManager
        schedule: ScalarmModel.
ScalarmMetric.Schedule1Sec
        quantifier: ANY
    }
    ...
    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
.CPUAvgMetricContextAny
        threshold: 80.0
        comparison operator: >
    }
    ...
    schedule Schedule1Min {
        type: FIXED_RATE
        interval: 1
        unit: ScalarmModel.ScalarmUnit.
minutes
    }
    schedule Schedule1Sec {
        type: FIXED_RATE
        interval: 1
        unit: ScalarmModel.ScalarmUnit.
seconds
    }
    window Win1Min {
        window type: SLIDING
        size type: TIME_ONLY
        time size: 1
        unit: ScalarmModel.ScalarmUnit.
minutes
    }
    ...
```

The CPUAverage composite metric is calculated by the Formula_Average formula, which applies the MEAN function over the CPUMetric, a raw metric computed by the push-based CPUSensor sensor, part of the PaaSage platform and especially the *Executionware* module.

CPUMetricCondition is a composite metric condition imposing that the metric refer to as CPUAverage should be less than 80%. This condition refers to the CPUAvgMetricContextAny composite metric context. This context explicates the CPUAverage metric's schedule and window, as well as that it is applied over the *Simulation-Manager* component. It also refers to the composing metric's raw metric context named as CPURawMetricContext. The CPUAverage's Schedule1Min schedule specifies that the metric's measurements will be computed repeatedly every 1 min, according to the metric's Win1Min sliding window.

CPURawMetricContext is the raw metric context for the CPUMetric. It explicates that the CPUSensor will be used to measure this metric and it is associated with the

`Schedule1Sec` schedule, which means that `CPUMetric`'s measurements will be calculated every 1 s.

Evaluation

Population

For evaluation purposes, CAMEL was exposed to different practitioners in the context of the PaaSage project use cases. Practitioners were recruited from the personnel of the organisations participating in the project that were responsible for specific use cases. 23 individuals participated in the study. In order to drive analysis of the results, using the two way ANOVA test, the participants were separated to four groups based on their MDE and cloud knowledge. MDE and cloud knowledge were selected as the two independent variables due to the need for evaluation of CAMEL against the dependent variable: i.e., usefulness or ease of use. In fact, the groups are: (i) 35% of the participants are under Group 1 with less to average knowledge of MDE and cloud ($MDE \leq 3, Cloud \leq 3$), (ii) 22% of the participants are under Group 2 with less to average knowledge of MDE and excellent knowledge of Cloud ($MDE \leq 3, Cloud > 3$), (iii) 13% of the participants are under Group 3 with excellent knowledge of MDE and less to average knowledge of Cloud ($MDE > 3, Cloud \leq 3$) and (iv) 30% of the participants are under Group 4 with excellent knowledge of MDE and with excellent knowledge of Cloud ($MDE > 3, Cloud > 3$). Hence, the research questions were defined by taking into consideration the competences of the groups. Finally, all participants completed the evaluation questionnaire, which indicates that all results are valid for analysis.

Methodology

The main aim of the evaluation was to collect practitioners' feedback regarding the use and capabilities of CAMEL, and this feedback was considered in the first evaluation steps for updating CAMEL and its modelling environment, in order to make sure that the language covers well different needs. The research study evaluation methodology is based on two factors. In specific, the evaluation results were extracted on the basis of the Technology Acceptance Model (TAM) [4, 11], where the following TAM factors were considered:

– *Perceived Ease of Use (PEU)*: the degree to which a user believes that CAMEL reduces the effort in modelling tasks.
– *Perceived Usefulness (PU)*: the degree to which a user believes that using CAMEL enhances the modelling tasks' performance.

The participants used CAMEL language and editor in the context of different business and research domains, i.e., Data Farming, Automotive Simulation, Flight scheduling, ERP, Financial Services and Human Milk Bank, and completed a questionnaire for evaluating the above TAM factors. The studied use cases are summarised in Table 3. For more information on the use cases the interested reader may refer to the PaaSage website[17]. In specific, the following steps were used for the evaluation:

– The participants were familiarised with different CAMEL versions, reported bugs, requested features, and supplied feedback to developers.
– The participants modelled their use cases scenarios with the final version of CAMEL language and editor.
– The participants assessed CAMEL features via an online questionnaire[18].

Based on the above, the final questionnaire was divided into different section, covering : usability of the CAMEL Textual Editor, CAMEL documentation, CAMEL Requirements, CAMEL Metric Model, CAMEL Deployment Model, CAMEL Scalability Rules and CAMEL Organisation Model, whereas demographic data and prior user knowledge were also collected. The most important results are assessed in "Technology acceptance" section to examine the opinion of the participants as to the usefulness and the ease of use of the CAMEL language and editor, which indicates their willingness to accept and use the new technology. The evaluation of the whole PaaSage platform, e.g., in terms of performance, is not covered in this work.

Moreover, a statistical analysis is applied on the evaluation results for reliability purposes and for detecting useful conclusions (e.g., does the MDE experience of the participants affects their opinion in terms of PU and PEU for CAMEL). Initially, the Cronback's Alpha coefficient was used for testing the reliability of the scale items. Following, a two factor ANOVA with replication was performed to examine the effect of the two independent variables (MDE and Cloud experience) on a dependent variable – PU or PEU (i.e., the test was executed twice). This test also examines whether the interaction of the two independent variables affect each other to influence either the PU or PEU dependent variable. Finally, a paired sample t-test was performed to determine whether the mean difference between two sets of observations (i.e., PU and PEU) is significant for the same population.

Reliability analysis

Cronbach's alpha was used in this work as a measure of internal consistency (i.e., reliability) for the designed instrument. This coefficient is used since it provides the capability to determine if a scale that is composed of multiple Likert questions in a survey is reliable. In specific,

[17] PaaSage use cases - https://paasage.ercim.eu/use-cases/
[18] Evaluation Questionnaire - https://goo.gl/forms/Fwr3Lc33SGqTJj832

Table 3 The PaaSage use cases

Name	Sector	Use case provider	Organisation type	Relevant application
Data farming	eScience	AGH University of Science and Technology	research	Scalarm
Automotive simulation	eScience	High Performance Computing Centre, Automotive Simulation Centre Stuttgart	research	HPC systems, e.g. Computer Aided Engineering
Flight scheduling	industrial	Lufthansa Systems	consulting, IT services	NetLine/Sched
ERP	industrial	BeWan	IT services	Multi Tenant
Financial service	industrial	University of Cyprus, IBSCY	research, IT services	Quorum
Human milk bank	public	EVRY Solutions	IT services	Human Milk Bank Project

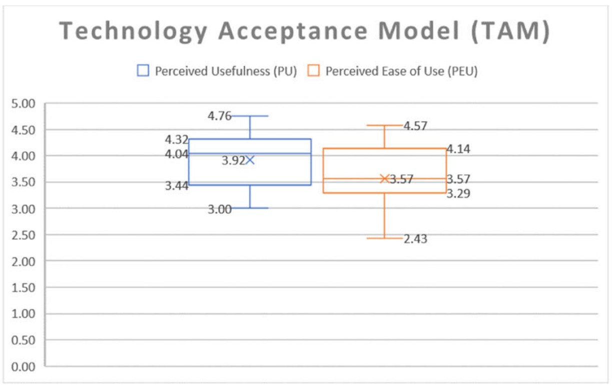
the reliability of each measurement is analogous to the extent that is a consistent measure of a concept, and the alpha coefficient is one way of measuring the strength of that consistency. It is calculated by correlating the score of each scale item (i.e., question) with the total score of a participant's observation and by comparing it to the variance of individual scale item scores. Based on the survey results the alpha coefficient was computed for PU ($\alpha = 0.93$) and for PEU ($\alpha = 0.70$). The results indicate high reliability of the scale items (i.e., questions) for PU and PEU. In fact, the literature accepts that a value of $\alpha \geq 0.70$ indicates high internal consistency (i.e., reliability) for the designed instrument. It also defines that as the number of questions increases then the reliability also increases. This actually showcases the difference between the PU and PEU alpha values, since for the model completeness attribute of PU more questions were defined for evaluating the completeness of each model (e.g., deployment, requirement).

Technology acceptance

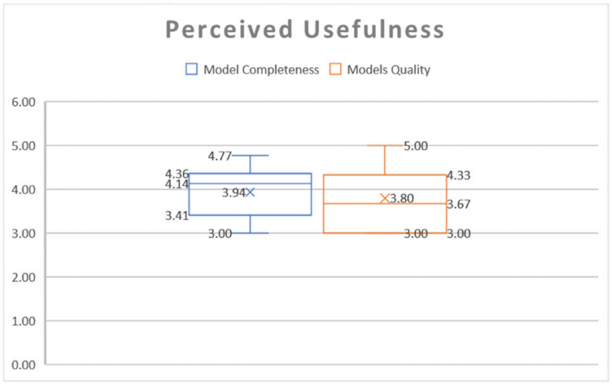
The evaluation results are first examined in terms of the two factors of PU and PEU. In specific, based on the participants responses it can be securely attested that the usefulness of CAMEL is high but the ease of use is rather low. This is evident in Fig. 9a, which shows that both the mean and median values for perceived usefulness are higher than that of perceived ease of use. Further examination of each TAM factor reveals more details as to the influence of individual attributes on perceived usefulness and perceived ease of use. In fact, for PU the two attributes examined are the models completeness and the models quality. Figure 9b presents the results from the analysis of these attributes. This indicates that both the model completeness and model quality are valued by the participants, which shows that CAMEL language covers a large and diverse set of requirements for defining complete and quality cloud management models. Moreover, the participants high scores for model quality indicates that they are satisfied also with the features provided by the CAMEL editor, e.g., code completion, syntax highlighting, error reporting.

Because of the importance of CAMEL model completeness, which is one of the main contributions of this work, the analysis was performed also at the level of individual models as presented in Fig. 9c. In fact, from the analysis of the results it is evident that the participants rate higher the deployment, scalability and requirements models, while the metric model is evaluated lower. This can be attributed to the fact that in most use cases simple metric models were defined using single metrics, such as CPU utilisation. Nevertheless, more complex models and composite metrics can be defined, but the platform limitation is that it only supports CPU and RAM sensors. This means that for complex monitoring of VM resources the appropriate sensors should be manually implemented and deployed in the platform as Java classes.

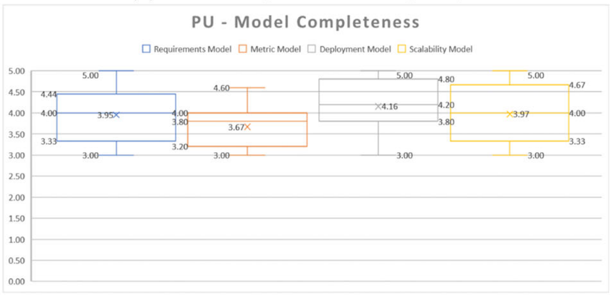
Moreover, the PEU of the CAMEL language and editor was evaluated based on the attributes of effectiveness and learning curve. The mean score for effectiveness (see Fig. 9d) is the lowest one recorded from the entire set of attributes. This can be attributed, by examining the context and results of the individual questions for effectiveness. In specific, the installation and use of the CAMEL editor and language is rated higher than the user-friendliness and model creation related questions. These outcomes are further supported by the learning curve's higher score. In fact, the participants gave a low score to the easiness of learning how to use the CAMEL language and textual editor, but the extensive documentation provided for CAMEL is highly valuable to the users as indicated by the high scores on user-support provided by the documentation. This strongly suggests that the users find the detailed documentation as a key aspect that can minimise the learning curve.



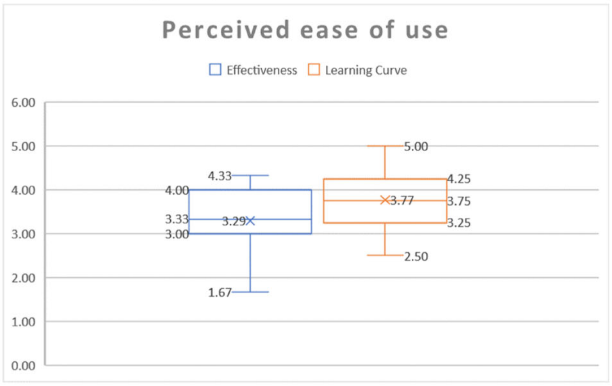
(a) Technology Acceptance Factors



(b) PU - Completeness and Quality



(c) Completeness of CAMEL models



(d) PEU - Effectiveness and Learning

Fig. 9 Evaluation Results. **a** Technology Acceptance Factors. **b** PU - Completeness and Quality. **c** Completeness of CAMEL models. **d** PEU - Effectiveness and Learning

Finally, a paired samples t-test was performed to confirm the results displayed in Fig. 9. This type of test is a statistical procedure used to determine the mean difference between two sets of observations for the same sample size. In this work it is used to confirm that there is a significant mean difference between the participants observations for PU and PEU. Like many statistical procedures, the paired sample t-test has a null hypothesis that assumes that the true mean difference between the paired samples is zero. Statistical significance is determined by looking at the $p - value$, which defines the probability of testing the survey results under the null hypothesis. Executing the t-test on the scores of the same set of participants on PU and PEU resulted to a $p - value = .005$. Hence, the computed p-value is less than or equal to the commonly accepted significance level ($p - value \leq .05$), which means that the null hypothesis (PU and PEU mean difference is zero) can be rejected. This indicates a statistically significant difference between the users opinions. This practically means that users find CAMEL highly useful, but believe that it can be improved in terms of ease of use.

Group-Based analysis

On the basis of the technology acceptance model the perceived usefulness and perceived ease of use factors are evaluated in this survey study. An important aspect that requires further analysis is what are the opinions of participants in accordance to the groups defined in this study. In specific, it maybe expected that participants under Group 1 that have low to average knowledge of MDE and Cloud (i.e., $MDE \leq 3, Cloud \leq 3$) would provide a lower score to perceived ease of use of the CAMEL language and editor. Therefore, in order to detect if there are differences in the observations of participants across groups a two-way ANOVA statistical test was performed. This kind of test compares the mean differences between groups that have been split based on two independent variables (i.e., MDE and Cloud experience of participants). The primary purpose of this test is to understand if there is an interaction between the two independent variables on the dependent variable (i.e., PU or PEU - test was executed twice). In this work, the two-way ANOVA was performed to understand whether there is an interaction between MDE and Cloud knowledge, which has an effect on the PU or PEU evaluation scores.

Table 4 presents the results of the two-way ANOVA test, i.e., for PU and PEU. The ANOVA was used to test the following null hypotheses:

(i) H_1 - **The means of observations grouped by one factor (i.e., MDE knowledge level) are the same**. On the basis of the tests executed, H_1 cannot be rejected for PU since the $p - value = .53$ and also it cannot be rejected for PEU since the $p - value = .20$. The values are way higher

Table 4 Two-way ANOVA analysis results

PU - MEANS				PEU - MEANS			
	Cloud ≤ 3	*Cloud* > 3			*Cloud* ≤ 3	*Cloud* > 3	
MDE ≤ 3	3.99	3.76	**3.88**	*MDE* ≤ 3	3.7	3.23	**3.46**
MDE > 3	4.23	3.82	**4.03**	*MDE* > 3	4.10	3.43	**3.76**
	4.11	3.79			3.90	3.33	

than the critical $p - value = .05$, which indicates that the means for the group $MDE \leq 3$ and the group $MDE > 3$ can be practically considered the same. This is because no statistical difference is observed from the sample, in terms of the participants opinions for the two groups. A possible explanation is that the participants are highly knowledgeable of the model-driven CAMEL language and editor due to their involvement in the research project.

(ii) H_2 - **The means of observations grouped by the other factor (i.e., Cloud knowledge level) are the same**. H_2 cannot be rejected for PU since the $p - value = .19$, but it can be rejected for PEU since the $p - value = .002$. The calculated value for PEU is lower than the critical $p - value = .05$, which indicates that the means for the group $Cloud \leq 3$ and the group $Cloud > 3$ are different. This is because a statistical difference is observed from the sample, in terms of the participants opinions for the two groups. An assumption that can be made in terms of this result is that cloud experts are able to think ahead and consider the complexity involved in defining a model for a complex cloud deployment and adaptation scenario.

(iii) H_3 - **There is no interaction between the factors (i.e., MDE and Cloud knowledge level)**. Finally, H_3 cannot be rejected for PU since the $p - value = .71$ and also it cannot be rejected for PEU since the $p - value = .66$. The values are way higher than the critical $p - value = .05$, which indicates that the means for the intersection groups (e.g., $MDE \leq 3$, $Cloud \leq 3$) can be practically considered the same since no statistical difference is observed from the sample. Hence, the participants opinions for the four intersection groups have a strong similarity.

Based on the above group-based statistical analysis it is strongly suggested that MDE knowledge level does not influence the observations of participants in terms of the factors of PU and PEU, while the Cloud knowledge level has an effect on the participants observations for the PEU. Finally, the claim can be made that the interaction between MDE and Cloud knowledge has no effect on the participants observations for both PU and PEU.

Threats to validity

In terms of *external validity* —i.e., the extent to which the conclusions can be generalised, the selected use cases cover a wide spectrum of identified aspects of self-adaptive cross-cloud applications. However, extending the evaluation of CAMEL to other scenarios, environments, or even demographics may alter the findings. *Internal validity*, i.e., the extent to which the conclusions based on a study are warranted is not affected, since the data are unambiguous. In terms of *construct validity*, i.e. the degree to which a test measures what it claims, is not affected, since all questions were carefully prepared to cover all capabilities of CAMEL and its textual editor. Finally, the small sample size ($N = 23$) and the fact that the participants were part of the PaaSage project are perhaps the greatest threat to the validity of the results. For this reason different statistical analysis test were performed for checking the reliability of the survey results (i.e., Cronbach's Alpha) and for cross-checking the validity of the conclusions (i.e., paired t-test), e.g., participants find CAMEL more useful and not that easy to use. Finally, ANOVA tests were performed to conclude if MDE and Cloud knowledge level affects the results.

Related work

In the following, the CAMEL language is compared with related work. The focus is mainly on CMLs that specialize on cloud computing and not generic languages that might cover one or more aspects relevant to MCRM. Such languages should also have the right abstraction level, this being able to cover multiple and not just one cloud. In this respect, cloud-specific languages, such as CloudFormation, which are tight to a certain cloud, as well as too detailed and technical ones are excluded from the analysis.

Comparison criteria

In the following, six comparison criteria are defined to evaluate the CMLs focusing on their usefulness, usability, and self-adaptation. The *abstract syntax* and *aspect coverage*, *delivery model support*, and *models@run-time support* reflect the usefulness of the language; *concrete syntax* and *integration level* reflect the usability; and *models@run-time support* also reflects the self-adaptation.

Abstract syntax. A DSL's abstract syntax describes the set of concepts, their attributes and relations, plus the rules for combining them to specify valid statements conforming to this syntax. The abstract syntax can be defined using formalisms with different capabilities. For instance, XML Schema is suitable for tree-based structures, while

MOF-based formalisms are more suited for graph-based structures, offer better tool support and are better integrated with constraint languages like OCL. This criterion identifies the formalism used by a CML. Its evaluation spans the values of "XML Schema" and "MOF".

Concrete syntax. A concrete syntax describes the textual/graphical notation rendering the abstract syntax concepts, their attributes and relations. The concrete syntax can be defined using notations that have a trade-off between the syntax intuitiveness and effectiveness. For instance, a textual syntax may be less intuitive but more effective than a graphical syntax. This criterion is used to identify the notations supported by a DSL. Its evaluation spans the values of "XML", "txt" (textual), and "gra" (graphical).

Aspect coverage. A language may cover multiple aspects within the same or across multiple domains. For instance, in CAMEL the life cycle of cross-cloud applications is specified using 11 aspects. This criterion reflects how many of these aspects are covered by a language. Its evaluation spans the values of "low" if the DSL covers at most three aspects, "medium" if it covers at most six aspects, and "high" otherwise.

Integration level. A DSL that covers multiple aspects may provide different integration levels, especially when these aspects include similar or equivalent concepts. The integration solution must: (a) join equivalent concepts and separate similar ones into respective sub-concepts; (b) homogenise the remaining concepts at the same granularity level; (c) enforce a uniform formalism and notation for the abstract and concrete syntaxes; (d) enforce model consistency, correctness, and integrity. Each of these steps is a precondition to the following one and requires an increasing amount of effort. This criterion reflects how many steps have been adopted to integrate the sub-DSLs. Its evaluation spans the values of "low" if the sub-DSLs 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. The last evaluation value maps to sub-DSL independence that leads to the following disadvantages: (a) it raises the DSL complexity, since each sub-DSL has its own abstract and concrete syntax; (b) it steepens the learning curve and increases the modelling effort for the same reason; (c) it leads to the modelling duplication for similar or equivalent concepts; (d) 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 (e.g., IaaS and PaaS). Thus, a language for specifying the life-cycle of such application should support concepts for every cloud delivery model. As such, this criterion attempts to examine this. Its evaluation spans the values of "IaaS", "PaaS" and "SaaS".

Models@run-time support. As indicated in "Introduction" section, models@run-time [9] can enable the automatic provisioning of multi-cloud applications and can be implemented using the type-instance pattern [3]. In CAMEL, the type-instance pattern was implemented in the deployment and metric aspects. In the deployment aspect, it allows to automatically adapt the component- and VM instances in the deployment model based on scalability rules (e.g., scale out a Scalarm service and its underlying VM). In the metric aspect, the deployment adaptation is reflected also on the monitoring infrastructure. This criterion reflects how many of these aspects within a CML implement the type-instance pattern. Its evaluation spans the values of "deployment" and "metric".

Analysis

Table 5 shows the comparison results for the DSLs based on the aforementioned criteria. As it can be seen below, CAMEL scores best in all criteria. Its superiority is highlighted in terms of the aspect coverage and integration level criteria, plus its better support to different kinds of cloud services and to the specification of different type-instance models focusing both on the deployment and monitoring aspects. Thus, the claim that CAMEL does advance the state-of-the-art in cloud application modelling and MCRM can be validated. The coverage of PaaS and SaaS services has been recently introduced in CAMEL via its extension in the CloudSocket project.

The key CAMEL competitors are the Arcadia Context Model, StratusML and more recently CloudMF. The first has been included, due to its good aspect coverage which does not, however, go to a sufficient level of detail. The second, due to its high DSL integration level, which is mainly the outcome of following a similar integration approach as in CAMEL. However, the main differentiation is that less integration effort has been put in StratusML, due to the generation of all DSLs from scratch and the minimalistic size of the overall language, containing around 60 concepts. StratusML does also support the modelling of semantic domain validation rules. However, also witnessed by its small size, this language is not expressive and extensive enough, not going to an appropriate level of detail in the aspects covered. Furthermore, the coverage of other aspects is missing. CloudMF is the only CML that supports deployment and metric in terms of the models@runtime support. In specific, it provides a domain-specific language for specifying the provisioning and deployment of multi-cloud applications, as well as an adaptation DSL implemented though as Java plain objects, offering a models@run-time environment for the continuous provisioning, deployment and adaptation of applications. Finally, CloudMF presents a medium aspects coverage with a minimal set of concepts and a low integration level as a result.

Table 5 Cloud Languages Comparative Analysis

Language	Abstract Syntax	Concrete Syntax	Aspect Coverage	Integration Level	Delivery Model Support	Models@run-time Support
Reservoir OVF Extension (2009)	XML Schema	XML	low	N/A	IaaS	N/A
Optimis OVF Extension (2010)	XML Schema	XML	medium	N/A	IaaS	N/A
Vamp (2011)	XML Schema	XML	low	N/A	IaaS	N/A
4CaaSt Blueprint Template (2011)	XML Schema	XML	low	N/A	IaaS, PaaS	N/A
TOSCA (2013)	XML Schema	XML, txt	medium	medium	IaaS, PaaS	N/A
Provider DSL [40] (2014)	MOF	XML, gra	low	medium	IaaS	N/A
GENTL (2014)	MOF	gra, XML	low	N/A	IaaS	N/A
ModaCloudML (2014)	MOF	XML, gra, txt	medium	low	IaaS, PaaS	deployment
CAML (2014)	MOF	gra	medium	medium	IaaS	N/A
CAMEL (2014)	MOF	XML, gra, txt	high	high	IaaS, PaaS, SaaS	deployment, metric
ARCADIA Context Model (2015)	XML Schema	XML	high	medium	IaaS	deployment
StratusML (2015)	MOF	XML, gra	medium	high	IaaS	deployment
CloudMF (2018)	MOF	XML, gra	medium	low	IaaS, PaaS	deployment, metric

TOSCA and CAML come in the third place. In our view, TOSCA is not a competitor to CAMEL. It is rather a standard that could benefit from CAMEL based on the following directions: (a) coverage of additional domains not captured by TOSCA; (b) support for the type-instance pattern and thus models@runtime. By following the second direction, there is some integration work currently being conducted in form of a TOSCA interest group attempting to bring the PSTM part of CAMEL deployment metamodel into TOSCA.

With the exception of TOSCA, the other three languages (i.e., StratusML, Arcadia Context Model and CAML) do not have a good community support. This is evident from the fact that StratusML has been developed from a university group, while the other two languages have been developed within certain European projects but their support seems to be discontinued. On the other hand, CAMEL undergoes constant evolution and some extensions have been already performed on it, like the aforementioned PaaS/SaaS features, while others are currently in development or planned. As such, CAMEL will be further optimised (e.g., Melodic EU H2020 Big-Data Cloud project), by also attempting to adopt some interesting modelling features from these languages.

As the languages are presented in a chronological order in the comparison table, some interesting time-based patterns can be inferred from this table:

- With the exception of Arcadia Context Model, most recent languages rely on MOF for their abstract syntax. Maybe this can be explained partly due to the use of the language in a model-driven management framework and due to the various advanced tools available for MOF-based languages that assist in their rapid development.
- Coupled with the first finding is the fact that the most recent languages do provide support for the production of graphical/textual models according to the language's concrete syntax. This enables then to move from the cumbersome XML-based to a more human-readable form, which also makes the models more concise and easier to be edited/manipulated.
- Most recent DSLs do cater for the models@runtime approach, thus providing better support for the adaptive provisioning of multi-cloud applications, with CAMEL and CloudMF being the only ones that support both deployment and metric. This means that they do not only support the adaptation of the application and VM instances in the deployment model based on scalability rules, but they cater so that the adaptation is reflected also on the monitoring infrastructure.

In this respect, based on these findings, both the design requirements and choices made by the CAMEL developers can be validated, as the exploitation of Eclipse EMF & Ecore enabled CAMEL to be rapidly developed and have the right modelling tools supporting its continuous evolution, while the models@runtime support enabled CAMEL to satisfy a quite recent, in its acknowledgement, but critical modelling feature.

Conclusions & future work

This article has explained the development and implementation of an innovative multi-DSL language called CAMEL, which advances the state-of-the-art by integrating DSLs covering all suitable aspects required for MCRM. The core parts of this DSL were analysed by also utilising a running use case drawn from the PaaSage project, the actual development space of CAMEL. CAMEL is also accompanied by a textual editor, covering its concrete syntax and targeting mainly DevOps users, which exhibits some nice features like syntax and error highlighting as well as auto-completion.

Both CAMEL and its textual editor were evaluated via a user study involving well-qualified participants from use case partner organisations in PaaSage. The evaluation results show that the editor's usability is appropriate and that CAMEL covers well its respective domains. Some interesting suggestions were also supplied, currently considered in the development of the forthcoming version of CAMEL. CAMEL is being continuously evolving due to its active community that spans at least three organisations: SINTEF, University of Ulm and ICS-FORTH. This has been evident through corresponding extensions that have been performed on it in the context of European projects that succeeded PaaSage. Two examples of these projects are defined below.

CloudSocket targeted the development of a platform supporting the design and adaptive provisioning of business-process-as-a-service (BPaaS) services. In that project, two main extensions of CAMEL have been achieved: (a) support for PaaS and SaaS modelling; (b) modelling of advanced adaptation rules [27] that map event patterns to adaptation workflows incorporating level-specific adaptation actions (e.g., scaling and service replacement ones). Melodic aims to support big data application management. CAMEL is at the core of Melodic, which attempts to build upon the PaaSage platform to provide support to this application kind. As such, CAMEL is planned to be enhanced to cover the big data aspect. As Melodic is a formal PaaSage successor, also the improvements over the user survey suggestions will be included in the forthcoming CAMEL release. That release is also planned to be enhanced with CAMEL extensions from other projects, like CloudSocket. This will then map in producing an even more complete and extensive DSL, also broadening its application scope. Hence, the CAMEL community will continue its effort in optimising CAMEL and further extending it, possibly via participating in forthcoming projects that guarantee the respective funding needed.

Abbreviations

CAMEL: Cloud application modelling and execution language; CML: Cloud modelling language; DSL: Domain specific language; EMF: Eclipse modelling framework; IaaS: Infrastructure as a service; MCRM: Multi-cloud resource management; MDE: Model driven engineering; PaaS: Platform as a service; PITM: Provider independent topology model; PSTM: Provider specific topology model; OCL: Object constraint language; SaaS: Software as a service; SL: Service level; SLO: Service level objective; SRL: Scalability rules language; TOSCA: Topology and orchestration specification for cloud applications; XML: Extensible metadata language; VM: Virtual machine

Acknowledgements
Special thanks to the PaaSage project partners for their individual contributions to CAMEL and the PaaSage platform.

Authors' contributions

AA defined the evaluation framework, executed the survey and produced the evaluation results including the statistical analysis. AA written the evaluation part of the manuscript and coordinated/refined/merged the different sections/contents of this manuscript. KK and AR contributed to the definition, design and implementation of CAMEL and contributed most of the content on the CAMEL DSLs. GK assisted in the formulation of the evaluation framework, the execution of the survey and the results of the evaluation, and revised the evaluation section content. JD contributed mainly to the implementation of CAMEL (in particular in respect with the interaction with PaaSage components) and contributed/reviewed the content on the CAMEL DSLs sections in the manuscript. MO contributed to the definition and realisation of the CAMEL scalarm use case and the writing of the use case section in the manuscript. DS and FG assisted mainly to the implementation of CAMEL (in particular in respect with the interaction with PaaSage components) and contributed/reviewed the content on the CAMEL DSLs sections in the manuscript. NN and DR assisted in the definition, design and implementation of CAMEL. GP assisted in the definition of the evaluation framework and revised the relevant section in the manuscript. All authors have reviewed the complete manuscript and provided comments and suggestions that were taken into consideration by AA to finalise the paper.

Authors' information

CAMEL is an innovative multi-DSL language, which advances the state-of-the-art by integrating and extending DSLs covering all suitable aspects required for multi-cloud resources management. It addresses both design time and model@runtime aspects and continues to evolve as a language to address the many different facets of multi-cloud resources management. In specific, it continues currently to evolve as part of the MELODIC H2020 project to enable big data-aware application deployments on geographically distributed and federated cloud infrastructures.

Funding
CAMEL was defined in the EU PaaSage project that was funded from the European Union's FP7 Research and Innovation Programme under the topic ICT-2011.1.2 - Cloud Computing, Internet of Services and Advanced Software Engineering with Grant Agreement No 317715.

Availability of data and materials

Survey Data and Evaluation results used in this work can be found here: http://tiny.cc/pmqibz Relevant documentation for CAMEL can be found here: http://camel-dsl.org/documentation/ and the code/models repositories can be found here: https://gitlab.ow2.org/paasage/ under the CAMEL repository. No additional data have been used in this study.

Competing interests

No competing interests between authors for this manuscript.

Author details
[1]Frederick University, Nicosia, Cyprus. [2]ICS-FORTH, Heraklion, Crete, Greece. [3]PwC Consulting, Oslo, Norway. [4]University of Cyprus, Nicosia, Cyprus. [5]Ulm University, Ulm, Germany. [6]AGH, Warsaw, Poland. [7]SINTEF, Oslo, Norway. [8]LIFL, Inria Lille, France.

Received: 27 March 2019 Accepted: 22 August 2019
Published online: 16 December 2019

References

1. Andrieux A, Czajkowski K, Dan A, Keahey K, Ludwig H, Nakata T, Pruyne J, Rofrano J, Tuecke S, Xu M (2010) Web Services Agreement Specification (WS-Agreement). https://www.ogf.org/ogf/doku.php/documents/documents
2. Andrikopoulos V, Binz T, Leymann F, Strauch S (2013) How to adapt applications for the cloud environment. Computing 95(6):493–535. https://doi.org/10.1007/s00607-012-0248-2
3. Atkinson C, Kühne T (2002) Rearchitecting the UML infrastructure. ACM Trans Model Comput Simul 12(4):290–321

4. Bagozzi RP, Davis FD, Warshaw PR (1992) Development and Test of a Theory of Technological Learning and Usage. Hum Relat 45:659–686
5. Baur D, Domaschka J (2016) Experiences from building a cross-cloud orchestration tool. In: Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms, CrossCloud '16. ACM, New York. pp 4:1–4:6. http://doi.acm.org/10.1145/2904111.2904116
6. Baur D, Seybold D, Griesinger F, Masata H, Domaschka J (2018) A provider-agnostic approach to multi-cloud orchestration using a constraint language. In: Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid '18. IEEE Press, Piscataway. pp 173–182. https://doi.org/10.1109/CCGRID.2018.00032
7. Benavides D, Segura S, Cortès AR (2010) Automated analysis of feature models 20 years later: A literature review. Inf Syst 35(6):615–636
8. Bergmayr A, Breitenbücher U, Ferry N, Rossini A, Solberg A, ManuelWimmer Kappel G, Leymann F (2018) A Systematic Review of Cloud Modeling Languages. ACM Comput Surv 51:1–38
9. Blair G, Bencomo N, France R (2009) Models@run.time. IEEE Comput 42(10):22–27
10. Chiu PH, Potekhin M (2010) Pilot factory - a Condor-based system for scalable Pilot Job generation in the Panda WMS framework. J Phys Conf Ser 219(6):062,041. http://stacks.iop.org/1742-6596/219/i=6/a=062041
11. Davis FD (1989) Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. MIS Q 13(3):319–340
12. Domaschka J, Baur D, Seybold D, Griesinger F (2015) Cloudiator: A Cross-Cloud, Multi-Tenant Deployment and Runtime Engine. In: SummerSOC 2015: 9th Workshop and Summer School On Service-Oriented Computing 2015. https://www.summersoc.eu/wp-content/uploads/2015/07/2.6.domaschka_cloudiator_summesoc2015.pdf
13. Domaschka J, Griesinger F, Seybold D, Wesner S (2017) A cloud-driven view on business process as a service. In: CLOSER. pp 739–746. https://doi.org/10.5220/0006393107670774
14. Domaschka J, Kritikos K, Rossini A (2015) Towards a Generic Language for Scalability Rules. In: Ortiz G, Tran C (eds). Advances in Service-Oriented and Cloud Computing—Workshops of ESOCC 2014, Communications in Computer and Information Science, vol. 508. Springer. pp 206–220. https://doi.org/10.1007/978-3-319-14886-1_19
15. Ferry N, Chauvel F, Rossini A, Morin B, Solberg A (2013) Managing multi-cloud systems with CloudMF. In: Solberg A, Babar MA, Dumas M, Cuesta CE (eds). NordiCloud 2013: 2nd Nordic Symposium on Cloud Computing and Internet Technologies. ACM. pp 38–45. https://doi.org/10.1145/2513534.2513542
16. Ferry N, Rossini A, Chauvel F, Morin B, Solberg A (2013) Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In: O'Conner L (ed). CLOUD 2013: 6th IEEE International Conference on Cloud Computing. IEEE Computer Society. pp 887–894. https://doi.org/10.1109/cloud.2013.133
17. Ferry N, Song H, Rossini A, Chauvel F, Solberg A (2014) CloudMF: Applying MDE to Tame the Complexity of Managing Multi-Cloud Applications. In: Bilof R (ed). UCC 2014: 7th IEEE/ACM International Conference on Utility and Cloud Computing. IEEE Computer Society. pp 269–277. https://doi.org/10.1109/ucc.2014.36
18. Griesinger F, Seybold D, Wesner S, Domaschka J, Woitsch R, Kritikos K, Hinkelmann K, Laurenzi E, Iranzo J, González RS, Tuguran CV (2017) Bpaas in multi-cloud environments - the cloudsocket approach. In: European Space Projects: Developments, Implementations and Impacts in a Changing World - Volume 1: EPS Porto 2017. INSTICC, SciTePress. pp 50–74. https://doi.org/10.5220/0007901700500074
19. Gruber TR (1993) A translation approach to portable ontology specifications. Knowl Acquis 5(2):199–220
20. Horn G, Skrzypek P (2018) Melodic: utility based cross cloud deployment optimisation. In: 2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA). IEEE. pp 360–367. https://doi.org/10.1109/waina.2018.00112
21. Jeffery K, Houssos N, Jörg B, Asserson A (2014) Research information management: the CERIF approach. IJMSO 9(1):5–14
22. Kritikos K, Domaschka J, Rossini A (2014) SRL: A Scalability Rule Language for Multi-Cloud Environments. In: Guerrero JE (ed). CloudCom 2014: 6th IEEE International Conference on Cloud Computing Technology and Science. IEEE Computer Society. pp 1–9. https://doi.org/10.1109/cloudcom.2014.170
23. Kritikos K, Kirkham T, Kryza B, Massonet P (2015) Security Enforcement for Multi-Cloud Platforms—The Case of PaaSage. Procedia Comput Sci 68:103–115. Cloud Forward 2015: 1st International Conference on Cloud Forward: From Distributed to Complete Computing
24. Kritikos K, Kirkham T, Kryza B, Massonet P (2017) Towards a security-enhanced paas platform for multi-cloud applications. Futur Gener Comp Syst 67:206–226
25. Kritikos K, Plexousakis D (2006) Semantic QoS Metric Matching. In: ECOWS. IEEE Computer Society. pp 265–274. https://doi.org/10.1109/ecows.2006.34
26. Kritikos K, Plexousakis D (2015) Multi-cloud application design through cloud service composition. In: 2015 IEEE 8th International Conference on Cloud Computing. IEEE. pp 686–693. https://doi.org/10.1109/cloud.2015.96
27. Kritikos K, Zeginis C, Seybold D, Griesinger F, Domaschka J (2017) A Cross-Layer BPaaS Adaptation Framework. In: FiCloud. https://doi.org/10.1109/ficloud.2017.12
28. Kròl D, Kitowski J (2016) Self-scalable services in service oriented software for cost-effective data farming. Futur Gener Comp Syst 54:1–15
29. Kühne T (2006) Matters of (meta-)modeling. Softw Syst Model 5(4):369–385
30. Magoutis K, Papoulas C, Papaioannou A, Karniavoura F, Akestoridis DG, Parotsidis N, Korozi M, Leonidis A, Ntoa S, Stephanidis C (2015) Design and implementation of a social networking platform for cloud deployment specialists. J Internet Serv Appl 6(1):19
31. Munteanu VI, Şandru C, Petcu D (2014) Multi-cloud resource management: cloud service interfacing. J Cloud Comput 3(1):3. https://doi.org/10.1186/2192-113X-3-3
32. Nikolov N, Rossini A, Kritikos K (2015) Integration of DSLs and Migration of Models: A Case Study in the Cloud Computing Domain. Procedia Comput Sci 68:53–66
33. (2014) Object Management Group: Object Constraint Language. http://www.omg.org/spec/OCL/2.4/
34. Opara-Martins J, Sahandi R, Tian F (2016) Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. J Cloud Comput 5(1):4. https://doi.org/10.1186/s13677-016-0054-z
35. Quinton C, Haderer N, Rouvoy R, Duchien L (2013) Towards multi-cloud configurations using feature models and ontologies. In: MultiCloud 2013: International Workshop on Multi-cloud Applications and Federated Clouds. ACM. pp 21–26. https://doi.org/10.1145/2462326.2462332
36. Quinton C, Romero D, Duchien L (2013) Cardinality-based feature models with constraints: a pragmatic approach. In: Kishi T, Jarzabek S, Gnesi S (eds). SPLC 2013: 17th International Software Product Line Conference. ACM. pp 162–166. https://doi.org/10.1145/2491627.2491638
37. Quinton C, Rouvoy R, Duchien L (2012) Leveraging Feature Models to Configure Virtual Appliances. In: CloudCP 2012: 2nd International Workshop on Cloud Computing Platforms. ACM. pp 21–26. https://doi.org/10.1145/2168697.2168699
38. Rossini A (2015) Cloud application modelling and execution language (CAMEL) and the PaaSage workflow. In: Celesti A, Leitner P (eds). ESOCC 2015: Workshops of the 4th European Conference on Service-Oriented and Cloud Computing, EU Research Projects Track. Springer. pp 437–439. https://doi.org/10.1007/978-3-319-33313-7
39. Rossini A, Kritikos K, Nikolov N, Domaschka J, Griesinger F, Seybold D, Romero D (2015) D2.1.3—CAMEL Documentation. In: PaaSage project deliverable. https://paasage.ercim.eu/about/project-deliverables/206-d2-1-3-camel-documentation
40. Silva GC, Rose LM, Calinescu R (2014) Cloud DSL: A language for supporting cloud portability by describing cloud entities. In: Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE), co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014), CEUR-WS.org., vol. 1242. pp 36–45. http://ceur-ws.org/Vol-1242/
41. Yu E, Giorgini P, Maiden N, Mylopoulos J (2011) Social Modeling for Requirements Engineering. In: The MIT Press. https://dl.acm.org/citation.cfm?id=1941925

Publisher's Note