PaaSage

Model Based Cloud Platform Upperware

Deliverable D3.1.2

Product Upperware

Version: 1.1

D3.1.1

Name, title and organisation of the scientific representative of the project's coordinator:

Mr Philippe Rohou Tel: +33 4 9715 5306 Fax: +33 4 9238 7822 E-mail: philippe.rohou@ercim.eu

Project website address: http://www.paasage.eu

Project	
Grant Agreement number	317715
Project acronym:	PaaSage
Project title:	Model Based Cloud Platform Upperware
Funding Scheme:	Integrated Project
Date of latest version of Annex I against which the assessment will be made:	3rd July 2014
Document	
Period covered:	M18-M36
Deliverable number:	D3.1.2
Deliverable title	Product Upperware
Contractual Date of Delivery:	30th September 2015 (M36)
Actual Date of Delivery:	30th October 2015
Editor (s):	Christian Perez
Author (s):	Shirley Crompton, Kamil Figiela, Geir Horn, Frank Griesinger, Dennis Hoppe, Tom Kirkham, Kyriakos Kritikos, Maciej Malawski, Nikos Parlavantzas, Christian Perez, Laurent Pouilloux, Daniel Romero, Craig Sheridan, Pedro Silva, Arnab Sinha
Reviewer (s):	Jörg Domaschka, Kyriakos Kritikos, Alessandro Rossini
Participant(s):	Same as authors.
Work package no.:	3
Work package title:	Upperware
Work package leader:	Christian Perez
Distribution:	PU
Version/Revision:	1.1
Draft/Final:	Final
Total number of pages (including cover):	108

DISCLAIMER

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

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

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

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

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

History

Date	Version	Authors	Description
2/10/2015	1.0	All	Final version
16/10/2015	1.1	All	Style improvement

Contents

List of Figures

List of Tables

Executive Summary

This document describes the architecture of the Upperware layer of PAASAGE at M36. The Upperware contains three main entities, known as the Profiler, the Reasoner, and the Adapter. This deliverable describes the implementation of each of them, as well their interaction through asynchronous messages. It also describes four meta-models internal to the Upperware to ease separation of concerns, in particular with respect to the Profiler and the Adapter.

This deliverable provides our view at M36. As long as experience will be gained using the developed software, the implementation of the Upperware layer can be updated. It is a refinement of Deliverable D3.1.1 [Bsi+13] that has presented the initial description of the Upperware.

Intended Audience

The deliverable is a public document designed for readers with some Cloud computing experience. It presumes the reader is familiar with the overall PAASAGE architecture as described in Deliverable D1.6.1 [Jef+13]. CAMEL is described in detail in Deliverable D2.1.3 [RP15] whereas the ExecutionWare is presented in Deliverable D5.1.2 [Hop+15]. This document is an updated version of Deliverable D3.1.1 [Bsi+13] that describes the first prototype version of the Upperware.

For the external reader, this deliverable provides an insight into the Upperware sub-module of PAASAGE, its architecture and its various entities. For the research and industrial partners in PAASAGE, this deliverable enables an understanding of the design of the Upperware, its capabilities and also its limitations.

1 Introduction

In order to facilitate the integration across the components managing the life-cycle of multi-cloud applications, PaaSage leverages upon CAMEL models that are progressively refined throughout the *modelling*, *deployment*, and *execution* phases of the PaaSage workflow (see Figure 1):

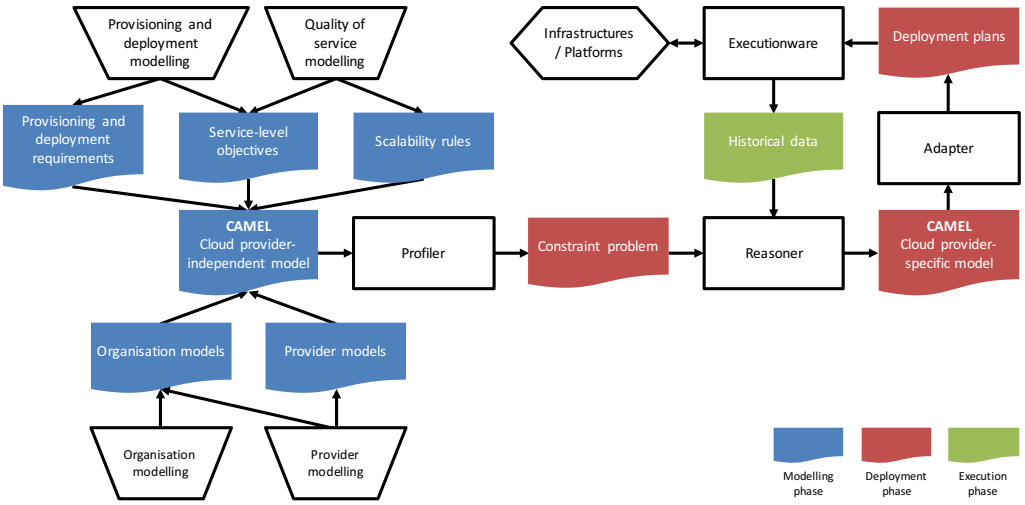

Figure 1: CAMEL models in the PaaSage workflow

- *Modelling phase*: The PaaSage users design a *cloud-provider independent model* (CPIM), which specifies the deployment of a multi-cloud application along with its requirements and objectives in a cloud provider-independent way.

- *Deployment phase*: It is handled by the Uppeware, made of three sub-components. First, the Profiler component consumes the CPIM, matches this model with the profile of cloud providers, and produces a *constraint problem*. Second, the Reasoner component solves the constraint problem (if possible) and produces a *cloud-provider specific model* (CPSM), which specifies the deployment of a multi-cloud application along with its requirements and objectives in a cloud provider-specific way. Third, the Adapter component consumes the CPSM and produces *deployment plans*, which specify platform-specific details of the deployment.

- *Execution phase*: The Executionware consumes the deployment plans and enacts the deployment of the application components on suitable cloud infrastructures. Finally, the Executionware records historical data about the

application execution, which allows the Reasoner to look at the performance of previous CPSMs when producing a new one.

Deliverable D1.6.1 [Jef+13] has provided a high level view of these elements. In particular, it gives some insights on the components of the Upperware (Profiler, Reasoner, and Adapter), and how they relate. This document present the major choices we have made for developing them, showing in particular the inputs and outputs of the various sub-components. One major change with respect to the initial architecture presented in D1.6.1 is the introduction of four meta-models private to the Upperware to ease separation of concerns, in particular with respect to the Profiler and the Adapter.

Deliverable D3.1.1 [Bsi+13] has presented an initial description of the Upperware, known as the prototype Upperware. The current document is an updated version of Deliverable D3.1.1 based on 18 more months of work. As a whole, the architecture does not have any major change.

Structure of the document

The structure of this document quite closely follows the structure of the Upperware. Section 2 sums up the architecture of the Upperware, its relationships with other PAASAGE elements, as well as the ZeroMQ based communication layer between Upperware elements. Next, Section 3 motivates and presents the four meta-models internal to the Upperware. The sub-components of the three major entities of the Upperware are then described: Section 4 for the Profiler elements, Section 5 for the Reasoner elements, and Section 6 for the Adapter elements. Section 7 concludes the deliverable.

This deliverable contains the appendices. Appendix A fully describes the Upperware meta-models. Appendix B fully represents the Saloon Ontology. Appendix C list the full data dictionary output by the Plan Generator of the Adapter.

2 Upperware Architecture Overview

2.1 Overview

As defined in Deliverable [Jef+13], the first objective of the Upperware is to compute which commands to send to the Executionware from a CAMEL configuration model (initial deployment). To this end, it can make use of the Metadata Database (MDDB) to retrieve information related for example to Cloud Providers or to historical data related to previous executions.

After the initial deployment, the Upperware will typically receive monitoring information from the Executionware and the MDDB. Its tasks will be to compute new commands to send to the Executionware to reconfigure the existing deployment in order to still conform to the user's deployment constraints.

As shown in Figure 1, at the end of the deployment description phase, that involves the deployment design, the identification of requirements and goals, all information is gathered into a CAMEL configuration model (aka CPIM). This document instance is the initial input to the Upperware. Its processing is as follows. First, the Profiler analyses it to produce an Upperware model , that contains a list of potential candidate providers that satisfy the constraints. Second, the Reasoner computes a CAMEL deployment model (aka CPSM), that is the chosen deployment solution. Third, the Adapter addresses the transformation of the output of the Reasoner into the target configuration in an efficient and consistent way, by derving a set of commands to the Executionware.

The Adapter is also responsible for performing high-level application management, which involves monitoring and adapting components deployed on multiple cloud providers, at runtime to still satisfy the user deployment and performance requirements initially posed.

The reminder of this section gives an overview of two transveral elements: the ZeroMQ messaging library as describe in the Section 2.2, and the Upperware Meta-Models as introduced in Section 2.3.

2.2 ZeroMQ

Overview

In order to distribute metric information effectively across the PaaSage architecture the project decided to use the ZeroMQ messaging library. The choice of ZeroMQ was made after an investigation over alternatives including RabbitMQ. It was decided that ZeroMQ offered the best option to support scalable implementations of PaaSage where the possibility of large amounts of messaging data is expected.

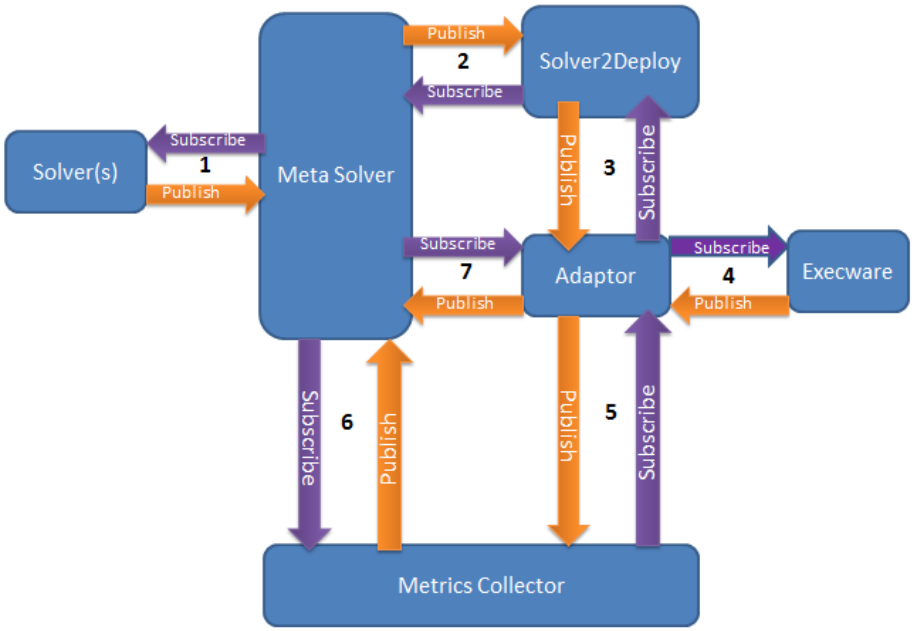

Figure 2: ZeroMQ Architecture of the Upperware

ZeroMQ supports multiple forms of message exchange and can achieve this without the use of a message broker. This enables point to point connections to be developed in the PaaSage architecture which improves performance and removes potential security risks associated with a broker (as a third party) handling messages from various sources.

Design

The use of ZeroMQ in the year 3 implementation of PaaSage is to enable the redeployment of applications. ZeroMQ is used to pass messages between Upperware and Executionware components currently associated with the deployment and reasoning of PaaSage deployments.

Using ZeroMQ metrics are sent to a number of components from the Metrics Collector. These metrics are used by the MetaSolver, Adaptor, the LA Solver, the SimWrapper to monitor the current application deployments and adapt operation in response. A typical adaptation is the creation of a new solution either directly by the Solver based on metrics or via a request directly from the MetaSolver's reading of metrics or indirectly via the MetaSolver through the Adaptors analysis of metrics.

Implementation

The project uses the publish and subscribe model for message exchange. When components in the PaaSage architecture are started they automatically look to subscribe to specific ZeroMQ publishing endpoints provided by other components. The address and port of these endpoints is contained in a configuration file that is exploited during a component's initialisation to adjust it.

ZeroMQ message exchange is visualised in Figure 2. As illustrated, the publish and subscribe message exchanges dictate the flow of data and are done in a point to point fashion.

As Figure 2 illustrates the main point to point connections between Reasoner and the Adapter internal components, and also the Executionware. This is because the key focus of the implementation in year 3 is to use the messaging framework to enable to the platform to adapt to changes. The key workflow is reflected in the order of messages (marked in numbers within Figure 2), explanations of the messaging can be seen in Table 1 below.

ID	Purpose	Format	Publisher / Port
1	Solution Notification	MQ: "solutionAvailable", Reference to Model in CDO (String)	Solvers: 5530 to 5540
2	Model to Deploy Request	MQ: "startDeployment", Reference to CPModel (String), Reference to Model in CDO(String)	Meta Solver: 5544
3	Deploy Model Request	MQ: newDeploymentCAMELModel, Ref to Updated / New CAMEL Model in CDO	Solver2Deploy: 5546
4	Scale Action Notification	MQ: entityName(String) , ID(String), typeOfAction(String)	ExecutionWare: 5548
5	Metric Configuration	MQ, "startCollection",Reference to model in CDO	Adaptor: 5550
6	Metric Notification	MQ, metricName(String), value (String), Reference to model in CDO	Metrics Collector: 5552
7	Solve Model Request	MQ: "startSolving", Reference to Model in CDO (String), Reference to CPModel (String)	Adaptor: 5550

Table 1: ZeroMQ Messages in the Upperware.

In addition to the message formats of the ZeroMQ implementation Table 1 also illustrates the implemented ports and publisher / subscriber relationships.

Model references passed are used by the components to retrieve specific models from the CDO server/MDDB and compare the contents against the received metric data from ZeroMQ.

2.3 Upperware Meta-Models

PAASAGE is a model based project. The Profiler and most Reasoner and Adapter elements are based on the usage of models. Therefore, we have defined four additional meta-models that aims at capturing elements important for Reasoner components. Section 3 presents these meta-models in details, and their usage is detailed when components of the Profiler (*cf.* Section 4) and the Reasoner (*cf.* Section 5) make use of them. Globally, the Profiler creates meta-model instances that are mainly read by Reasoner components. The only important modifications made by Reasoner components is for storing a deployment solution into them.

These meta-models are mainly defined to capture information related to the deployment problem that the Reasoner has to solve. We have aimed to minimise dependencies to CAMEL, for example by controlling the exposition of CAMEL concepts. Our goal was to separate as much as possible CAMEL evolutions from Reasoner internals.

3 Upperware Meta-Models

As introduced in Section 2.3, four meta-models have been defined to minimise model transformations and to minimise Reasoner dependencies to CAMEL. Figure 3 provides an overview of these meta-models and their relationships. The *Constraint Problem Meta-Model* (CP Meta-Model) and *Types Meta-Model* enable the definition of the Cloud provider selection problem as a constraint problem. The *PaaSage Application Meta-Model* (PaaSage App Meta-Model) and *PaaSage Type Meta-Model* establish the relationship between concepts from the Cloud and constraint problem worlds. These meta-models are presented in the remaining of this section.

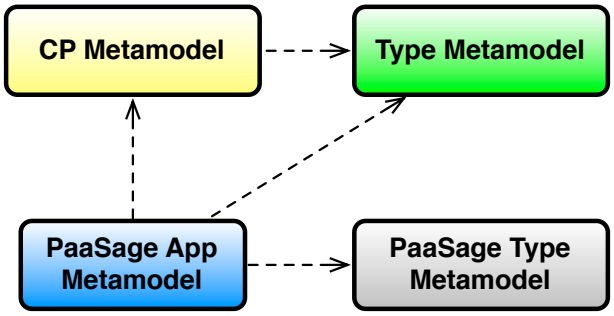

Figure 3: Meta-Models overview.

3.1 Types and Constraint Problem Meta-Model

Overview

The CP meta-model contains the different concepts needed to define a constraint problem: variables, constants, constraints, objective functions and solutions. Figure 4 presents an overview of this meta-model. An objective function is reified through a *NumericExpression* which value will be maximised or minimised (*cf. Goal* and *GoalEnumTypes*). The constraints are *ComparisonExpressions* that are defined by means of auxiliary expressions. A *Solution* provides values for different variables and metric variables that are in the constraint problem and a time stamp indicating when the problem was solved.

Expressions

The *Expressions* are mainly *NumericExpressions* and *BooleanExpressions*. *Variables*, *Metric Variables*, *Constants* and *ComposedExpressions* are numeric expressions as depicted in Figure 5. A *ComposedExpression* contains a set of nu-

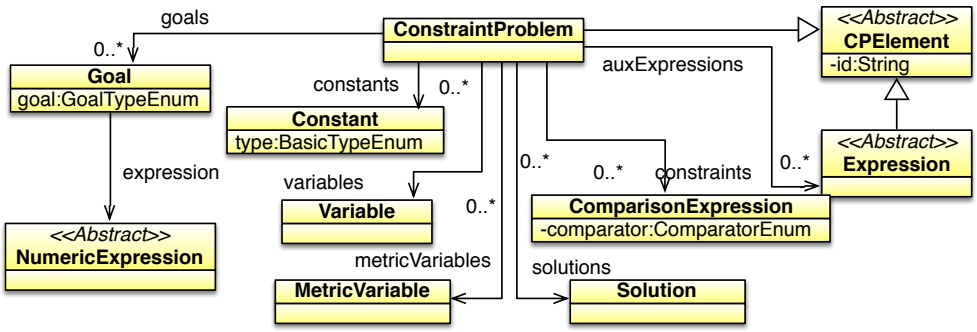

Figure 4: CP Meta-Model overview.

meric expressions related through an operation, *i.e.,*, addition, subtraction, multiplication, and division (*cf. OperatorEnum* in Figure 5).

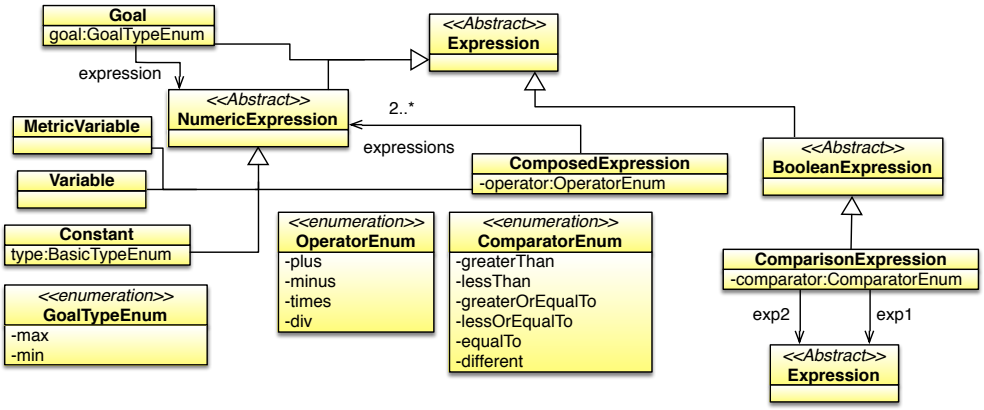

Figure 5: Expressions in the CP Meta-Model.

A boolean expression is a *ComparisonExpression* that relates two expressions by using a comparator, such as >,<, $\geq$, $\leq$, $=$ and $\neq$ (*cf. ComparatorEnum* in Figure 5).

Types, Variables, Metric Variables, Solutions and Constants

Figure 6 shows the *Types Meta-Model* concepts in green, and the concepts in yellow are the ones related to CP Meta-Model. As observed, this meta-model defines the basic values types, *i.e.,*, integer, long integer, float, double, boolean and string, which are used to characterise variables, metric variables and constants in a constraint problem.

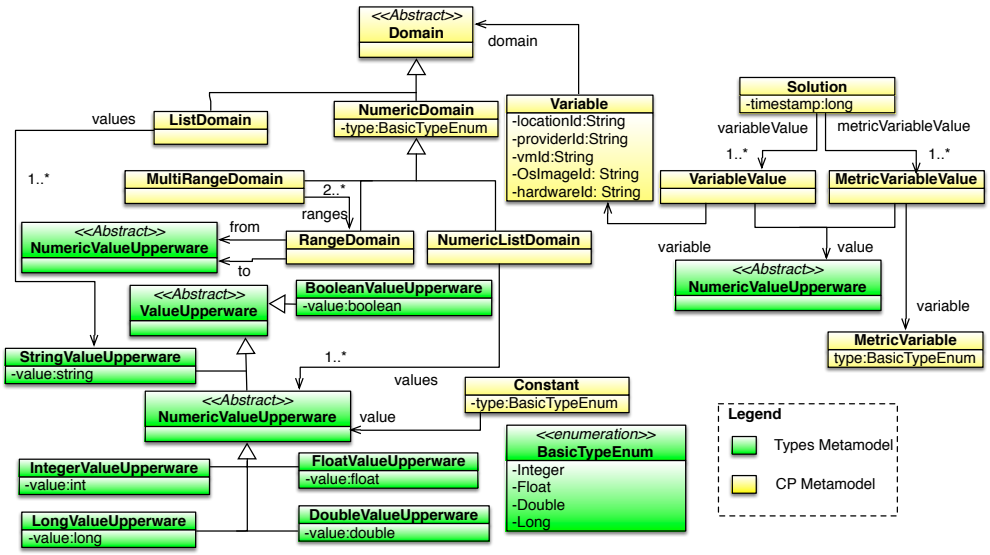

Figure 6: Type Meta-Model: types, variables, and constants in the CP Meta-Model.

A *Variable* has a *Domain* that can be numeric or a list of strings while a *MetricVariable* has a basic type (*cf. BasicTypeEnum* in Figure 6) as domain. A *Variable* also defines identifiers as strings which are related to different requirements defined by the CAMEL model of the application. A *NumericDomain* is defined by basic types. However, this kind of domain can be specialised for only considering a subset of values through *NumericListRange*, *RangeDomain* and *MultiRangeDomain*. A *MultiRangeDomain* contains two or more ranges. On the other hand, a *Constant* also has a value but its domain is defined only by basic types.

A *Solution* defines *VariableValues* and *MetricVariableValues* which specify numeric values for variables and metric variables respectively. A CP can have several solutions as Cloud offers can evolve at runtime making necessary to modify the deployment of the application.

Figure 38 in Appendix A depicts the whole *Types Meta-Model* and *CP Meta-Model* and their relationships.

3.2 PaaSage Type and Application Meta-Models

Overview

The *PaaSage Application Meta-Model* (or *PaaSage App Meta-Model*) combined with the *PaaSage Type Meta-Models* contain the required concepts to charac-

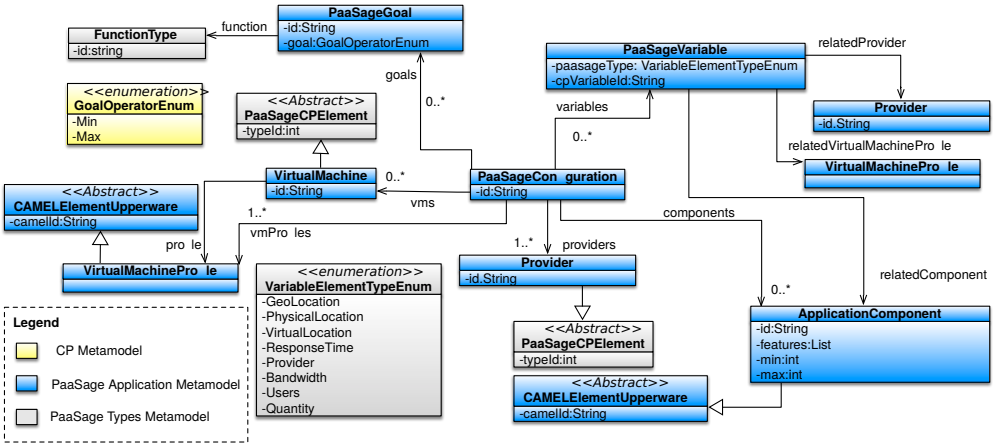

Figure 7: PaaSage Type and Application Meta-Models overview.

terise an application to be deployed by using the PaaSage platform. Figure 7 provides an overview of these models. As observed, an application is described with a *PaaSageConfiguration* containing *VirtualMachineProfiles*, and a set of *Providers*. The configuration also contains *PaaSageGoals*, *i.e.,*, minimisation and/or maximisation of relevant dimensions for users. The final objective of the *PaaSage Type and Application Meta-Models* is to allow the derivation of the CP Model used by the `Reasoner` (*cf.* Section 5) and the generation of virtual machine and component instances for the `Adapter` component (*cf.* Section 6).

Virtual Machines and Providers

The Cloud providers are reified by the *ProviderType* class in the *PaaSage Type Meta-Model* (*cf.* Figure 8). This means that, for example, Amazon EC2, ElasticHosts and Windows Azure are provider types. As these providers can be located in different regions and their location is defined according to application requirements, the *PaaSage App Meta-Model* includes a *Provider* concept with a specific position, *i.e.,*, continent, country or city.

As Cloud providers, virtual machines (VM) are represented through two concepts in the meta-model (*cf.* Figure 8): *VirtualMachineProfiles* and *VirtualMachines*. The former represents the types of VM supported by providers or defined by the users themselves. The latter represents concrete instances of *VirtualMachineProfiles* that will potentially enable the application execution.

VirtualMachineProfiles have an operating system (*OS*), memory, storage capacity, CPU (with frequency and number of cores) and location as depicted in Figure 8. These profiles are also related to a *ProviderDimension* that depends on a specific metric (*e.g.*, cost and availability) related to the provider.

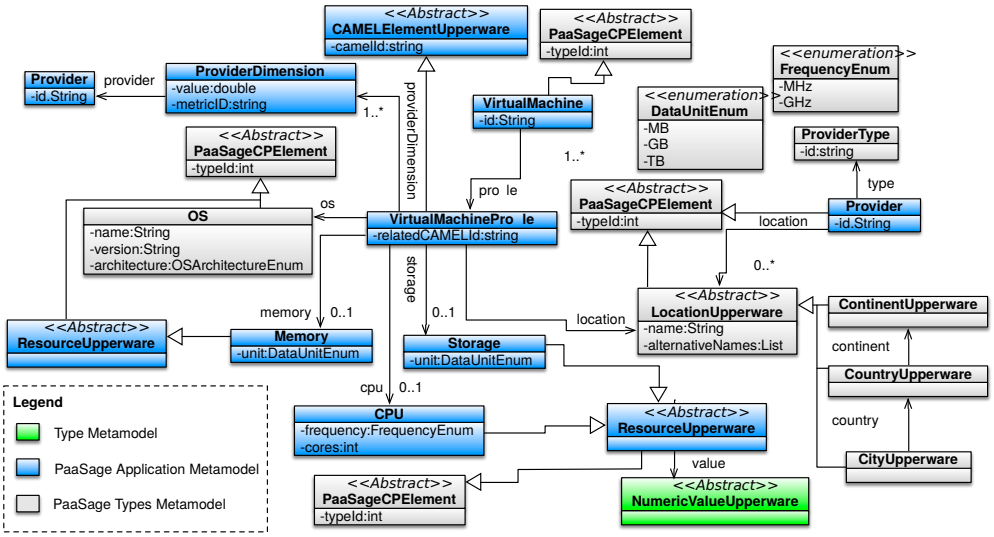

Figure 8: Virtual Machines and Providers in the PaaSage Type and App Meta-Models.

Variables and Application Components

ApplicationComponents in the *PaaSage App Meta-Model* represents the internal components in CAMEL [Ros+15]. An *ApplicationComponent* has *RequiredFeatures*, which are dependencies to other application components or virtual machines. An *ApplicationComponent* also has a list of preferred providers to be deployed and a VM that could contain it. Figure 9 shows *ApplicationComponent* and these relationships.

PaaSageVariables represents the connection between a *PaaSage App Model* and a *CP Model*. They are related to *ApplicationComponents* and they typically define relationships between VM, providers and application components.

3.3 Example

In order to illustrate the use of the presented meta-models, we use a Java application, called Simple Application, with one component containing a *Web application ARchive* (WAR) file. Let assume the requirements are as follow:

- *Required resources:* ≥ 512 MB of RAM, ≥ 1 GB of hard disk, ≥ 1.6 GHz of CPU frequency.

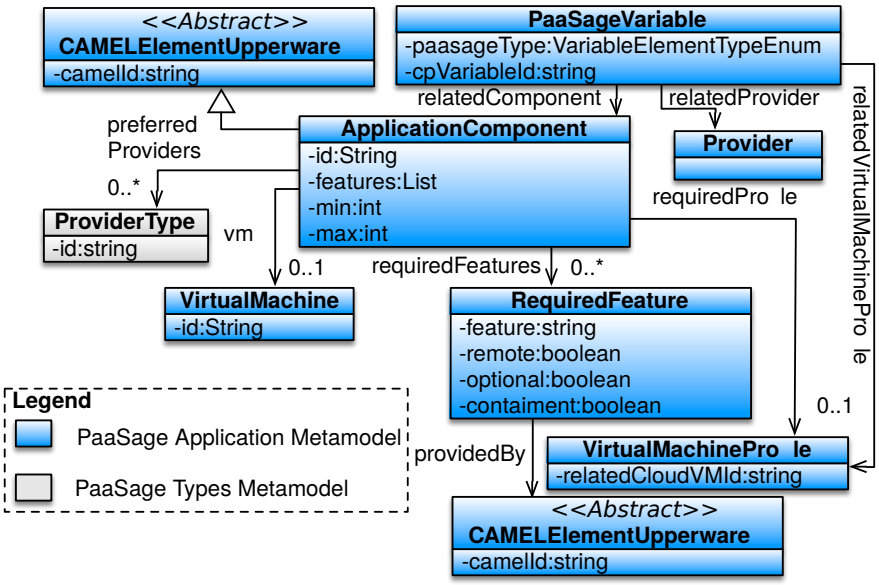

Figure 9: Application Components and Variables in the PaaSage Type and App Meta-Model.

- *Preferred providers:* Amazon EC2[1], ElasticHosts[2], Windows Azure[3].
- *Preferred operating system:* Ubuntu Server 13.X.
- *Additional services:* Apache Tomcat 7.X.
- *User goal:* Minimisation of the deployment cost.

CP Model

Equation 1 represents a simple cost function that we want to minimise for the simple application example.

$$\sum_{c \in COMP} \sum_{p \in PROV} (number\_of_{vm1,c,p}) \times Price_{vm1,p} \equiv \quad (1)$$

$$(number\_of_{vm1\_simpleApplicationWar\_amazon1}) \times Price_{vm1\_amazon1}) + (number\_of_{vm1\_simpleApplicationWar\_elastichosts1}) \times Price_{vm1\_elastichosts1}) + \ldots \quad (2)$$

[1]Amazon EC2: `http://aws.amazon.com/ec2/`
[2]ElasticHosts: `http://www.elastichosts.com/`
[3]Windows Azure: `www.windowsazure.com/`

where

$vm1 =$ Virtual machine defining the hardware and OS requirements
$PROV = \{amazon1, elastichosts1, windowsAzure1\}$
$Price_{vm1,p} =$ Cost of deploying Virtual Machine $vm1$ on the provider p
$COMP = \{simpleApplicationWar, tomcat\}$
$(\forall p \in PROV| : (\forall c \in COMP| : number\_of_{vm1\_c\_p} \in [0, 128]))$

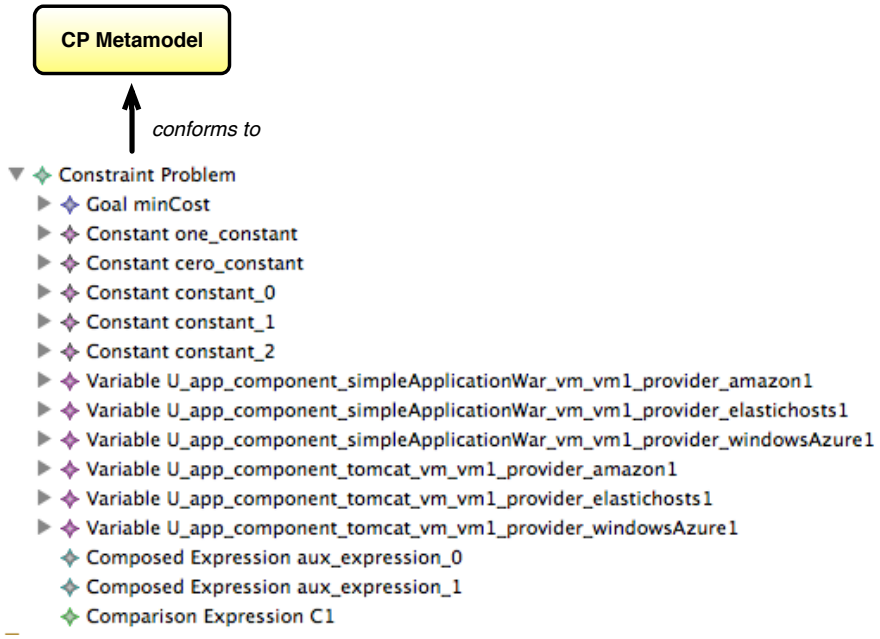

Figure 10: CP Model of the Simple Application example.

Figure 10 presents a screenshot of the related *CP Model* by using the *Eclipse Modeling Framework*[4] (EMF). In this example $vm1$ satisfies the requirements of application in terms of memory, storage, CPU and operating system. The constraint to be satisfied in order to minimise this function is presented below.

- Tomcat and application WAR have to be deployed on the same virtual machine:

$$(\forall c1, c2 | c1, c2 \in COMP \wedge c1 \neq c2 : (\forall p | p \in PROV : number\_of_{vm1\_c1\_p} = number\_of_{vm1\_c2\_p})) \quad (3)$$

[4]EMF: `http://www.eclipse.org/modeling/emf/`

PaaSage Application Model

Figure 11 shows a screenshot of the *PaaSage Model* for the Simple Application. Such a model defines the minimisation goal, the three preferred providers, the virtual machine profile that supports the required resources and the variables that define the relationship between virtual machine, providers and components.

The minimisation goal has *cost* as function type as depicted in Figure 12 *a)*. In the provider case, Figure 12 *b)* shows an instance of Amazon called *amazon1* that has Europe as location. Figure 12 *c)* indicates that Simple Application should be deployed in Europe on any of the three providers. Finally, Figure 12 *d)* relates *U_app_component_simpleApplicationWar_vm_vm1_provider_elastichosts1* variable to *simpleApplicationWar* component, *vm1* and *elastichosts1* provider.

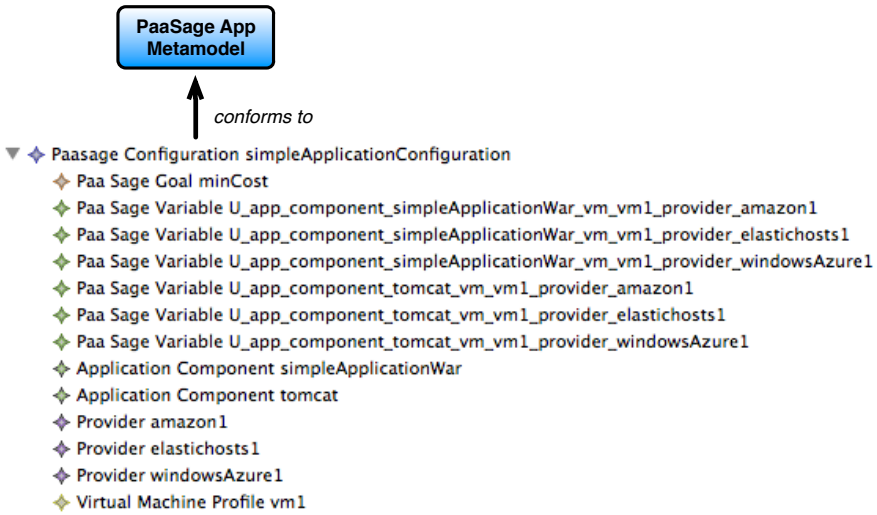

Figure 11: PaaSage Model of the Simple Application.

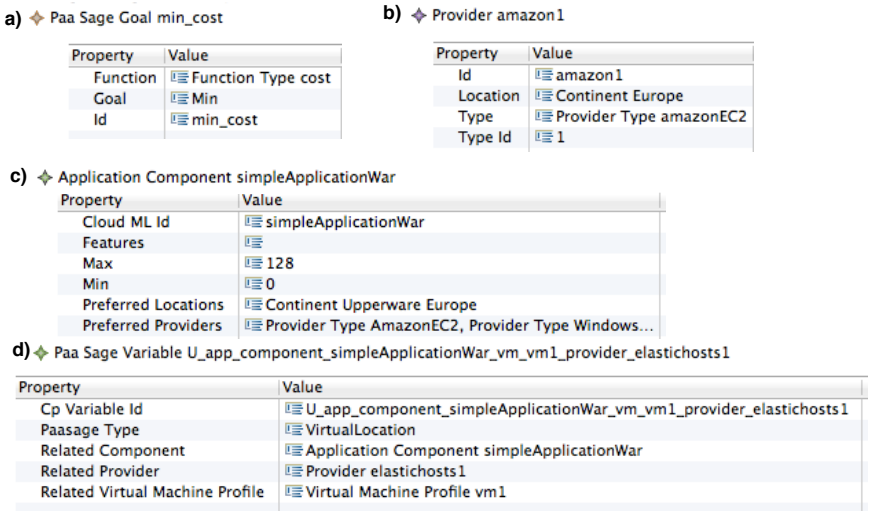

Figure 12: Properties of some elements of the Simple Application example.

4 Profiler

The Profiler represents the entry point of the Upperware, which means that it processes the different application requirements, user goals and preferences (*e.g.*, a list of desirable providers or the deployment in a specific region) in order to produce a constraint problem description used by the reasoner to find a suitable provider. Requirements refer to different computational resources (*e.g.*, memory, CPU cores and storage) and software elements (*e.g.*, operating system, database and frameworks) that the application needs to work properly. User goals (called optimisation requirements in CAMEL) are minimisation or maximisation of dimensions that have important business impact, such as cost or application response time.

Figure 13 depicts the global architecture of the Profiler. The `CP Generator Model-to-Solver` component produces a constraint problem description that is improved by the `Rule Processor` component by removing redundancies and verifying the list of Cloud providers candidates. A detailed description of these two components is given in the following sections.

4.1 CP Generator Model-to-Solver

Overview

This component receives as input a CAMEL Model, which captures the requirements in terms of computational resources and services related to the PAASAGE

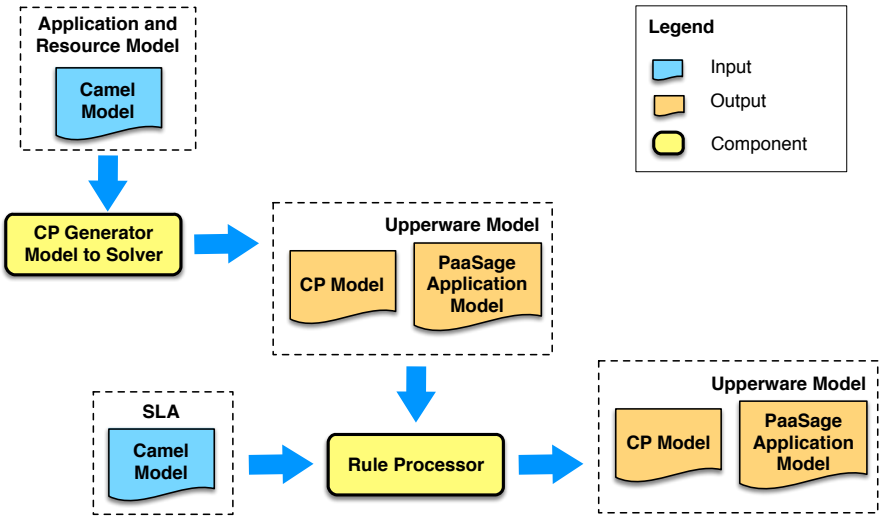

Figure 13: Profiler Architecture.

application as well as the user goals such as the minimisation of cost and response time. The outputs are a *CP Model* (*cf.* Section 3.1) representing the selection problem as a constraint problem and a *PaaSage Application Model* (*cf.* Section 3.2) that relates the variables in the *CP Model* with the Cloud concepts in the CAMEL Model. Listing 1 presents an excerpt of the CAMEL Model for the Simple Application (*cf.* Section 3.3). In particular, it shows a deployment model that defines a VM (named "vm1") satisfying the application requirements in terms of resources, location and operating system.

Implementation

Figure 14 depicts the various components that compose the `CP Generator Model-to-Solver` component. The `CAMEL Model Processor` uses the `Deployment Model Processor` and `Provider Model Processor` to deal with the input CAMEL Model. In particular, the `Deployment Model Processor` fills the *PaaSage Application Model* with information related to virtual machine profiles and their related Cloud providers as well as to the different elements that compose the application. On the other hand, the `Provider Model Processor` benefits from the core of the Saloon Framework [QRD13], which is based on the CHOCO library[5] for constraint programming, to filter the Cloud providers defined by the CAMEL Model according to the application requirements. The `CAMEL Model Processor` extracts these

[5]The CHOCO Solver: `http://www.emn.fr/z-info/choco-solver/`

Listing 1: CAMEL Model for the Simple Application example (excerpt)

```
camel model  SimpleApplication {
    ...
    deployment model  simpleApplicationDeploymentModel {
        vm  vm1 {
            requirement set  vm1RequirementSet
            provided host  vm1ProvidedHost
        }
        internal component  simpleApplicationWar {
            required host  simpleApplicationWarRequiredHost
        }
        ...
        requirement set  vm1RequirementSet {
            location:  simpleApplicationRequirementModel.
                vm1LocationRequirement
            quantitative hardware: simpleApplicationRequirementModel.
                vm1HardwareRequirement
            os: simpleApplicationRequirementModel.vm1OSRequirement
        }
        hosting  simpleApplicationWarTovm1 {
            from  simpleApplicationWar.
                simpleApplicationWarRequiredHost
            to  vm1.vm1ProvidedHost
        }
        ...
    }
    location model  simpleApplicationLocationModel {
        region  EU {
            name:  Europe
            alternative names  [ eu, europe ]
        }
    }
    requirement model  simpleApplicationRequirementModel {
        quantitative hardware  myVmHardwareRequirement {
            cpu:  1.6 ..
            ram:  512 ..
            storage:  1 ..
        }
        os  vm1OSRequirement {
            os:  ubuntu
        }
        location requirement  vm1LocationRequirement {
            locations  [ simpleApplicationLocationModel.EU ]
        }
    }
    ...
}
```

requirements from the requirement model (in the CAMEL Model) and defines them on the Saloon Ontology depicted in Figure 15. As observed, the selected concepts and some of the related values correspond to the application requirements specified in Section 3.3 and they are independent from Cloud providers. A complete version of the *Saloon ontology* is defined in Appendix B.

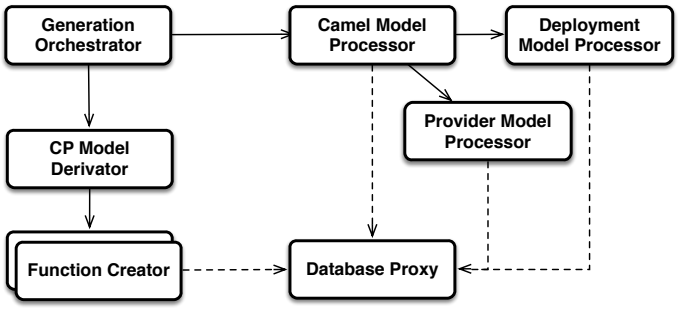

Figure 14: CP Generator - Model to Solver Architecture.

The `CP Model Derivator` component processes the *PaaSage Application Model* produced by the `CAMEL Processor` in order to generate a first version of the *CP Model* that will be later improved by the `Rule Processor` component. The list of variables, constants and constraints are derived from the different kind of virtual machines as well as the relationships between the different elements that compose the application. According to the dimensions to be optimised such as cost, response time or availability, the `CP Model Derivator` provides `Function Creator` components that create basic objective functions that will be used for the provider selection. For example, the `Cost Function Creator` generates a cost function by using the Cloud provider candidates selected by the `Provider Model Processor`. The information related to provider configurations (*i.e.*, selected virtual machine types) is retrieved through the `Database Proxy`. Finally, the `Generator Orchestrator` coordinates the whole model processesing that leads to the generation of the output models, which are stored via the `Database Proxy`. Figure 10 and Figure 11 present a screenshot of the output models in the EMF editor.

Summary

The `CP Generator Model-to-Solver` component takes a CAMEL Model as input. The outputs are a *CP Model* and a *PaaSage Application Model*. Considering the different subcomponents of the `CP Generator`

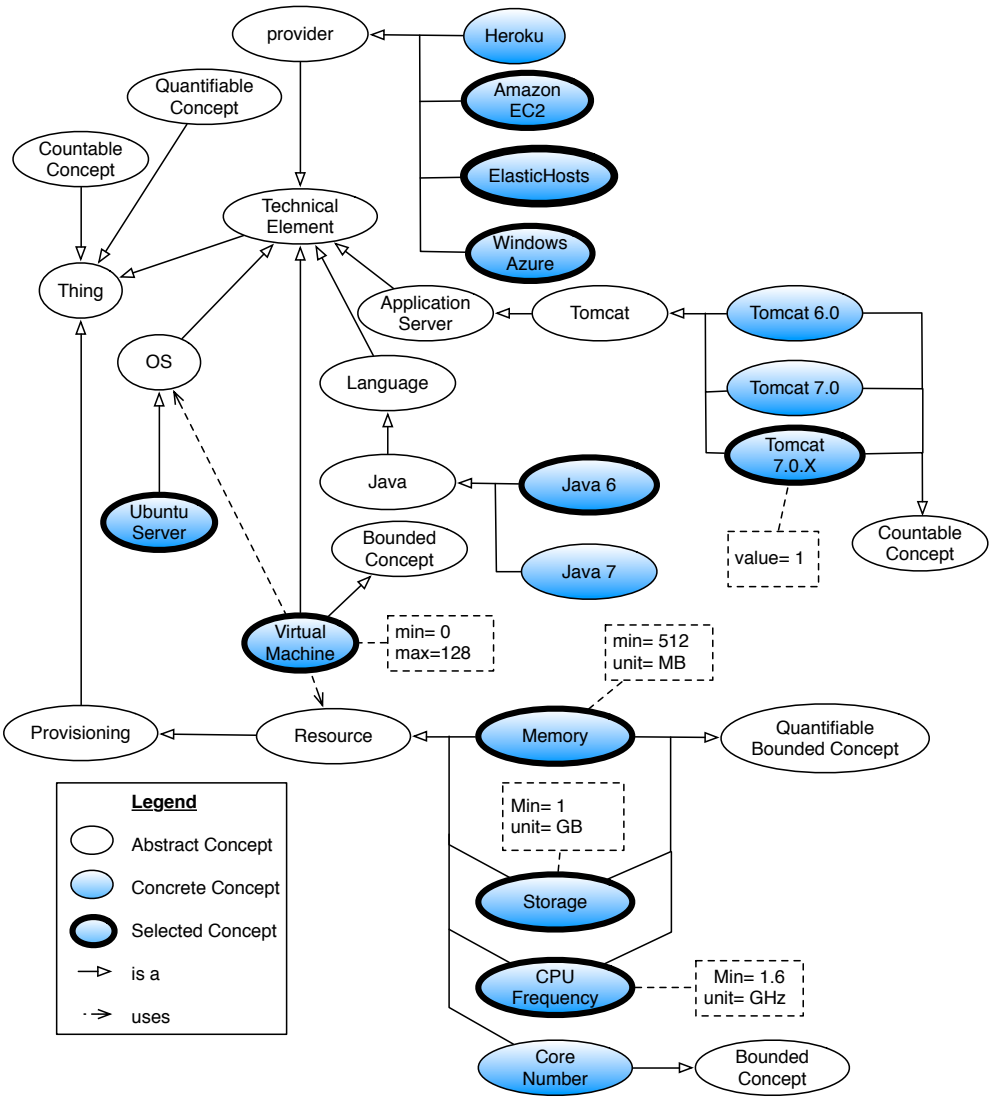

Figure 15: Saloon Ontology (excerpt) with the selected concepts for the Simple Application example.

`Model-to-Solver` (*cf.* Figure 14) the following process is executed in order to generate these outputs:

1. The `Generation Orchestrator` creates an empty *PaaSage App Model* and loads the CAMEL Model through the `CAMEL Model Processor`.
2. The `Deployment Model Processor` extracts from the Deployment model information related to virtual machines, providers and application components and fills the *PaaSage App Model* with them.
3. The `CAMEL Model Processor` defines on an ontology the application requirements by processing the *Requirement Model* from the CAMEL Model.
4. The `Provider Model Processor` filters the providers in the *PaaSage App Model* according to the requirements defined in the ontology.
5. The `CP Derivator` generates a *CP Model* from the *PaaSage App Model* and uses the `Function Creator` to generate a basic objective function.
6. The generated models are stored in CDO Server to be retrieved by the `Rule Processor` component.

The previous process is also depicted in Figure 16.

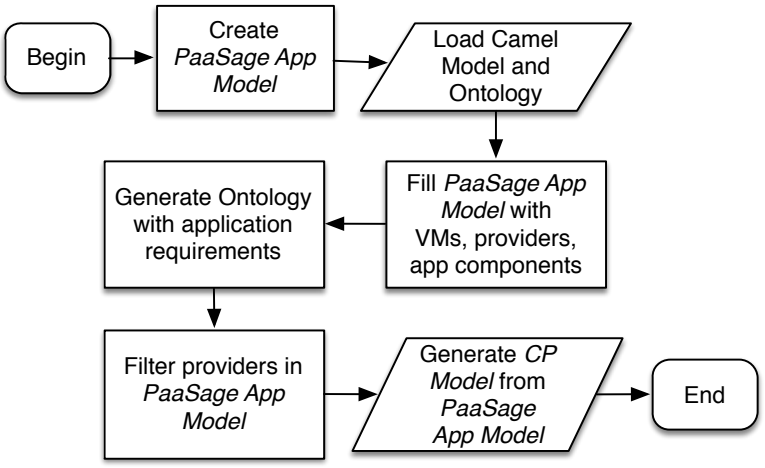

Figure 16: Process executed by the CP Generator Model-to-Solver component.

4.2 Rule Processor

Overview

The `Rule Processor` component improves, as already motivated in Section 4.1, the *CP Model* generated by the `CP Generator Model-to-Solver` component. The `Rule Processor` guarantees that the *CP Model* will eventually only include feasible deployments. This is achieved by ensuring that the list of potential deployments and Cloud providers, as provided by the generated *CP Model*, satisfies all user-defined constraints.

The `Rule Processor` component receives as input a CAMEL Model as well as the corresponding *CP Model* generated by the `CP Generator Model-to-Solver`. The CAMEL Model provides the `Rule Processor` with information about user-defined constraints with respect to valid Cloud providers, whereas the *CP Model* itself is revised while being processed by the `Rule Processor`. The final output of the `Rule Processor` is thus an updated *CP Model*, where redundancies and infeasible deployments are removed. It should be noted that deployments formed by the `Rule Processor` take constraints and details of the implementation in PAASAGE's Metadata Database (MDDB) into account.

The Profiler provides the `Rule Processor` with a list of potential Cloud providers based on requirements specified by the application designer via the IDE. The `Rule Processor` then checks these providers against implementation specific rules in the MDDB. These rules define essential rules of the overall system, and are expressed in terms of performance and data processing constraints associated with the specific instance of the PAASAGE platform.

An example: The `Rule Processor` component could contain a rule that limits data processing to private clouds; the MDDB could further have SLA specific data from associated service providers that fulfil this rule. Thus, it is added to the list of potential deployments. If during this phase the `Rule Processor` encounters a requirement that can't be fulfilled by the PAASAGE platform, e.g., no private Cloud providers are available for data processing, the `Rule Processor` returns an error message explaining this fact to the application designer.

Implementation

The `Rule Processor` receives as input the CP Description from the `CP Generator Model-to-Solver`. The description defines a list of input constraints for a planned deployment. The content of the CP Description is a *CP Model* and a PAASAGE Application Model. The latter describes characteristics of an application including service level objectives, CPU and other provider

goals. The CP Description is passed to the `Rule Processor` as an input parameter.

The `Rule Processor` retrieves the list of potential Cloud providers from the *CP Model*, and then validates individual Cloud providers defined in the CAMEL Model (Organization Model). The purpose of the validation step is to evaluate if provider types match with the ones selected by the `CP Generator Model-to-Solver` component. In the extreme case when no Cloud provider is defined in the Organization Model, the `Rule Processor` accepts the Cloud provider added by the `CP Generator Model-to-Solver` component, and passes the *CP Model* unmodified to the next component in PAASAGE's workflow. Otherwise, the `Rule Processor` will remove from the *CP Model* all Cloud providers that do not correspond with the type of providers defined in the Organization Model. This process of removing Cloud providers can, however, result in having no provider associated to deploy an application on the PAASAGE platform. Then, the `Rule Processor` returns with an appropriate error message. The user is then encouraged to adapt her requirements in order to be able to run her application on PAASAGE's platform.

The `Rule Processor` component uses the `CDODatabaseProxy` of the `CP Generator Model-to-Solver` component to access directly the MDDB database for data extraction and *CP Model* revision.

4.3 Summary

The `Rule Processor` component receives as input a CAMEL Model as well as the corresponding *CP Model* generated by the `CP Generator Model-to-Solver`, *i.e.,* a CP Description. The output is a revised CP Description including feasible deployments—wrapped in the Deployment Model. This model is linked to a wider Cloud Application Modeling & Execution Language (CAMEL) object, where resource parameters are linked and described using specific DSLs. The `Rule Processor` then passes the CAMEL data onto the *Reasoner*.

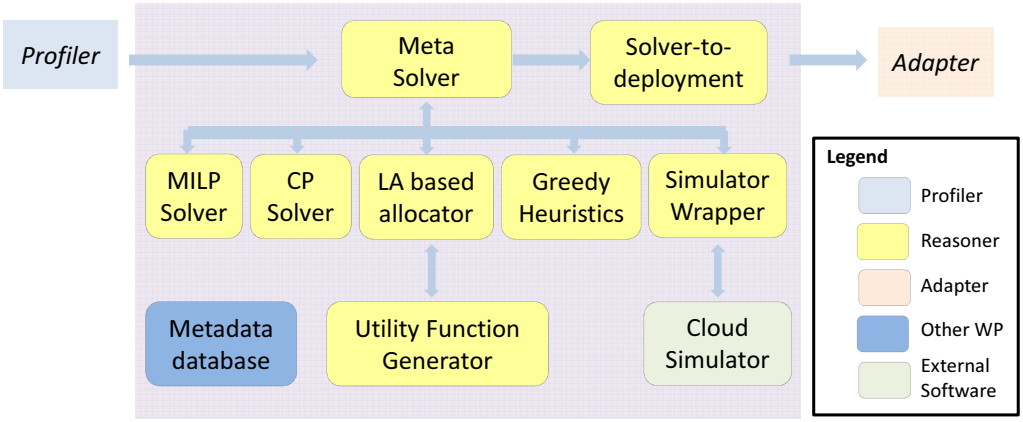

Figure 17: Reasoner: Architecture and main components.

5 Reasoner

In a nutshell, the Reasoner part of the Upperware receives a CAMEL model and a constraint problem (CP model) from the Profiler. Its goal is to compute a solution, *i.e.,* to provide a placement of an application on concrete VMs. It may also use the data and knowledge stored in the the meta-data database when computing a solution.

As there are many ways in the litterature to compute a solution, depending mainly on tradeoff about required knowledege, time-to-solution, general versus specific solutions, we decide to have a very flexible Reasoner layer such that very different type of approaches can be integrated. Hence, the Upperware will be easely adaptable and/or extendable.

The architecture of the Reasoner is displayed in Figure 17. Central to the Reasoner is the concept of Solvers. The Solvers sit at the centre of the component and conduct the main functions in the Reasoner. The Reasoner shall support various kinds of Solvers: Learning Automata based allocator, Constraint Programming based solvers, Heuristics based solvers, Simulation based solver, and Meta-Solvers. These solvers can access the MDDB to access historical data and/or metrics. For very reactive solvers, it is also possible to directly received metrics without going through the MDDB.

When a solution has been computed, the Solver-to-deployment component translates the solution in a CAMEL deployment model format (aka CPSM).

5.1 Learning Automata (LA) based Assignments

Overview

The problem to solve is defined by *variables*, each defined over a given *domain*. Then there are constraints, *i.e.,* functional relations of the variables. Each constraint is either *satisfied* when the functional expression evaluates to *true* or *unsatisfied* when it evaluates to *false*. A particular assignment of variables is *feasible* if all constraints are satisfied. At the conceptual level, *reasoning* on the deployment problem means assigning values to all parameters from their respective *domains* to form a *feasible* deployment *configuration*.

An inherent weakness of semantic reasoning is its inability to deal with probabilistic knowledge [LH13]. There have rightly been attempts on logic based stochastic reasoning like the Probabilistic Logic Network [Ben+08] or, more recently, the Non-Axiomatic Logic [Pei13], which aims to be a complete model for how humans learn and reason. To our knowledge these approaches are not yet supported by accessible reasoners making it hard to adopt the frameworks within the PaaSage project.

Fuzzy reasoning [CC05] can make decisions under uncertainty conditioned on the *a priori* system knowledge encoded into the fuzzy sets defined for the input and output variables, and the control strategy encoded in the fuzzy rules. Even though there are heuristic methodologies to support the development of these rules, they will be subject to the same issues as ordinary rules in non-stationary environments. Alternatively, the rules could be derived from data mining, *i.e.,* statistical pattern recognition and parameter identification on logged data. Hence, there must be a *model* defined *a priori* whose parameters are identified. In the case of Cloud deployment this would probably mean that the fuzzy rules must be defined by a Cloud deployment expert, and it is therefore contradictory to the vision of PaaSage as a platform to aid autonomously the application owner with the deployment task. A further issue with fuzzy reasoning is the difficulty in analysing the adaptive system analytically with respect to key aspects of automatic control systems like scalability and stability.

Given that it is very difficult to extract universally available expert knowledge on generic Cloud application deployment that can automate the deployment of any application, the only solution would be to *learn* what is the better way of deploying the particular application at hand.

Different learning approaches can broadly be categorised as either:

Training approaches that use available information to train the algorithms or a controller, and after the training phase the learned knowledge is reused on similar problems. Data mining techniques, pattern recognition, statistical parameter estimation and regression, clustering, and neural networks are all examples of *training based* approaches.

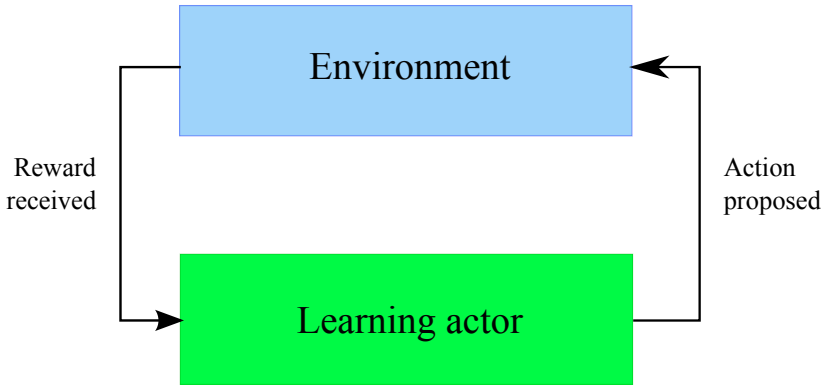

Figure 18: The fundamental learning loop: The learning actor proposes an action to the environment. In this case, the action is a particular deployment configuration. The environment then provides feedback on the quality of this configuration in terms of a reward to the learning agent.

Reinforcement learning algorithms that learn as new information arrives, and the learned knowledge is immediately available, albeit it may take some time (iterations) for the algorithm to gain confidence in the selection of its strategy. The idea is that the learning actor will select the action at any moment that it perceives to give the highest future reward. Examples of approaches belonging to this class of learning are Markov Decision Problems (MDPs), Learning Automata (LA), Parameter identification, control charts, and statistical hypothesis testing.

Both classes of learning are applicable in stationary environments; however only reinforcement learning algorithms can be used in non-stationary environments because they may need to unlearn previous knowledge if the operational constraints change.

The basic learning loop is illustrated in Figure 18. The learning actor selects the appropriate "action", which in our context is a set of values for all variables in the deployment configuration with each variable value taken within the domain allowed for the concerned variable. Then the environment provides a "reward" for this choice of deployment configuration. The reward can be the strength or goodness scaled to the unit interval $[0, 1]$ from bad to good; it can be binary taken from the set $\{0, 1\}$ indicating that the action was respectively bad or good; or it can be taken from a finite set of options like {very bad, bad, almost good, good}.

The environment can be the real world, and so the action represents an actual deployment and the reward can, for instance, be the fraction of the execution cost budget that was left unused by this particular deployment, provided that minimal cost is the main goal for the user. However, learning is an iterative process and

many deployments may be necessary before the learning algorithm may confidently conclude on the best one. The need for actual deployments can be reduced if one is able to "simulate" the effect of a particular employment; either by using historical data or using a simulated model of the available infrastructures and extra-functional aspects like cost and performance. Simulated deployment is further discussed in Section 5.5.

Finally, learning can also be made against a *utility function* representing the combined set of goals and preferences specified by the application owner. In this case, any proposed deployment configuration that increases the utility for the user will be rewarded. For instance, if the user wants to execute the application at minimal cost, then the utility function can simply be the negative cost (since an increase in the utility then corresponds to less cost). Utility functions are further discussed in Section 5.7.

The solver must be able to use constructively the stochastic feedback from the environment to converge on the better deployment configuration over time. The learning environment must provide some kind of ranking of the possible solutions, and for the sake of exposition we can assume that this is the utility function[6]. From the second requirement it is clear that the maximal utility should be found respecting the constraints of the application, and the domains of the variables. The problem is therefore akin to a *mathematical non-linear program* whose canonical form is [DY08]:

$$\text{maximize } U(\mathbf{x}) \tag{4}$$

subject to

$$\mathbf{g}(\mathbf{x}) \leq \mathbf{0} \tag{5}$$
$$\mathbf{h}(\mathbf{x}) = \mathbf{0} \tag{6}$$
$$x_i \in \mathbb{X}_i \tag{7}$$

A complicating factor is that many of the parameters in the configuration $\mathbf{x}$ are discrete: as an example, the parameter for the Cloud provider can only take its values from the finite set of possible providers. If the constraints and utility function are all linear, the problem belongs to the class of *mixed integer optimisation* problems [Lau08], otherwise it is a *combinatorial optimisation* problem [BJ08]. The size of the solution space, *i.e.,* the number of possible configurations, will generally grow like the product of the sizes of the domains for

[6]Please note that this choice is made without prejudice to any of the other ways outlined for obtaining the environment's feedback to the learning actor. From this point on, a utility function value can therefore also be understood as outcome of an actual deployment or the output of a simulated system.

each discrete parameter, and finding the optimum will necessitate testing each and every possible configuration. This is obviously feasible only for small deployments, so in general the optimisation of Equation (4) must be understood as the best possible configuration tested within the search time available. The found configuration will therefore be a *feasible* configuration satisfying all the constraints, which can be safely deployed, although a better configuration might still be possible.

There is one important aspect of the deployment problem: the constraints as well as the utility function may be stochastic. For instance one constraint can specify that the average response time experienced by the user of a web server should not exceed 3 seconds. This is easily achieved with a configuration of a few web servers when there are few users, but might require a different configuration with more web servers when there are many users. Thus, for a given deployment $\mathbf{x}$, the constraints (5) and (6) will only be satisfied with a certain probability. The same is the case for the utility function. Assuming, as an example, that the utility measures cost, then a certain provider can have a discount at a particular time, or the data pattern of the application requires less communication and thus incur less communication cost. Evaluating the "utility function", which in this case could be the real deployment, twice with the same deployment configuration $\mathbf{x}$ could give two different utility values. We therefore have to consider the non-linear *stochastic* program where we would like to find the configuration that will give the best utility *on average* with expected satisfaction of the constraints, *i.e.,* we have to consider the following program:

$$\text{maximise } E\left\{U(\mathbf{x})\right\} \tag{8}$$

subject to

$$E\left\{\mathbf{g}(\mathbf{x})\right\} \leq \mathbf{0} \tag{9}$$

$$E\left\{\mathbf{h}(\mathbf{x})\right\} = \mathbf{0} \tag{10}$$

$$x_i \in \mathbb{X}_i \tag{11}$$

If the distributions for the parameters x_i were known, the problem could be approached with *stochastic programming* [PR09]. It should be possible to estimate unknown parameters of hypothesised distribution functions from available historical data, or even use empirical density functions or fitted functions as representations for the generally unknown probability density functions. Again, this would be possible only for stationary environments where the involved distributions would be constant over time. Otherwise, one would have the problem of estimating the distributions over a window of only the most recent observations of the involved system parameters.

An alternative approach requiring only known bounds for the system parameters is *robust optimisation* [ALA09], and it is attractive that efficient methods exist for robust integer programming [DM03]. However, as only the bounds are known for the parameters, robust optimisation can mainly give a worst case analysis with bounds on the robustness of the found solution.

An issue with both the stochastic programming and the robust optimisation is that the optimisation program has to be solved again from scratch when the results of a new deployment is observed since either the distributions involved or the bounds may have changed. Fortunately, there are many heuristics that can be used for stochastic search [Jam03]. For the parameters with discrete domains, *i.e.,* the combinatorial optimisation part, we will adopt a *reinforcement learning* [RA98] approach based on *Learning Automata* [KM89]. Albeit other reinforcement learning techniques can be used, learning automata theory is based on the theory of Markov chains and therefore admits rigorous mathematical analysis of key aspects like scalability and convergence.

Poznyak and Najim [AK97] developed a theory for stochastic optimisation using learning automata based on Baba's multi-teacher approach [Nor83] where the utility function (4) and the constraints (5) and (6) are all considered to be independent stochastic teachers for the learning automata. Given that this approach is feasible for the learning of a single parameter, we will use one learning automata for each discrete parameter. Thus our proposed approach corresponds to an *automata game* [MP04]. The algorithm of the proposed stochastic reasoner is shown in Algorithm 1.

Two lines of Algorithm 1 requires further attention: Line 15 states that the continuous optimisation problem should be "solved". Non-linear optimisation problems are often themselves solved by iterative algorithms [DY08], where each iteration requires sampling the objective function (4). This sampling can be costly as described above, and we need to investigate if this step of the algorithm should be finding a complete solution, or if it can be understood as "performing the next iteration of the iterative solver for the non-linear program".

Furthermore, the probability updating function of line 18 should be defined. Half a century of research on learning automata has produced a plethora of algorithms to choose from. Given that our approach is a game of many automata, we need to ensure that each automaton converges in the sense that its probabilities converge to a pure deployment strategy with only one probability equal to unity and all the others equal to zero, *i.e.,* $\lim_{k\to\infty} \mathbf{p}_i = [0, \ldots, 1, \ldots, 0]^T$. The stable behaviour of various algorithms under non-stationary environments also needs further research.

Recall that a change in a single discrete variable leads to a completely new configuration. The **foreach** loop on line 12 will therefore make a major change in the configuration. This can be seen as positive from the perspective of explor-

Algorithm 1: Stochastic reasoning.

1 Identify the variables x_i of the deployment problem
2 Identify their respective domains $\mathbb{X}_i$ from the given constraints and rules, *i.e.,* $x_i \in \mathbb{X}_i$
3 Partition the parameter set in two parts: $\mathbb{X}_{\text{Discrete}}$ for the parameters with discrete domains, *i.e.,* for those x_i whose domain $\mathbb{X}_i$ is not an interval of $\mathbb{R}$, and $\mathbb{X}_{\text{Continuous}}$ for those x_i whose $\mathbb{X}_i \subseteq \mathbb{R}$
4 **foreach** $x_i \in \mathbb{X}_{Discrete}$ **do**
5 Form probability vectors over the possible values in the domain: $\mathbf{p}_i(0) = [p_{(i,1)}(0), \ldots, p_{(i,|\mathbb{X}_i|)}(0)]^T$
6 **if** a priori *knowledge* **then**
7 Initialise the probabilities in $\mathbf{p}_i(0)$ accordingly
8 **else**
9 Initialise the probabilities equally $p_{(i,j)}(0) = 1/|\mathbb{X}_i|$
10 Initialise the step counter $k = 0$
11 **repeat**
12 **foreach** $x_i \in \mathbb{X}_{Discrete}$ **do**
13 Select a random index $I_i \sim \mathbf{p}_i(k)$
14 Assign the variable value $x_i = \mathbb{X}_i[I_i]$
15 Solve the optimisation problem for the continuous parameters in $\mathbb{X}_{\text{Continuous}}$ with the assigned values of the variables in $\mathbb{X}_{\text{Discrete}}$
16 Obtain the environment's reward $r(k)$ for the complete set of parameters $\mathbb{X}_{\text{Discrete}} \cup \mathbb{X}_{\text{Continuous}}$
17 **foreach** $x_i \in \mathbb{X}_{Discrete}$ **do**
18 Update the probabilities $\mathbf{p}_i(k+1) = \mathbf{T}\left(\mathbf{p}_i(k), r(k)\right)$
19 $k = k + 1$
20 **until** *converged*

ing the configuration space quickly, by sampling many distant configurations. However, it may have a negative impact on the convergence of the learning algorithms [GB10]. Making the selection in line 13 and line 14 for only one variable and then subsequently update the probabilities in line 18 only for this single variable may be better, but it will come at the cost of more frequent environment feedback, which can be costly to obtain especially in the case where this means making a full deployment. Arguably, as time goes and most of the automata for the different parameters converge, the **foreach** loops on line 12 and 17 will degenerate to updating only one, or a few, variables (automata) with respect to the current configuration. The trade-off between exploration speed and the cost of

evaluating the environment feedback consequently needs careful attention when developing the final stochastic search algorithm.

It should be noted that the learning automata based approach is only used for the discrete variables and *assigns* a value to each of them in a stochastic environment from their respective domains. It must therefore be coupled with a constraint solver if there are continuous variables, where the constraint solver will find values for the continuous variables conditioned on the discrete values fixed by the stochastic learning.

Implementation

The *Learning Automata* (LA) based solver is implemented in terms of the University of Oslo's open source LA framework written in C++, and contributes to the further expansion of this framework. The framework is implemented using the *actor model* [CPR73] for concurrent and parallel operation of independent actors that asynchronously exchange messages. Additionally, actors may create other actors, and designate the actions to take for the next arriving message. There is no synchronisation between an actor sending a message and the actor receiving it, and the message is self-contained with addresses of both the sender and receiver. This means that there is no fixed interaction topology among the actors. Furthermore, each actor can only act on its own, private memory. Consequently, the actor model supports inherent concurrency of computation.

The agent model is implemented in the framework by the *Theron* library[7], released as open source under the MIT licence. The Theron library is robust, efficient, and complete compared with other alternatives: *libcppa*[8] is still not in official release and the current version maps each actor to an individual execution thread, which limits the number of concurrent actors that can be created under most operating systems. The *actor-cpp*[9] implementation seems to be a minimal fragment not actively maintained. *libprocess*[10] adopts the view that each actor is a *process*. It is poorly documented with the best information source being a presentation[11] by its author even though libprocess is under active maintenance and the library is packaged with many of the Linux distributions. The lack of documentation makes it hard to evaluate libprocess.

The Theron library essentially maintains a pool of threads and schedules the message handlers of the actors with pending messages onto these threads. The Theron scheduler supports two yield modes: condition and spin. In the former

[7] `http://www.theron-library.com/`
[8] `http://libcppa.blogspot.no/`
[9] `http://code.google.com/p/actor-cpp/`
[10] `https://github.com/3rdparty/libprocess`
[11] `https://www.dropbox.com/s/50buds6t0vizr4w/libprocess.pdf`

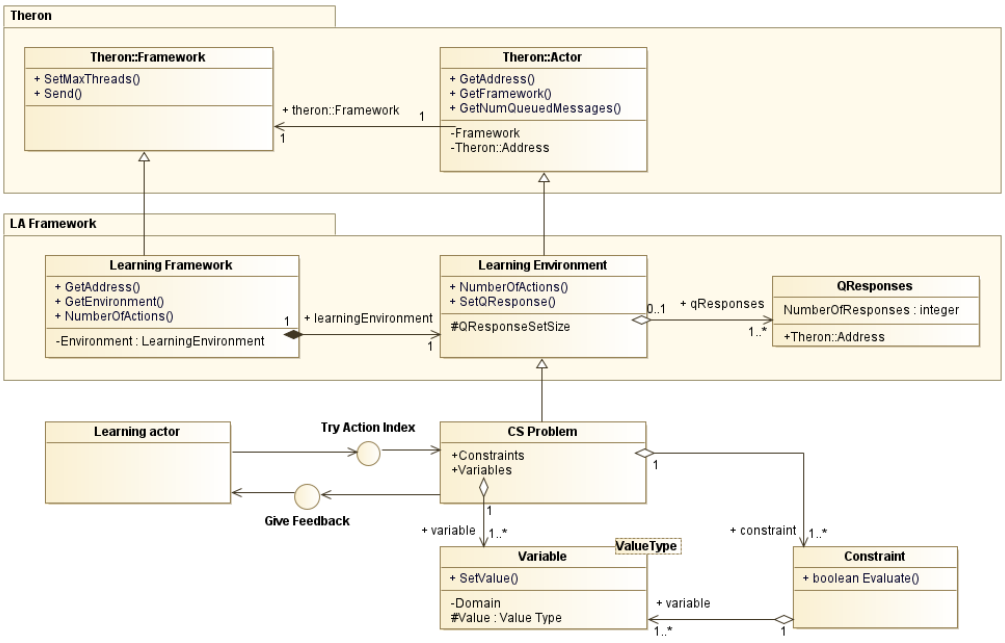

Figure 19: The learning environment controlling the problem variables and constraints is an actor that interacts with a learning actors through messages.

mode, a thread is halted if no actors uses it. This saves system resources and allows the CPU to be used for other applications. However, when many actors become active again, then the thread must be restarted by the operating system, which will introduce latency in the execution of the actors' thread handlers. The spin strategy keeps all threads active, even if they are empty. Potentially this wastes CPU cycles, but the thread is running when an actor receives a message and needs the thread to execute the message handler. Since it is anticipated that the LA based solver will run in the same machine as the other Upperware components, the "condition" type scheduler is used.

Learning actor A learning automata is basically a *Markov chain*, i.e. a set of connected states with probabilistic transitions among the states. This structure is called a Fixed Structure Stochastic Automata (FSSA), and with each state there is associated an "action" taken by the automata in that state. The feedback can either be enforcing ("reward") or discouraging ("penalty"), and the automata selects randomly a transition out of the state from the set of reward or penalty transitions respectively. Thus an FSSA is completely characterised as a graph by its set of states and the associated actions for each state, and two probabilistic adjacency matrices: one for the reward transitions and one for the penalty

transitions. The LA framework uses the *Armadillo*[12] linear algebra library to represent matrices. It was selected over the *Eigen*[13] template library for linear algebra because Armadillo had an easier interface, in particular for manipulating sub-matrices and Armadillo is also used for implementing the *mlpack*[14] machine learning library.

Only a few years after the first paper on FSSA [Mik61], Varshavskii and Vorontsova introduced the family of Variable Structure Stochastic Automata (VSSA) [VI63]. Basically they studied Markov chains where the transition probabilities were increased if the state transitions resulted in rewards, or otherwise decreased. This model was developed for binary feedback from the environment, and it was extended by McMurtry and Fu to a continuous feedback model in [GK66]. More importantly, McMurtry and Fu also introduced the concept of an *action probability vector*, i.e. the stationary action probabilities were directly updated instead of via the changing transition probabilities of the underlying Markov chain.

Each discrete variable of the CS model is represented by a learning actor, whose set of actions is the values of the domain of the variable. The learning actor selects one of the domain elements based on the probability vector (or its state if the actor implements an FSSA). The selected "action" is proposed to the learning environment, and based on the feedback from the learning environment the probability vector (or state) is updated according to the learning algorithm used by the actor.

Learning framework This class essentially sets up the Theron execution framework. All learning actors will run in this environment. The framework instantiates a learning environment and encapsulates it. This ensures that it is not possible to interact with the environment except through messages, thus enforcing the actor model.

Learning environment This class defines the learning environment. It accepts messages containing "actions" from the learning actors and produces "rewards" for the chosen actions. In the context of the LA solver, the learning environment will first evaluate all constraints, and it will then evaluate the configuration if all the constraints were satisfied by the current configuration of variable assignments proposed as "actions" by the learning agents. If the configuration is *feasible*, then it is ranked against other feasible configurations by one of the evaluation methods: actual deployment, simulated deployment or through an evaluation of the utility function.

[12] `http://arma.sourceforge.net/`
[13] `http://eigen.tuxfamily.org`
[14] `http://mlpack.org/`

Based on how this configuration ranks, an individual "reward" is calculated for each of the participating learning actors and returned to the actor to update its views on the better choices for a particular variable. Chandrasekaran and Shen [BD68] were the first to introduce the term *P-model* feedback for a *probability* model where the stochastic feedback was binary and a penalty response was given with a certain probability; and the term *S-model* feedback where the *strength* of the feedback was measured over the unit interval. Viswanathan and Narendra introduced the *Q-model* as the third feedback model where the response from the environment can take its value only from a finite set of values [RK71].

Binding the model with the solver The constraints are mathematical functions of the variables. A mathematical operator is either unary or binary and acts respectively upon one or two variables or results of other operators. Consider for example $(a + b)^2$. Here the binary "plus" operator acts on the two variables a and b and the unary operator "square" acts on the result of the plus operator. In other words the expression forms a tree of sub-expressions with the involved variables as the leaf nodes.

The profiler model essentially holds the constraints in this way, and any *interpreted* language would need to traverse this expression tree in order to evaluate the constraint value. However, a compiler would generate a compact set of CPU instructions from this tree. Given that the constraints will be evaluated over and over again for new choices of the variables, it would be a huge performance gain if the constraints could be compiled. The implication of this is that the solver will have to be linked with the object code generated for a particular problem and does not exist as a component independent of this problem.

From an Upperware meta-model instance, a `LA-Dumper` component will therefore generate a C++ source file containing the variables, their domains, and the constraint expressions. This file will be compiled as a part of starting the solver. Then the resulting object code file will be linked with the solver code in one of three possible ways.[15]

Static linking will construct one executable file by combining the problem specific object file with the object files of the solver. The problem description is then just one of the source files of the solver producing a single, standalone application that can the be executed.

Static binding is binding the compiled solver code to a dynamic library created from the compiled problem description. This dynamic library will then

[15] `http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html`

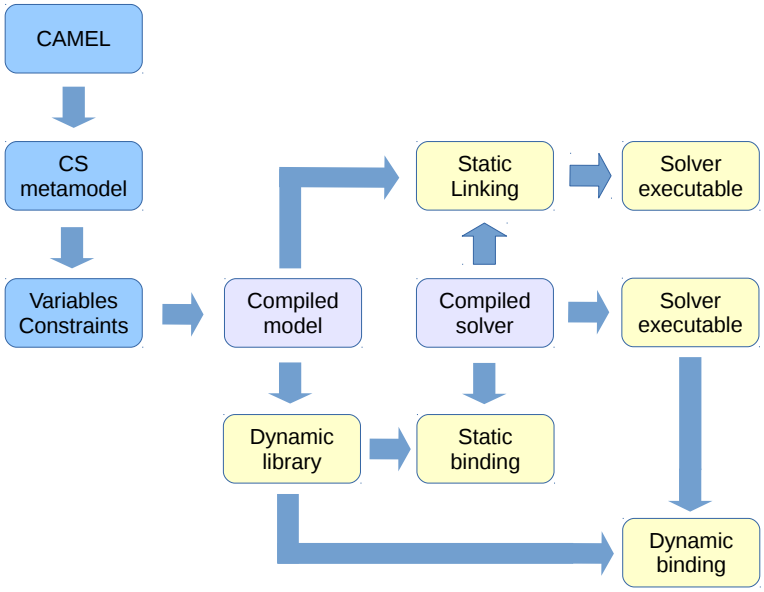

Figure 20: The various options of binding the solver code with the compiled variables and constraints of the problem at hand.

be loaded by the operating system when the solver starts. The variables and constraints can be used by the solver as if they had been statically linked with the solver. This approach would allow several versions of the solver to share the problem description, as the library will be loaded only once even if there are several solvers using it. If the problem changes, only the dynamic library needs to be recreated, and the new version is automatically loaded by the solver provided that the new library has the same name as the one that was statically bound to the solver.

Dynamic binding is similar to the static binding except that the solver can be compiled and linked with no knowledge about the problem library. The name of the library can be passed on the command line when starting the solver, which will then load the correct library into memory. There are however significant limitations on how the components in the dynamically bound library can be accessed from the solver code.

The different options are illustrated in Figure 20. Which option to choose depends on the deployment of PaaSage. The machine running the solver must

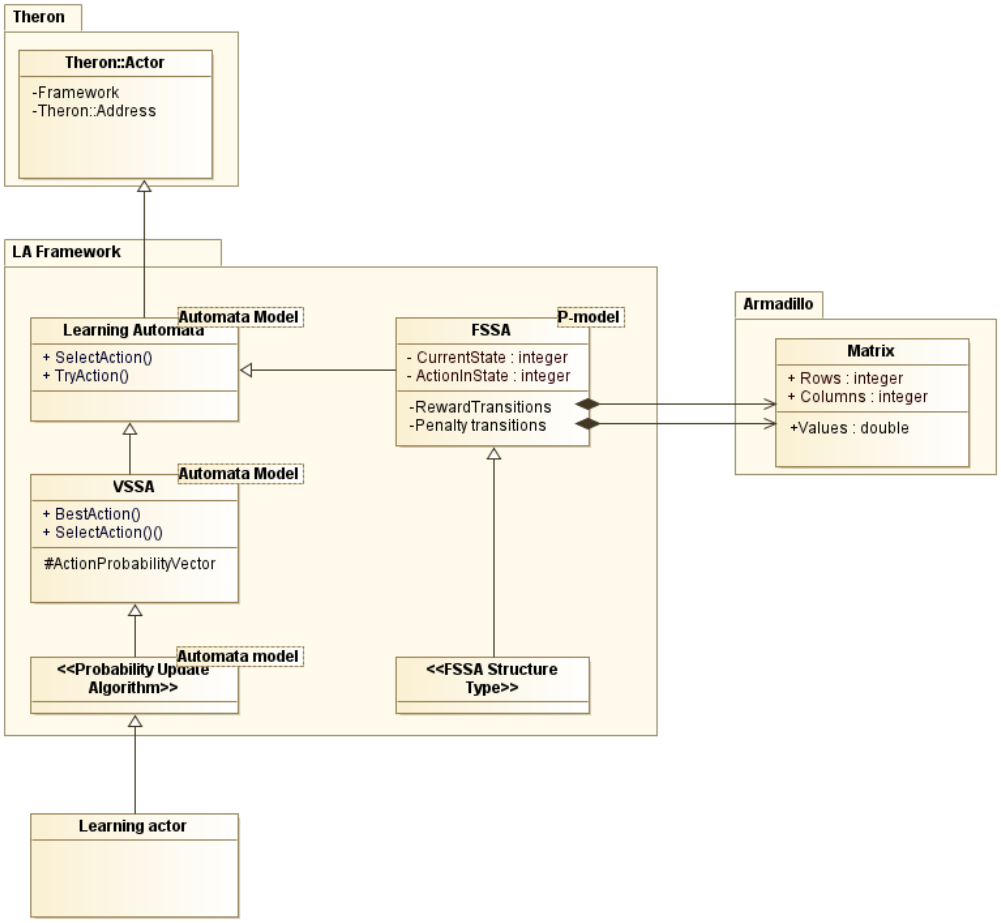

Figure 21: The hierarchy of a learning actor implementing Variable Structure Stochastic Learning. Alternatively the Learning Actor could have inherited the class implementing a Fixed Structure Stochastic Learning type.

have a compiler installed in all cases, and the object code of the solver can be compiled at installation time since it is independent of the problem. The time it takes to compile the problem description and linking or producing the dynamic library is probably similar for all three alternatives, and anyway ignorable compared with the time the reasoning will take.

Owing to the limitations of the dynamic binding, this is the least desired alternative. It is also not clear if it makes sense to have several versions of the solver working on the same problem in parallel. This could be the case if one would like to test different learning algorithms, since the learning actor is bound to the algorithm as illustrated in Figure 21. However, in order to benefit from the dynamic library the solvers must run in the same computer, *i.e.,* all solvers

will compete for the same resources. It could be better to send the source file of the problem to different computers, and then compile and statically link the problem with the solver in each machine. Thus, as static linking allows the most transparent access to the problem's constraints, this will be used initially in PaaSage with the opportunity to shift to the alternative binding models at a later stage if needed.

5.2 MILP Solver

Overview

The solver developed by AGH uses Mixed Integer Linear Programming (MILP) approach in order to optimise application deployment. The solver has been introduced into PAASAGE platform for supporting workflow applications within the extended eScience use case [Mal+13]. The `MILP solver` is to be used as a generic solver within the `Reasoner` component.

The main premise of the `MILP Solver` is to use an existing mathematical modelling framework and ready to use external solvers. To this end we decided to use CMPL [Ste14] mathematical modelling language and optimisation system. CMPL supports many commercial and open source general purpose solvers for linear and mixed integer programming problems that can be described using a high-level mathematical notation. The advantage of CMPL is that it is available as an open source project, so it can be integrated into an open platform as PAASAGE.

Design

To enable integration of the solver with the PAASAGE environment, the `MILP solver` encapsulates several components shown in Figure 22:

Solver interface This component is responsible for communication with other PAASAGE components (meta-solver and MDDB). It translates data between PaaSage CP model and internal problem description and binds the different technologies used.

CP problem transformer The function of this component is to transform CP problem to make it suitable for CMPL. For instance, it replaces variable names used in PaaSage with the generated ones to ensure that they are compilant with CMPL limitations.

CMPL problem generator The goal of this component is to generate an instance of MILP problem that is subject to optimisation by CMPL. It converts CMPL file from internal CP representation.

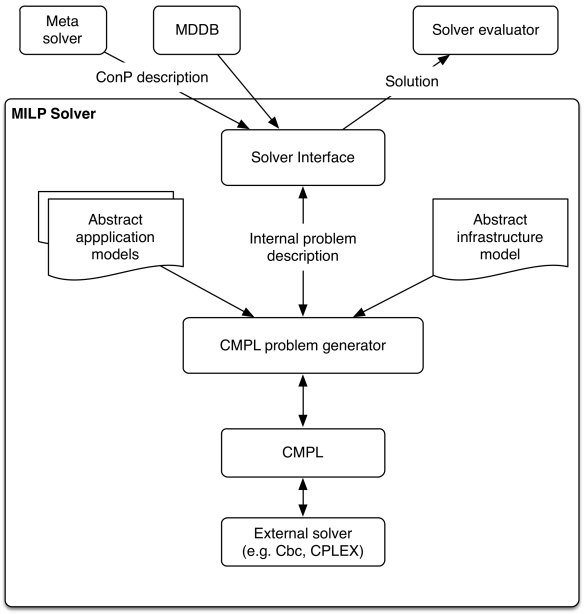

Figure 22: Architecture of MILP solver.

CMPL and external solver In order to solve mixed integer linear problem, PAASAGE solver will use open-source CMPL modelling language backed by one of the available solvers (both open-source and commercial, *e.g.* Cbc, CPLEX).

Implementation

The current version of the MILP solver implements all features described above and final integration with Meta-Solver is being implemented. The MILP solver is designed to be called as a commandline tool or from a ZeroMQ bus. First, it fetches CAMEL model and CP problem description from MDDB, then it translates problem to internal representation that is transformed and later translated to CMPL. Next, CMPL and the solver are run. As soon as they finish, the MILP solver converts back the solution, stores it into MDDB. If invoked through ZeroMQ, it sends message to Meta-Solver via ZeroMQ.

5.3 CP Solver

Overview

While the Solver developed by FORTH relied on Constraint Logic Programming techniques to solve the deployment plan problem, it was decided to update it

with a different technique, namely, constraint programming (CP) based on the following rationale:

- Exploitation of well-known CP techniques which have been proven to work well under finite domains. The problem at hand actually maps to such type of domains
- Better handling of constraints over sets
- Exploitation of recent trends towards combining CP techniques with Linear Programming (LP) ones or other CP techniques by splitting the problem into discrete and continuous parts which are solved independently by utilizing well-known algorithms from these two techniques.
- Better cooperation/integration with the KnowledgeBase which has been developed in Java (while the CLP version of the CP Solver relied on ECLiPSe environment).

The final result is a new solver [KP15] which apart from the above advantages also exhibits the following features:

- Deal with decisions of whether to exploit an external service to realize the functionality of an application component or rely on existing internal implementation of this component which should be deployed on a particular VM whose requirements are provided. We advocate that such decisions must be taken by solving a combined cloud selection problem at the SaaS and IaaS levels. In this way, we avoid independently solving two different problems for each level as there can be inter-dependencies between these levels, like the ones indicated through the decisions to be taken. As such, in the end, our algorithm is able to also discover SaaS services to realize the functionality of application components, something not provided by the other solvers in PaaSage. As this constitutes a unique feature of this solver, it can lead to its selection for execution by the meta-solver whenever the respective SaaS requirements are posed by the application end-user. Such a feature will, however, need to be backed up at the CAMEL model level and especially the deployment model as currently there are no means to express SaaS requirements.
- Use of functions, possibly non-linear, to map low-level (VM or PaaS) capabilities to high-level (application or component) ones. In this way, through the propagation of values from low to high-levels, we will be able to compare the global, high-level requirements posed for an application to the respective capabilities exhibited by the currently selected solution.

More details about how such functions can be constructed can be found in [KP15].

- Combine constraints through logical operators to cater for cases where user requirements are not expressed solely as a set of constraints that all need to be satisfied. For instance, user requirements can be given as a disjunction of conjunctive constraints in order to express requirements on different (service) levels enabling to associate each of these levels to a different cost that the user is willing to pay.

- Deal with high-level security requirements and capabilities in the form of security controls. The matching between this type of security specifications maps to matching security controls sets (i.e., those required and those exhibited by a specific cloud provider). We consider that security control requirements can be posed either at the global application level or at the local application component level. In the first case, any cloud provider whose VM has been selected to deploy an application component should exhibit security control capabilities which match the global requirements set. In the second case, only the cloud provider whose VM has been selected to deploy the component for which the security requirements have been set should exhibit security capabilities that match these requirements.

- Addressing of co-location constraints which can be of the following two types: (a) a component C_1 can or cannot be co-located with another component C_2 at the same VM and (b) a component C_1 can or cannot be co-located with another component C_2 at the same cloud.

- Express optimisation formulas involving utility functions on quality metrics and attributes as well as cost. The main advantage of the new version of the solver with respect to the previous one is its capability to include both continuous and discrete variables into the optimisation formula. In this way, we can deal with objective functions which consider all the objectives/optimisation requirements that have been posed for a particular application, irrespectively of the domain of the metrics mapping to these objectives.

More formally, the CP problem that can be solved by the CP Solver can be expressed as follows:

$$maximize\left(\sum_{q=1}^{Q} w_q * uf_q\left(val_q\right)\right) \tag{1}$$

$$val_q = f_q\left(val_i^q\right) \tag{2}$$

$$\sum_j \sum_k x_{ijk} + \sum_l z_{il} = 1 \tag{3}$$

$$val_i^q = y_i * f_i^q\left(core_i, mem_i, store_i\right) + (1 - y_i) * \left(\sum_l z_{il} * val_{il}^q\right) \tag{4}$$

$$core_i = \sum_{jk} x_{ijk} * core_{jk} \tag{5}$$

$$mem_i = \sum_{jk} x_{ijk} * mem_{jk} \tag{6}$$

$$store_i = \sum_{jk} x_{ijk} * store_{jk} \tag{7}$$

$$x_{ijk} = x_{i'jk} \tag{8}$$

$$\sum_k x_{ijk} = \sum_k x_{i'jk} \tag{9}$$

$$\begin{aligned} &\text{if } \left(y_i \wedge \sum_k x_{ijk} == 1\right) \Longrightarrow cc - ccp_j = \emptyset \\ &\text{else if } (\not{y}_i \wedge z_{il}) \Longrightarrow cc - ccp_{z_{il}.provider} = \emptyset \end{aligned} \tag{10}$$

$$\begin{aligned} &\text{if } \left(y_i \wedge \sum_k x_{ijk} == 1\right) \Longrightarrow seq_j^p \geq seq^p \\ &\text{else if } (\not{y}_i \wedge z_{il}) \Longrightarrow seq_{z_{il}.provider}^p \geq sec^p \end{aligned} \tag{11}$$

Constraint (1) indicates that the main objective is to maximise the weighted sum of the utility functions application on the metric values considered for each high-level metric (including cost). In this sense, we follow the Simple Weighting Additive (SAW) technique [HY81] to transform the multiple optimisation requirements posed by the end-user into a single optimisation objective. In the constraint, Q denotes the total number of metrics used in the end-user's optimisation requirements, w_q denotes the weight given to the metric, while uf_q indicates the utility function used to compute the utility for this metric based on its global value val_q at the application level.

A global metric value can be computed from the respective metric values at the component level through a particular function. This is actually expressed by

Constraint (2), where f_q is the aggregation function and val_i^q is the metric value for the application component i.

Constraint (3) imposes the main set of constraints on the problem's decision variables. In particular, it is indicated that for each application component, we have the choice to either select a particular VM to deploy its internal implementation or to select a particular external SaaS service able to realize its functionality. Obviously, only one of the two choices can be selected and for each choice, only one alternative can be chosen. The main decision variables of the problem are the following:

- y_i which indicates whether the internal service or an external SaaS service will be used to realize component i (i.e., only one of the two possible choices can be made)
- x_{ijk} which indicates whether for component i, the IaaS offering k of the cloud provider j has been selected (internal service selection case).
- z_{il} which indicates whether for component i, the SaaS service l has been selected (external service selection case).

The next constraint (4) expresses the way a metric value for a component is calculated. This calculation depends on the realisation/deployment choice for this component. When the internal implementation of this component is selected, then the metric value is expressed as a function over the respective resources supporting the execution of this component, which are the number of cores and the size of main memory and storage. On the other hand, when an external SaaS service is used to realise the functionality of the component, then the component metric value is actually the respective metric value advertised by this service.

Constraints (5-7) actually propagate the resource values of the selected VM to the respective values of the component in order to facilitate the computation of higher metric values as expressed by the previous constraint. Constraint (5) deals with the number of cores while Constraints (6-7) with the size of main memory and storage, respectively.

Constraints (8-9) express the two types of co-location constraints addressed. The 8th constraint indicates that components i and $i^{'}$ should be deployed on the same VM, while the 9th constraint indicates that these two components should be deployed on the same cloud. More details about how the respective negated co-location constraints are expressed can be found in [KP15].

Finally, Constraints (10-11) express security control constraints at the global application and the local component level, respectively. Constraint (10) indicates that irrespectively of whether an internal service deployment or an external service selection has been decided for each application component, every respective

IaaS or SaaS cloud provider should comply with the global security control requirements posed. Constraint (11) is similar but maps to the case of a single application component indicating that the cloud provider whose service (either IaaS or SaaS) is selected should comply to the security control requirements posed for this component.

The above formal problem representation is simplified in the current variant of the CP solver employed in PaaSage. Once the support for SaaS selection is decided and included as a feature of the PaaSage prototype, the switch to the full variant will be performed.

Current Challenges

One of the challenges [Bar+13] with CP optimisation problems is that resolution time can be high as a huge solution space needs to be explored, thus not covering situations where a deployment solution needs to be calculated very quickly. We are currently considering the following remedies:

1. Restrain the resolution time: this alternative, however, leads to a deployment solution which is not the best possible according to the user requirements provided. As the user requirements are not violated, we can deduce that this alternative is acceptable and can enhance the performance of the reasoner.

2. Exploit the knowledge derived from the Knowledge Base [Kri+14]: instead of considering the whole solution space, the reasoner can exploit the fact that the execution history for the application at hand or for equivalent applications with this application is available and the Knowledge Base has deduced the best deployments for these applications, such that only these deployment solutions are explored in order to solve the optimisation problem. This alternative seems even better than the previous one as (a) the reasoner relies on real, summarised data derived from the applications' execution history and not data which might have been advertised by a particular cloud provider and (b) the solution space is significantly restrained.

Obviously, when no execution data are available for a particular application (which is not similar or equivalent with any other application), the second alternative cannot be applied, so the first one should be picked up, provided that the solution space is large.

The first challenge is already supported by the CP solver and is planned to enable its support through a configuration parameter. The second challenge is under implementation and will be finalized in the beginning of the fourth year of the project.

Implementation

The current implementation of the CP Solver relies on Choco (`choco-solver.org`), a free Constraint Satisfaction Optimisation Programming solver supported by a very active community and competing well-known proprietary solvers. This solver supports all types of variabes required, mapping to the different types of metric domains expected, and has implemented well-known state-of-the-art constraint types (e.g., *all different*) and search strategies. Real variables are addressed through the Ibex CP engine (`www.ibex-lib.org`), offered as a C++ library, which relies on interval and affine arithmetic and is able to handle non-linear constraints and roundoff errors.

The CP Solver is integrated into the Upperware as it supports the (programmatic) transformation of Choco models from to CP models which comply to the Upperware CP meta-model. In this sense, the respective input to successfully call the solver as well as the respective output to be consumed by the next components in the Upperware flow is guaranteed. The solver has been already tested with particular input produced by the CP Generator which maps to specific PaaSage use cases.

As indicated above, the integration with the Knowledge Base is planned for the beginning of the fourth year. The same holds with respect to the integration with the MetricsCollector component of the Executionware such that fresh metric values will be considered for the re-execution of the solver for the same application deployment planning problem.

5.4 Greedy Heuristics

Overview

Exact solution solvers (*e.g.* MILP and CP Solver) always give a set of optimal solutions. However, when the search space increases, the time to compute those solutions grows exponentially. This happens because the application placement problem is a generalization of the bin packing problem which is **NP-Hard**, meaning that a polynomial time algorithm to solve it is unknown.

Thus, when using a multi-cloud platform and a medium to large application, solvers might not be able to scale and give a solution within reasonable time. A common approach to solve this kind of problem is the usage of **heuristics** which can find near optimal solutions in feasible time. The major drawback is that it has to be developed and/or specialized for each class of application.

In the state of the art there are two main types of heuristics used to solve the placement problem, namely greedy heuristics and meta-heuristics [Cha14;

LKK99; GZ13; Jea+13; Kum+11; MNC11; Pan+11; Sha+11]. Due to its scalable characteristics and ease of implementation, we are interested on the former.

The **Heuristic Based Solver** component calculates a good solution for a N-Tier application placement problem in feasible time by means of a set of heuristics based on First Fit Decreasing and Best Fit for solving the placement problem.

Design

In short, the first fit decreasing algorithm – applied to the N-Tier application placement problem – selects tier instances, organized in decreasing order of size, and assigns them to virtual machine (VM) instances sufficiently large for them to be placed on. If there is no such VM instance, a new one sufficiently large is rented, chosen from a list of VM types, possibly from different cloud providers. Best Fit works in a similar manner, but it also sorts VM types and available VM instances in increasing order of size and cost aiming at assigning the largest tier instances to the smallest suitable VM instances.

The algorithms are straightforward, however, there are some peculiarities concerning their application to Paasage that must be discussed.

Firstly, the notion of size is associated to the amount of available resources or capacities in the case of VM instances or VM types and to the amount of required resources for tier instances. It is possible to think of tier instances, VM instances or VM Types as **multi-dimensional vectors** where each dimension is the amount of required or offered resources. Furthermore, as it will be necessary to sort those multi-dimensional vectors, a comparison strategy, known as a measure [GZ13], is needed. Finally, rented virtual machines are instances derived from VM types and each one has a renting price that must be taken in consideration when sorting them.

We use a group of **seven heuristics**, adapted from the state of the art bibliography to the needs of Paasage. Six of them are based on first fit decreasing, namely First Fit Decreasing Weighted Sum, First Fit Decreasing Priority, First Fit Decreasing Average Sum, First Fit Decreasing Exponential Sum, First Fit Decreasing Random Bins N Times and First Fit Decreasing Windowed Multi-Capacity, and one is based on best fit called Best Fit Dot Product.

First Fit Decreasing Based Heuristics All first fit decreasing based heuristics (Algoritm 2) share the same basic structure differing mostly on the **measure** used, which interferes on the way elements are sorted (Line 4 from Algorithm 2). As briefly discussed before, a measure, is a method for assigning a size (or score) to the multi-dimensional representation of a tier instance, VM instance or VM type.

Algorithm 2: First Fit Decreasing Algorithm

Require: $\mathcal{C}$ – set of tier instances, $\mathcal{N}$ – set of V.M. types
Ensure: Placement of tier instances on V.M. instances
1: $\mathcal{B} \leftarrow \emptyset$ {Instanced Virtual Machines}
2: $\mathcal{P} \leftarrow \{\}$ {Placement configuration}
3: $SortVMIncreasingByCost(\mathcal{N})$
4: $SortItemsDecreasing(\mathcal{C})$
5: **for** $c \in \mathcal{C}$ **do**
6: $b \leftarrow FindSuitableBin(c, \mathcal{B})$
7: **if** b is null **then**
8: $b \leftarrow FindSuitableBin(c, \mathcal{N})$
9: **if** b is null **then**
10: **return** $\{\}$
11: **else**
12: **Insert** b on $\mathcal{B}$
13: **end if**
14: **end if**
15: **Remove** c from $\mathcal{C}$
16: **Insert** the pair (c, b) on $\mathcal{P}$
17: **end for**
18:
19: **return** $\mathcal{P}$

Next, we go through a brief overview of each first fit decreasing based heuristics that we employ.

First Fit Decreasing Weighted Sum [GZ13] This heuristic uses the weighted sum of the resource requirements or capacities as measure.

First Fit Decreasing Priority [GZ13] This heuristic uses the maximal normalized resource requirements or capacities as measure.

First Fit Decreasing Average Sum [Pan+11] This heuristic uses the average sum of resource requirements or capacities as measure.

First Fit Decreasing Exponential Sum [Pan+11] This heuristic uses the exponential sum of resource requirements or capacities as measure.

First Fit Decreasing Random Bins N Times As its name suggests, it arranges the virtual machine types randomly and assigns items, sorted in decreasing order, to them. The experiment is repeated N times, with N depending on the number of virtual machine types.

Algorithm 3: Best Fit Dot Product Algorithm

Require: $\mathcal{C}$ – list of tier instances, $\mathcal{N}$ – list of V.M. types
Ensure: Placement of tier instances on V.M. instances

$\mathcal{B} \leftarrow \emptyset$ {Instanced Virtual Machines}
$\mathcal{P} \leftarrow \{\}$ {Placement configuration}
$\mathcal{D} \leftarrow CalculateDotProducts(\mathcal{C}, \mathcal{N})$ {Dot Product $\mathcal{C} \times \mathcal{N}$}
$\mathcal{D}' \leftarrow \emptyset$ {Dot Product $\mathcal{C} \times \mathcal{B}$}
for $c \in \mathcal{C}$ **do**
  if $\mathcal{B} \neq \emptyset$ **then**
    $b \leftarrow FindMostSuitedVmForTierInst(c, B, D')$
  end if
  if b is null **then**
    $b \leftarrow FindMostSuitedVmForTierInst(c, \mathcal{N}, D)$
    if b is null **then**
      return $\{\}$
    else
      Insert b on $\mathcal{B}$
    end if
  end if
  Remove c from $\mathcal{C}$
  Insert the pair (c, b) on $\mathcal{P}$
  $UpdateDotProduct(\mathcal{C}, B, D')$
end for

return $\mathcal{P}$

First Fit Decreasing Windowed Multi-Capacity [Kum+11] uses a different strategy based on balancing the resource usage on each dimension of VM instances.

Best Fit Dot Product The Dot Product strategy [Pan+11; GZ13] proposes an approach to calculate the size of an element by using a weighted dot product between tier instances dimensions and VM instance or VM type dimensions. The basic idea is to calculate the "best suited" VM instance to place each tier instance by using the dot product as measure for comparison. The outline of this algorithm is illustrated in Algorithm 3.

Heuristic Based Solver After having presenting the heuristics individually, let now describe how the **Heuristic Based Solver** (HBS) works. The objective of

Algorithm 4: Heuristic Based Solver Algorithm

Require: $\mathcal{C}$ – list of tiers, $\mathcal{N}$ – list of V.M. types
Ensure: Problem configuration and placement

```
1: P ← GetAllPossibleProblemConfigurations(C)
2: S ← ∅
3: for p ∈ P do
4:    for each considered greedy heuristic h do
5:       s ← CalculatePlacement(h, p, N)
6:       if s ≠ null then
7:          insert (s, p) on S
8:       end if
9:    end for
10: end for
11:
12: return LessExpensive(S)
```

HBS is to calculate a near optimal placement that meets **application performance** constraints without exceeding a **budget**. To do that, as the time consumed by an heuristic to solve an instance of a placement problem lies on milliseconds, HBS solves a given placement problem using all seven heuristics discussed previously but only keeps the best solution.

Algorithm 4, summarizes the mechanisms of this algorithm as it is applied to Paasage. One can notice that HBS does not receive a list of tier instances as input like the heuristics discussed above, but instead, it receives a list of tiers. This is because the input of HBS is given by the CAMEL model and the CP model, *i.e.,*, it receives a **list of tiers**, a list of minimum and maximum number of tier instances per tier and a list of VM types. This means that the algorithm does not know *a priori* how many instances of each tier will need to be placed neither what is the **problem configuration** to be solved. The only available information is that, among the numerous possibilities of problem configurations derived from tiers and their associated minimum and maximum number of instances, there is a set of configurations that allow for a less expensive placement while meeting application constraints.

To solve this, again, taking advantage of the **low execution times** of our heuristics, HBS generates all possible problem configurations and the respective placements for each one of them. The output is a placement problem configuration and a near optimal solution for this problem. It is important to notice that the found configuration is one of those whose placement is the less expensive among all other configurations.

It is important to notice that this straightforward approach to solve this problem was only possible because of the efficiency of the greedy heuristics handled by HBS. Preliminary evaluation of the greedy heuristics on experimental sets showed that even using large data sets as inputs, they manage to give near optimal solutions in feasible time.

Implementation

The current version of HBS prototype was written in Python with a Java layer to read from CDO (CAMEL model and CP model) and to write into it (a solution of CP model).

5.5 Simulator Wrapper

Overview

The goal of this component is to provide to the PAASAGE platform a solver that determine the best deployment solution using a cloud simulator. `Simulator Wrapper` retrieves informations from the CAMEL model to generate an application mockup, namely a scenario like 3-tier app, and explore a range of parameters corresponding to the application requirements. To run a scenario, virtual machines are instantiated on a simulated cloud platform that is generated from the cloud provider model (VM type, price, etc.) and which performance and behaviour can be improved thanks to the MDDB. Using analytic performance model and/or historical data, `Simulator Wrapper` simulates the resource consumption of the different workloads of the application and computes a set of metrics and trade-offs between them. Using this information, it can rank the different mappings between cloud resources and applications components and transfer to the meta-solver the generated Upperware meta-model. Figure 23 describes how `Simulator Wrapper` component works internally.

Design

Layer #1 To accurately simulate an application on a cloud, it is required to have a good understanding of the application components and their interactionsm, as well as the performance characteristics of the cloud platform. The first step executed by `Simulator Wrapper` is to select a scenario that is compliant with the PAASAGE application model. To do so the PAASAGE application model is transformed into internal application model (see Figure 24). Each class of application relies on a specific dataflow which is obtained form Informations about the cloud infrastructure are retrieved from the MDDB.

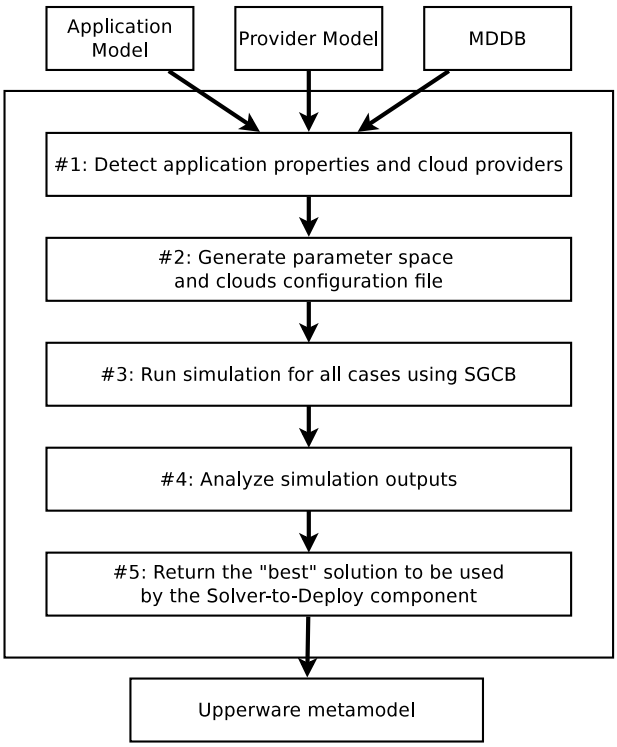

Figure 23: Internal architecture of the Simulator Wrapper.

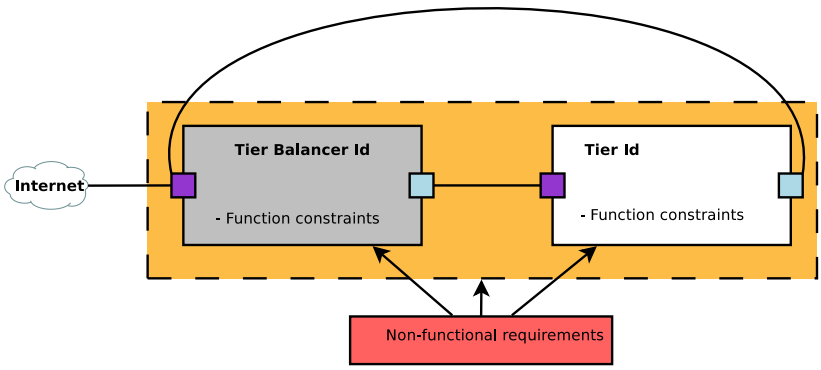

Figure 24: Generic application model.

Layer #2 Once the scenario has been selected, a parameter range is generated using requirement model of the application to explore the different deployment solutions: number of VM instances for each components, hardware parameter of virtual machines that are hosting the components, localization of cloud provider datacenters. During this step, a specific cloud topology file and cost model for

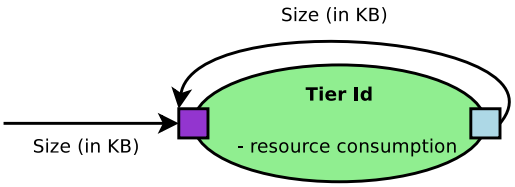

Figure 25: Generic request's dataflow model.

the usage is generated, *i.e.,* the Cloud offers (VM types, storage solutions, prices, etc.).

Layer #3 This layer is a wrapper for the SimGrid Cloud Broker (SGCB) Cloud simulator [DR13]. This toolkit is based on the well-known SimGrid framework which has been extended to be able to simulate virtual machines behavior. For each parameter combination, a run is executed and trace files are stored in a specific directory, which can be reused later as a cache.

Layer #4 Layer #4 parses output files from the simulator in order to select combinations that meet performance requirements, *e.g.* the required response time from a client, and returns the cheapest solution within them.

Layer #5 This final layer simply convert the selected solution to a solution into the CP Upperware model that will be used by the `solver-to-deployer` component.

Implementation

Currently, all layers have a prototype implementation that allows to test every step of the solving procedure, with some restrictions. First, as data for cloud provider topology is not in the MDDB, simulator uses a generic topology file obtain computed from experiments on the Amazon Web Service platform. Second, the dataflow and component behavior of an n-tier application requires benchmarking, which has only be done for a few cases. A working example is presented below for the Rubbos application (3-tier scenario). The BeWan use case is currently studied to obtain the SGCB model.

Example of a 3-tier application

Rubbos application model It uses RUBBoS[16] as test application, that is composed of an HTTP front-end, an application server back-end and a database.

[16] `http://jmob.ow2.org/rubbos.html`

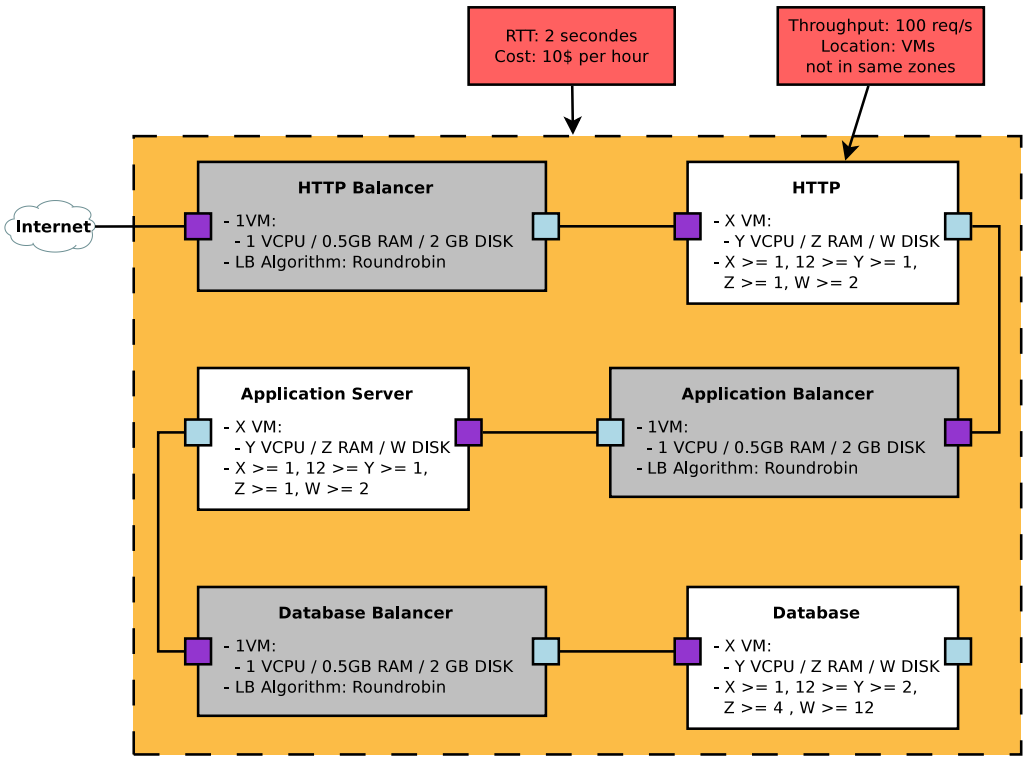

Figure 26: Generic application model for the RUBBoS application.

Moreover, for each tier, there is a tier load balancer that spreads the incoming requests between the different instances of each components. Figure 26 presents a generic description of the application.

The application workflow reads as:

- For each Tier Server
  1. Register to Tier Balancer
  2. Launch X Tier Process (one per core by default)
  3. Process requests
     a) Receive requests
     b) Read data
     c) Compute requests
     d) Write data
     e) Send to next tier (optional)
  4. De-register and die

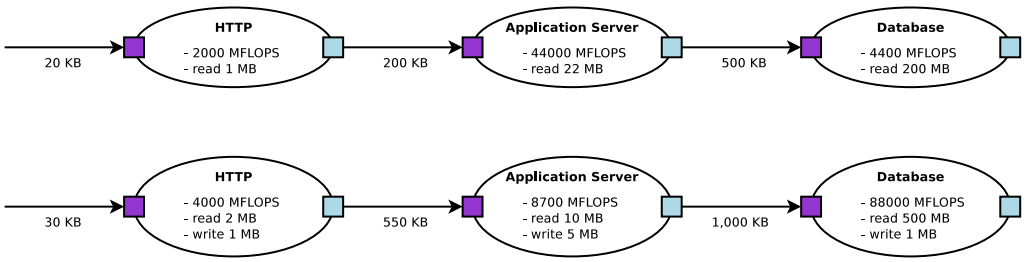

Figure 27: 2 request types' dataflow for the RUBBoS application.

- For each Tier Balancer
  1. Receive registering requests
  2. Process requests
     a) Receive requests
     b) Elect a tier server
     c) Send to the selected tier server

See Figure 27 for more details.

Cloud topology and pricing policy The layer #2 of the `Simulator Wrapper` component is also in charge of generating the platform configuration file. In the case of SGCB, the file is divided into 2 parts: one for the **topology** and one for the **Cloud offers**. The topology file describes all the network links (latency and bandwidth), the physical machines (speed and number of cores, amount of memory and storage, etc.) and their interconnections. An example of such file is given in Listing 2. As one can see, the example describes a single Cluster with an output bandwidth of 1.25 GB/s, composed of 50 physical nodes with 4 cores, 15 GB of memory and 1,690 GB local hard drive.

The **Cloud offers** file describes all the different VM types and their characteristics but also the other services, *e.g.* storage. An example of such file is given in Listing 3. The example contains the description of the Cloud offering of one VM type and the price of Internet data transfer. Moreover, the layer #2 of the simulator wrapper is also in charge of generating Java code based on the platform performance models. Thanks to the modular architecture of SGCB, it is simple to plug new platform performance models. Using this interface, the simulator wrapper can generate on-the-fly code that describes and implements the performance models of the targeted platform.

Listing 2: Example of the platform topology

```
<cluster bb_bw="1.25E9" bb_lat="1.0E-4" bw="1.25E8" core="4" id="
    AS_usa_west_1_pm_m3.xlarge" lat="1.0E-4" power="1.5354508E10"
    prefix="usa_west_1_pm_m3.xlarge-" radical="1-50" suffix=".
    usa_west_1.broker.simgrid.org">
<prop id="memory" value="15000"/>
<prop id="disk" value="1690"/>
</cluster>
```

Listing 3: Example of the Cloud offering.

```
[...]
 <instance id="c1.xlarge" vcpu="8" computing_unit="2.5" memory="7000"
     disk="1690">
                    <on_demand_price price="0.580"/>

  </instance>
[...]
<data_transfer_prices>
                 <internet>
                     <input>
                         <range price="0.0"/>
                     </input>
                     <output>
                         <range begin="0" end="1" price="0.0"/>
                         <range begin="1" end="10000" price="0.120"/>
                         <range begin="10000" end="40000" price
                             ="0.090"/>
                         <range begin="40000" end="100000" price
                             ="0.070"/>
                         <range begin="100000" price="0.050"/>
                     </output>
[...]
```

Internal representation of the Rubbos instance For each combination of the parameter sweeper, a configuration file is generated that is used by the simulator, *i.e.,* for each component to resource mapping. See Listing 4 for an example. The executions traces are stored in a specific directory and will then be used by the layer #4

Trace analysis Then the `Simulator Wrapper` component retrieves the execution traces of all the simulations. In the case of SGCB, the traces are in Paje binary format[17]. The traces are analysed by the layer #4 of `Simulator Wrapper` . These traces contain the billing of all resources, virtual resources' information, *e.g.* startup duration of each VM and overall and per request application execution information. An example of such trace is given in Listing 5.

[17] `https://github.com/schnorr/pajeng`

Listing 4: Example of a configuration file

```
<nTierApplication version="1">
[...]

  <proxy>
    <webProxy region="eu_1" instanceType="m1.small"/>
    [...]
  </proxy>

  <services>
    <webService>
      <region name="eu_1">
        <instance type="VM_TYPE_TIER_1" quantity="INSTANCE_NB_TIER_1
            "/>
      </region>
    </webService>
    [...]
  </services>
</nTierApplication>
```

This trace represents the different states, *i.e.,* step in the request's dataflow, of a request (`REQTASK_STATE_mwthanr_439`). For each state, it contains where the state has changed, *e.g.* on the `eu_1.m2.2xlarge.14` resource: for example, the request has changed its state to 16. Moreover, the trace also contains when each state has begun and ended and its duration. Accordingly, it is possible to recreate a complete view of the execution of each request. This layer finalley returns metrics and price values `Simulator Wrapper` also produces trade-off curves between the different metrics. Finally, based on this trade-off, the layer #4 of the `Simulator Wrapper` component ranks the different component to resource mappings.

Selecting the "best" deployment solution According to the performance model (cf Figure 28), the layer #4 determines which parameter combinations corresponds to the trade-off between price and round-trip time (RTT). Finally, layer #5 transmits stores the solution into the Upperware model.

5.6 Meta-Solver

Overview

The Meta-Solver acts as a gateway to the Solvers which provide the reasoning in the PaaSage platform. To date we have five main Solvers which are the MILP Solver, CP Solver, LA Solver, Greedy Heuritics, and SimWrapper. Each one approaches the task of finding an optimal PaaSage deployment in a different way. In year 2 of the project the Meta-Solver was triggered as part of a one way from

Listing 5: Example of the pre-analysis trace of a simulation.

```
Variable, node-1.broker.broker.simgrid.org, REQTASK_STATE_mwthanr_439,
     56336.3, 65670, 9333.74, 0
Variable, eu_1.m1.small.2, REQTASK_STATE_mwthanr_439, 56345.7, 65670,
    9324.31, 15
Variable, eu_1.m1.small.1, REQTASK_STATE_mwthanr_439, 56338.2, 65670,
    9331.84, 8
Variable, eu_1.m1.small.0, REQTASK_STATE_mwthanr_439, 56337.9, 65670,
    9332.19, 1
Variable, eu_1.m2.2xlarge.14, REQTASK_STATE_mwthanr_439, 56345.8,
    56345.8, 0.041315, 16
Variable, eu_1.m2.2xlarge.14, REQTASK_STATE_mwthanr_439, 56345.8,
    56345.8, 0.002897, 18
Variable, eu_1.m2.2xlarge.14, REQTASK_STATE_mwthanr_439, 56345.8,
    56348.7, 2.86561, 19
Variable, eu_1.m2.2xlarge.14, REQTASK_STATE_mwthanr_439, 56348.7,
    56348.7, 0.008236, 20
Variable, eu_1.m2.2xlarge.14, REQTASK_STATE_mwthanr_439, 56348.7,
    65670, 9321.37, 21
Variable, eu_1.m2.2xlarge.8, REQTASK_STATE_mwthanr_439, 56338.2,
    56342.8, 4.61757, 9
Variable, eu_1.m2.2xlarge.8, REQTASK_STATE_mwthanr_439, 56342.8,
    56342.8, 0.002897, 11
Variable, eu_1.m2.2xlarge.8, REQTASK_STATE_mwthanr_439, 56342.8,
    56345.7, 2.86561, 12
Variable, eu_1.m2.2xlarge.8, REQTASK_STATE_mwthanr_439, 56345.7,
    56345.7, 0.008236, 13
Variable, eu_1.m2.2xlarge.8, REQTASK_STATE_mwthanr_439, 56345.7,
    65670, 9324.33, 14
Variable, eu_1.m2.2xlarge.3, REQTASK_STATE_mwthanr_439, 56337.9,
    56337.9, 0.007831, 2
Variable, eu_1.m2.2xlarge.3, REQTASK_STATE_mwthanr_439, 56337.9,
    56337.9, 0.00029, 4
Variable, eu_1.m2.2xlarge.3, REQTASK_STATE_mwthanr_439, 56337.9,
    56338.2, 0.286561, 5
Variable, eu_1.m2.2xlarge.3, REQTASK_STATE_mwthanr_439, 56338.2,
    56338.2, 0.008236, 6
Variable, eu_1.m2.2xlarge.3, REQTASK_STATE_mwthanr_439, 56338.2,
    65670, 9331.87, 7
```

profile to execution invocation chain managed by a master script. Appropriate solvers were selected based on the judgement of Meta-Solver over deployment requirements in the CAMEL model being either a set of linear or non-linear constraints.

In year 3 the Meta-Solver is triggered either via a direct call from the Rule Processor in the case of an initial solution request or Adaptor for a new solution request. Integration with the ZeroMQ Paasage messaging architecture also enables the Meta-Solver to make new solution requests based on live data and deployed model constraints.

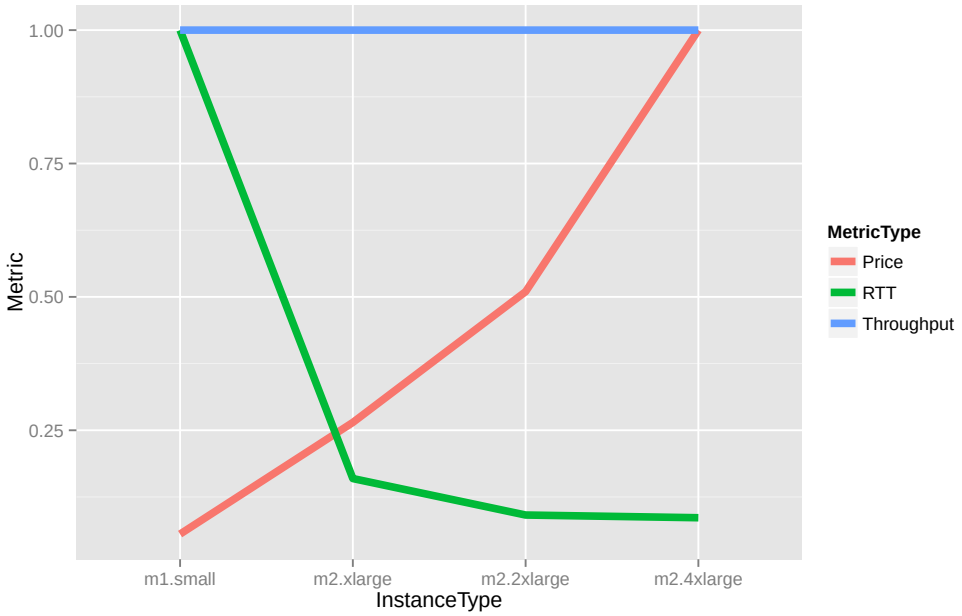

Figure 28: Trade-off between the metrics for the RUBBoS application and the horizontal scalability of the application tier.

Design

During year 3 the Meta-Solver has been adapted to take into account 3 main functions:

1. Communication via ZeroMQ
2. Modification of models to incorporate monitored metrics
3. Continuous invocation of Solvers

Communication via ZeroMQ is a wider architectural change in the project and has focused on using the messaging technology to propagate monitored metrics across the PaaSage architecture. The Meta-Solver subscribes to messages from the Reasoning components present in the Solvers using ZeroMQ and also the Upperware with messages received from the Adaptor and Metrics Collector. The message relationships can be seen in Figure 29.

Messages received from the Solvers notify the creation new deployment models and are expressed as references to CAMEL. Although the Solvers in the architecture are invoked by the Meta-Solver using non ZeroMQ calls the results of the solvers are transmitted to the Meta-Solver via ZeroMQ. This is because

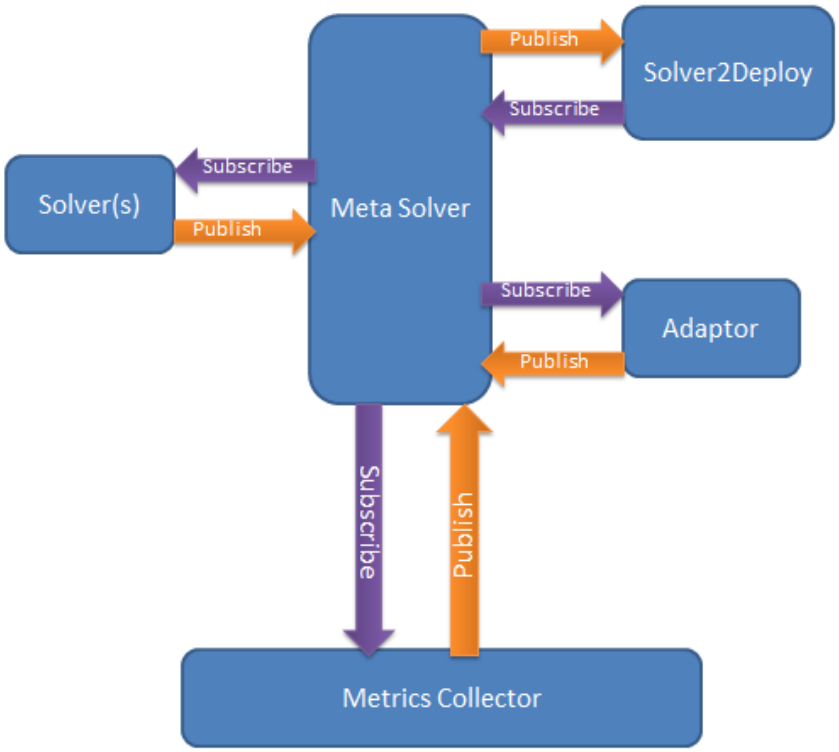

Figure 29: Meta-Solver ZeroMQ Interaction

the results from the solvers are likely to be more frequent and time specific. Thus using ZeroMQ the Meta-Solver can process them as they arise in as close to real time as possible. The format for these messages is notification that a specific solution is available and its reference in the CDO server.

The messages received from the Metrics Collector are direct values taken from monitored data related to the infrastructure that deployed application models reside on. The format of the message as illustrated in Table 2 contains the metric name, its value and corresponding model reference in the CDO server.

Implementation

Support for ZeroMQ has been added to the Meta-Solver to enable the adaptor or Rule Processor to send messages via ZeroMQ to start solving on specific models. This is a difference from year 2 where the requests were trigger sequentially by the Master Script. In year 3 the trigger for these requests is either based on direct input from the user (i.e. via the social network) or by analysis of monitored data from the Metrics Collector. The Executionware can also trigger the Adaptor into

Solution Notification (Solver to Meta-Solver)	MQ: "(SOLVERNAME)solutionAvailable", Reference to Model in CDO (String)
Model to Deploy Request (Meta-Solver to Solver2Deploy)	MQ: "startDeployment", Reference to CPModel (String), Reference to Model in CDO(String)
Metric Notification (Metrics Collector to Meta-Solver)	MQ, metricName(String), value (String), Reference to model in CDO
Solve Model Request (Adaptor to Meta-Solver)	MQ: "startSolving", Reference to Model in CDO (String), Reference to CPModel (String)

Table 2: ZeroMQ message integration with the Meta-Solver.

a redeployment action on the Meta-Solver. The format of this message is the request to start solving on a specific model that a reference in the CDO server is supplied for.

The only component the Meta-Solver publishes data to via ZeroMQ is the Solver2Deployer component. Here, the Meta-Solver notifies the Solver2Deployer of the new solutions generated by the Solvers. The Meta-Solver sends this notification if the new Solutions presented by the solvers is different from existing solutions (it is expected that some solvers will return the same result). The content of the message here is the reference to the new model and request to deploy.

Modification of models by the Meta-Solver is a new function for year 3. In the previous year model modification was conducted by the Solver. Now the Meta-Solver adds data values into the models using the metrics provided by the ZeroMQ link to the Metrics Collector. Adding live metrics to the models enable the Solvers to take into account live values such as current memory use, thus enabling the Solvers to create new solutions in the case where constraints are breached.

Finally, the continuous invocation of the Solvers is part of the ongoing use of live metrics and to support possible new solution requests as the Solvers are executing. In year 2 solvers were only invoked once. The support for multiple invocations enables the platform to keep a constant stream of up to date solutions available to support rapid redeployments in the platform. For example, in the case of the LA Solver it will continuously learn and adapt to the state of the deployment via its own ZeroMQ link to the Metrics Collector.

5.7 Utility Function Generator

Overview

Recall from Section 5.1 that a feasible configuration is defined as an assignment of variable values that satisfies all the constraints of the deployment problem. Thus only by addressing the user's preferences and goals can we distinguish between these feasible configurations. The only purpose of the *utility function* is consequently to rank the various feasible configurations by assigning a numerical value to each feasible configuration such that the configuration with the largest utility value is the "better" configuration seen from the user.

The utility function is the objective function for mathematical solvers, and serves to train learning based solvers. Learning is by definition an iterative process, and learning from trial and error in the real world will be unfeasible. Most of the learning iterations will therefore be performed against the utility function similar to the *hybrid reinforcement learning* approach used by Tesauro *et al.* for autonomic resource allocation [Ger+06].

The concept of *utility* is well established in economics as a representation of preferences over some goods and services, and has long been used for decision making [Pet70]. Chu and Halpern showed that essentially all decision rules can be represented as a generalised notion of expected utility [FJ04]. Arguably, one of the first applications in computer science was when Sutherland designed an auctioning system where users could bid for computer time depending on their perceived utility of the computation [I E68]. The concept of utility for system management and operation is closely linked with the vision of autonomic computing [JD03] and was first used to allocate resources in a data centre [Ter03], an application close to the allocation problems considered in PaaSage. Kephart and Das argued that both rule based policies and goal policies are in general insufficient and inflexible for decision making in autonomic systems when one has to trade-off potentially conflicting goals, and showed how utility functions could be seen as an extension of goal policies [JR07]. Furthermore, Walsh *et al.* have shown how the attributes of high level services could be expressed in the utility function in high-level business terms [Wil+04].

Utility functions were used for self-adaptive applications in the MADAM project [Kur+09], seeding ideas that were carried forward for context aware ubiquitous computing systems in the MUSIC project [Sve+12]. Extensive experimentation with real applications revealed that even experienced software developers would find it difficult to develop utility functions [Jac+06], and that they often reduced the utility function to some kind of situation-action rule [Jac+13]. These experiments thus confirmed the conjecture of Walsh *et al.* [Wil+04] who stated that *"(...) humans will often find it difficult to express their utility for various components of a large, complex system. Carefully designed interfaces and*

preference elicitation techniques are needed to represent human notions of value accurately."

The DiVA project developed a model to assist the user in specifying implicitly the utility function as a sum of "properties" to be optimised under the current configuration where the terms were weighted by a discrete priority factor like "high", "medium", or "low" [FA09]. Although the configurations were ranked by the utility function, different priority dimensions were balanced using priority rules. However, Cheng *et al.* realised that when more than one dimension must be considered for adaptation *"(...) updating and maintaining consistency between the trade-off preferences quickly becomes unmanageable"* [SDB06].

Another approach to elicit a utility function was offered by Valetto *et al.* [GPD11]. They instrumented the application with monitors extracting data on various performance features in laboratory tests, and then tried to correlate the features impacting the application utility. However, this approach requires significant off-line testing of the application in the laboratory, and significant manual effort in the statistical analysis of the recorded data.

Finding the optimal configuration is relatively easy if the utility function is a linear combination of independent utility functions for the application's artefacts like components, modules, virtual machines, and similar. The utility function of Kephart and Das was of this kind [JR07]. The best configuration can then be found in polynomial time by an application of the Bellman-Ford algorithm [Ric58] as explained in [Mou+06]. However, the experiments in MUSIC showed that decomposable utility functions cannot be expected in general [Jac+13].

The approach to elicit and build a suitable utility functions currently under investigation in PaaSage resembles the approach in DiVA where the user's goals will be captured as discrete preference sets over specific system properties. The user will not be asked to formulate the utility function directly, but rather provide certain rules for how the user would assess the application's utility under different situations. As an example consider that one can measure performance and estimate the cost of the execution of a deployment. The user might then specify the perceived utility as in the following examples.

R1: **if** *performance* is *acceptable* **then** *utility* is *acceptable*

R2: **if** *performance* is *bad* **then** *utility* is *very low*

R3: **if** *cost* is *high* **then** *utility* is *low*

The salient feature is that the different factors influencing the utility like performance and cost, are classified and based on this classification the utility can

take values from subsets of all possible utility values. This is different from the approach in DiVA where rules specified directly the property to be prioritised, e.g. “if the battery runs low, the power consumption should prioritised over performance” [FA09]. In PaaSage this is done as an implicit trade-off between the various properties based on the parts of the utility value chosen for that property value. One can for instance assume that the user setting the rules R1–R3 above prioritises performance over cost since the utility will never be assessed as worse than “low” even if the cost is “high” whereas the utility could be taken as “very low” if the performance is “bad”.

Implementation

Our intent is to use *fuzzy numbers* [JE02] to establish a utility value in the unit interval, $[0, 1]$. This approach entails the following steps that are illustrated in Figure 30.

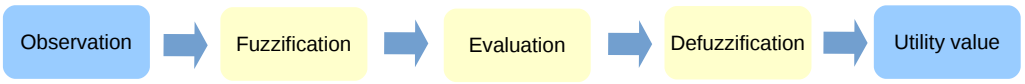

Figure 30: The steps needed in order to compute the fuzzy utility value.

1. **Observation** of the values of the property monitors. For each configuration there are monitors stating how property values should be obtained, and so in the example there must be ways to measure “performance” and “cost” for the example rules above. In a real and simulated deployment, the way to monitor the properties will be given by the infrastructures or the simulator. In a utility function setting these values must be estimated based on the configuration choices and historical information. One can for instance assume that a large virtual machine with many cores will give better performance than a smaller one, and from the price tables of the Cloud providers and passed executions one can monitor the cost of a virtual machine as a random variate with a given probability density function.

2. **Fuzzyfication** is the process of mapping the observed property value to one of the possible alternatives. Each alternative is represented as a *fuzzy number*. A fuzzy number is a fuzzy subset of the real numbers $\mathbb{R}$ and has a *membership* function indicating on a scale from zero to unity whether a given value *is* the fuzzy number. For instance the fuzzy number “bad performance” will have a membership function that covers low values of performance but drops off to zero as the performance value increases. An observed value may consequently map to more than one fuzzy number if

their membership functions overlap, and can therefore fractionally represent several fuzzy numbers. This reflects the underlying variability of the observations. In our example, the specific value measured for the *Performance* property can belong, say, 0.65 to the fuzzy number "acceptable performance", and 0.35 to the fuzzy number "bad performance".

3. **Evaluation** of the rules uses the fuzzy values to assess the conditions of the rules. Continuing our example, this implies that we should "weight" the outcome of the first rule with 0.65 and the second rule with 0.35 since the measured value indicates that the performance *is* mostly acceptable, but may also be bad. These conclusions can be mapped to utility values by inverting the membership functions of the fuzzy utility values so that, in our case, the utility resulting from the first rule is the real value which has a membership value of at least 0.65 to the fuzzy number "acceptable utility". The "very low utility" set of the second rule may cover a different range of utility values, and again the utility value is the one whose membership value to this fuzzy number is larger than, or equal to 0.35.

4. **Defuzzification** is combining the evaluation of different rules into one single utility value. It is not a simple sum of the different outputs of the rule evaluation since the fuzzy numbers can overlap, e.g. the upper values in the "very bad utility" coverage can overlap with some of the lower values in the "bad utility". Popular ways of doing defuzzification is the *mean of maxima* technique, the *centroid* technique, or the *centre of maxima* technique. The most intuitive one is the centroid technique that will be tried first in PaaSage. It aims to find the "centre of gravity" of the membership functions of the outcome of the evaluation of the different rules. It integrates the area under the membership functions of the resulting fuzzy numbers, but truncates these at the level at which the rule made the decision, and averages. In the example at hand, one would integrate the membership function for the "acceptable utility" limiting the membership values to 0.65, integrating the "very low utility" membership function limiting the values to 0.35, and integrating the "low utility" to the level of decision of rule number three. The single utility value returned will be the one splitting this combined area in two equal parts (the center of gravity).

In order to use this approach, fuzzy numbers must be defined for the the input properties, as well as for the different utility classes. For each fuzzy number there must be a membership function defined. It is clear that the range covered by each of these numbers and the corresponding membership function chosen will influence the produced utility value. These choices are therefore crucial to the usefulness of the fuzzy utility function.

The literature reports, however, that most of the fuzzy number membership numbers are specified as either triangles or trapezoids, and as such it could be that these functions will be good enough for PaaSage where the utility value has no precise meaning in. It is only required to give a consistent ranking of the deployment configurations.

Work has therefore started on evaluating this approach with the PaaSage end-users to understand what will be the easiest way to formulate the utility function, either directly or through the above outlined use of fuzzy numbers. This will continue up to month 18. The fuzzy utility function will thus only be included in the second phase of the project to be demonstrated at month 36.

5.8 Flexiant Utility Function Cost Trigger

Overview

As a study on the improved relationships beteen the Upperware and a Cloud provider service, a cost related 'Trigger' has been created by Flexiant. It exposes cost related information from within a user FCO account which can then be used by the 'Utility Function'.

Firstly, to define a Trigger, they are written as a block of Flexiant Development Language (FDL) code that runs either before an event occurs (a pre trigger) or after an event occurs (a post trigger). These are executed from within the Cloud platform itself and allow an action in Flexiant Cloud Orchestrator to initiate a second action, which can be external to Flexiant Cloud Orchestrator. These are also highly exploitable areas of technology.

The developed cost trigger is called once a VM is shutdown within the PaaSage user account. The trigger does this by comparing a required key set at a user account level once a VM is shutdown with FCO, if this key matches the trigger continues. Next the cost information is pulled using the FCO API to list the total credits used that day, using the listUnitTransactionSummary API call[18].

This cost information, server UUID and timestamp is then emailed to the PaaSage account owner for their information, or can be passed to any PaaSage component such as the 'Utility Function'. In the background, a script gauges the rate of various PaaSage cost related assets within FCO and converts this to a usage-versus-time amount.

The Trigger is for use in PaaSage by the use cases when taking advantage of the Utility Function. Cost is a key metric for decision making and will be one of several handled by the Utility Function. Architecturally, the Upperware will tell the Executionware what to monitor including cost. Cost should be [a] metric(s)

[18] `http://docs.flexiant.com/display/DOCS/SOAP+User+listUnitTransactionSummary`

that is taken from the Metric Collector via a probe on FCO which picks up the output of the Trigger.

The Executionware should feed back the monitored data to the MDDB/Upperware where it should also be picked up by CAMEL, and the Utility Function will use it along with other relevant metrics for a specific use case.

Further improvements

To further improve the cost information for use within the Paasage project, additional work could be used to develop the existing trigger.

The trigger could feed the exposed data straight from the FCO platform to the Upperware rather than passing this information via email.

In addition more detailed cost information could be provided by using the listUnitTransactions API call2, which provides a detailed granular breakdown of the unit costs within the users account. This way more intelligent decisions could be taken by the Upperware.

5.9 Solver-to-deployment

Overview

The `Solver-to-deployer` component is a glue layer between the *Reasoner* and the *Adapter* (*cf.* Section 6). It participates to lowering the dependencies of solvers to the remaining of PAASAGE. As described in Section 3, Upperware meta-models aim at enabling interactions between the *Profiler* and the *Reasoner* while lowering dependencies to CAMEL. Solvers produce solutions using these Upperware meta-models. The main objective of the `Solver-to-deployment` component is to translate the output of the Solvers into the CAMEL deployment model (aka CPSM). Figure 31 describes the various subcomponents that compose the *Solver-to-deployer* component.

The `Model Processor` component loads the models received as input and passes them to the `Derivator` component, that encapsulates the functionality to parse and extract the required elements. The `Derivator` component is concerned with matching the CP and PaaSage Application Models of applications so as to extend the CAMEL model with solutions. Finally, the `CAMEL Generator` component save the enhanced CAMEL model.

Matching Algorithm

The `Solver-to-deployement` component is implemented in Java. It receives the solution as a list of objects with the `PaaSageVariable` type. A `PaaSageVariable` object is described as follows:

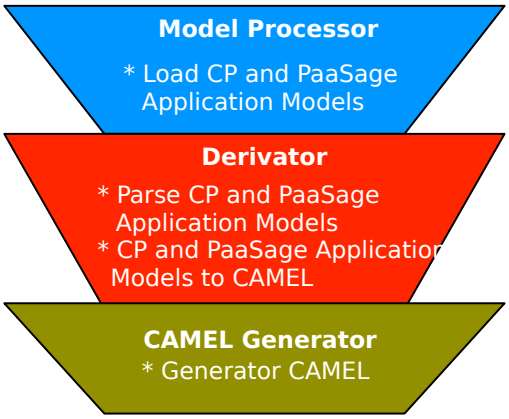

Figure 31: Solver-to-deployer-overview.

RelatedComponent It gives the ApplicationComponent of the UpperModel to instanciate in the CAMEL model.

RelatedVirtualMachineProfile It enables determining the VM template to instanciate in the CAMEL model.

RelatedProvider It gives the Provider from the UpperwareModel that allows finding in which provider the VM must be instanciated.

The creation process comprises the following steps.

1. *Creation of the InternalComponentInstances.*

   A PaaSageVariable's relatedComponent is used to create one corresponding InternalComponentInstance. To do so, we need to find in the CAMEL model the associated InternalComponent. This InternalComponent is associated to a list of ProvidedCommunications and a list of requiredCommunications. For each item of these lists, we must create a corresponding instance, either ProvidedCommunicationInstance or RequiredCommunicationInstance (depending on the original type). The two resulting lists can then be associated to the InternalComponentInstance.

2. *Creation of the VmInstance.*

   PaaSageVariable's RelatedVirtualMachineProfile and RelatedProvider are used to create a vmInstance. Those two values allow finding the VM and the ProviderModel. From the VM, we create a ProvidedHostInstances that gets associated to the VmInstance. The ProviderModel is used to find the VMType and VMTypeValue of the VmInstance.

3. *Creation of the HostingInstances.*

   The HostingInstances are created using the previously-created VmInstance and InternalComponentInstances, as well as the InternalComponent associated to the ComponentInstance.

   A HostingInstance must be created for each ProvidedHostInstance associated to the VmInstance. Each HostingInstance must be associated to the current ProvidedHostInstance and to the RequiredHostInstance matching the InternalComponentInstance.

6 Adapter

The purpose of the *Adapter* is to transform the currently running application configuration into the target configuration in an efficient and safe way. The *Adapter* is composed of three components, the `Adaptation Manager`, the `Plan Generator`, and the `Application Controller`, as shown in Figure 32. These three components are discussed in the following three sections. The last two sections discuss the `SRL Adapter`, a component that helps with configuring the Executionware, and the `Executionware client`, a library for using the REST interface of the Executionware.

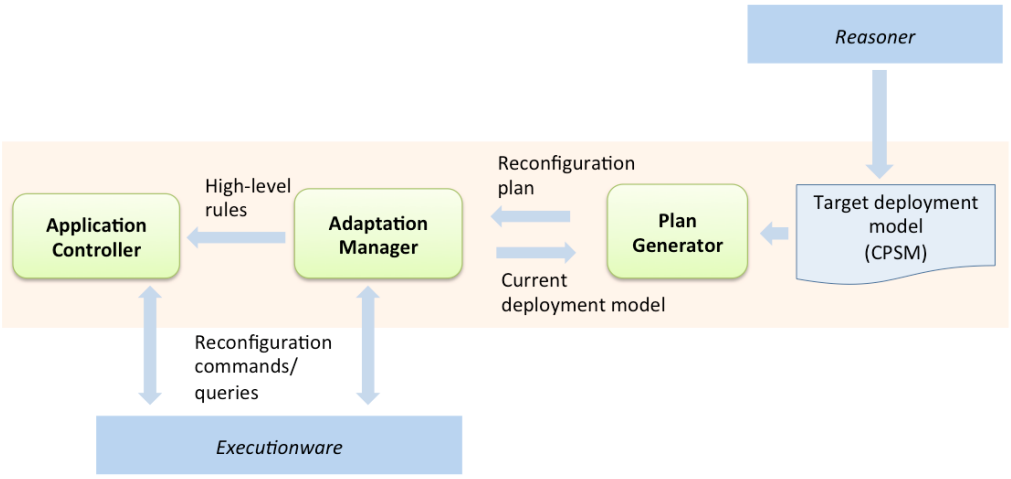

Figure 32: Adapter Architecture.

6.1 Adaptation Manager

Overview

`Adaptation Manager` drives the overall reconfiguration process. It has three main responsibilities: (1) validating reconfiguration plans, (2) applying the plans to the running system in an efficient and safe way, and (3) maintaining an up-to-date representation of the current system state. It communicates with the *Reasoner* to obtain target deployment models, with the `Plan Generator` to obtain plans, with the `Application Controller` to configure high-level monitoring rules, and with the *Executionware* to collect information and to execute reconfiguration actions.

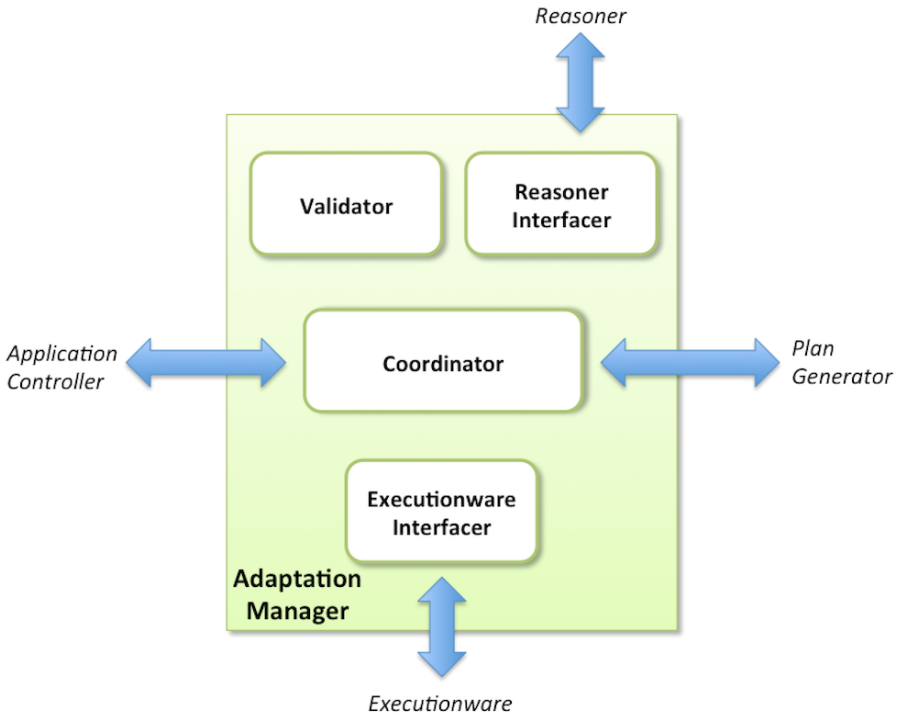

Figure 33: Adaptation Manager structure.

Implementation

The Adaptation Manager is decomposed into four main components shown in Figure 33. The `Reasoner Interfacer` loads the deployment model delivered by the *Reasoner*. The `ExecutionWare Interfacer` provides a wrapper interface to the REST API used for *Upperware-Executionware* interactions (see the next section). The `Validator` decides whether a reconfiguration plan is acceptable. Finally, the `Coordinator` directs the other components and maintains a deployment model that reflects the state of the current system.

The following workflow illustrates the reconfiguration process.

1. The `Reasoner Interfacer` obtains a new target deployment model from the *Reasoner* (*cf.* Section 5.9).

2. The `Plan Generator` receives the target model along with the current model and produces a reconfiguration plan.

3. The `Validator` verifies that the plan execution will be beneficial to the system.

4. The Coordinator configures the Application Controller to supervise application execution.

5. The Coordinator invokes the ExecutionWare Interfacer to deploy the *Executionware* specific actions generated from the plan.

6. The Coordinator updates the current deployment model.

At Step 3, if validation fails, the Coordinator restarts the process with a new target deployment model, when available. At Step 5, if one of the actions fails, the Coordinator terminates plan execution, updates the deployment model, and asks the Plan Generator for a new reconfiguration plan starting from the updated state.

The Adaptation Manager is implemented as a Java process that uses the Apache HTTPClient library for invoking REST operations and the ZeroMQ library for exchanging messages with other PaaSaage components. The remaining of this section provides more details on the Coordinator, Validator and the ExecutionWare Interfacer.

Coordinator

As previously mentioned, the Coordinator directs the execution of the other components within the *Adapter*. Once a new target deployment model is received, it invokes the Plan Generator to obtain a reconfiguration plan, and passes this plan through the Validator to verify its applicability. It can then proceed to execute the reconfiguration plan. Notably, the plan is executed in parallel, reducing the overall reconfiguration duration. The plan execution process is described next.

The target reconfiguration plan contains a set of tasks that have dependencies among themselves. Examples of such tasks are creation/update/deletion of VM, VM Instance, Internal Component, Communication, Hosting etc. The first step of executing the plan consists in casting the task dependencies within the plan into a graph data structure, where each node represents a task. As a result, the execution and verification of the tasks becomes easier, requiring traversal between the nodes. The second step is to translate the plan from the initial CAMEL-based perspective to an *Executionware*-based perspective as a preparation for the deployment using the Executionware Interfacer. Specifically, each of the tasks is mapped to a set of concrete executable actions for deployment in the Executionware. For example, the CommunicationType in the CAMEL model contains the following information:

- name

- provider port number
- consumer port number
- provider component
- consumer component

To realize this in the *Executionware*, three entities need to be created, *i.e.,* two port entities (provider and consumer) linked to a communication entity. The two port entities are independent and hence their creation can be parallelized. The provider and consumer components (also as entities) should already exist in the *Executionware* and be provided during the creation of the communication entity. The execution hierarchy can be clearly noticed while creating the communication type of the model. Nodes representing *communication type* creation task in the reconfiguration plan graph discussed above are replaced with following actions: creation action (two ports and communication entities), verifying existence and then linking the existing provider and consumer component entities. Similarly, every node (*i.e.,* task) in the reconfiguration plan graph is substituted by relevant actions to be performed in the *Executionware*. Thus the transformation into reconfiguration action graph is performed which is ready to be deployed.

The reconfiguration action graph may or may not contain dependent actions. As in the above example, the actions for creating the two port entities are independent. The Coordinator component traverses this graph and allocates each node to a pool of processors, *i.e.,* enqueues each of these nodes as threads onto a processor. The graph has design provisions such that a processor would wait for executing the actions within a node, in case of dependency with other neighbouring nodes within the graph. In this way the hierarchical execution is maintained in case of dependency while on the other hand deployment is parallelized as much as possible.

Validator

The `Validator` decides whether the proposed reconfiguration plan is acceptable. The decision relies on weighing potential reconfiguration benefits, such as increases in the application performance, against potential costs, such as disruption of the application operation. To take this decision, the validator uses information about the operational state of the application, its execution context as well as historical information stored in the metadata database.

The current implementation performs a cost-benefit calculation based on the utility values associated with the current deployment and the target deployment

as well as the estimated reconfiguration duration, derived from past executions of reconfiguration actions. Improved approaches are currently being investigated, taking into account the possibility of failures of reconfiguration actions.

Executionware Interfacer

The `Executionware Interfacer` supports the interaction with the *Executionware*, which relies on a REST API. The API is based on the main concepts of CAMEL, namely, applications, components, component instances, relationships, and virtual machines. It exposes operations for obtaining information about the running application (*e.g.* retrieving the state of a particular component instance) and sending reconfiguration commands (*e.g.* deploying an application or adding a new instance). Resource representations are in JSON. To deploy an application, the API client makes a series of HTTP POST requests. Successful deployment produces a set of linked resources that enables the client to monitor and manipulate the application through HTTP requests. Table 3 provides an overview of the API.

However, as described in section 6.1, the CAMEL concepts are realized by creating multiple entities in *Executionware*; reiterating its example, the CAMEL CommunicationType is actualized by creating three entities in the *Executionware*. And the *Executionware* provides unique id for every entity. The id is required to perform future actions like update or delete on the corresponding entity. Hence the `Executionware Interfacer` has a subcomponent `CamelExecwareMapper` to store these ids. This subcomponent is a data structure that represents the mapping between the CAMEL concepts and its represented entities in *Executionware* and stores information about the entities created during runtime for further actions.

6.2 Plan Generator

Overview

The `Plan Generator` is a sub-component of the `Adapter`. Its key role is to generate re/configuration plans which the `Adapter` uses to orchestrate the deployment/reconfiguration of a cloud application managed by the Paasage platform. In the simple deployment usage scenario, the `Plan Generator` takes in a single CAMEL DeploymentModel object which describes the target deployment desired by the *Reasoner*. In the reconfiguration usage scenario, it additionally takes in a current DeploymentModel object which describes the existing system deployment when the comparison is launched. For both usage scenarios, the `Plan Generator` generates a plan which contains a set of configuration tasks and the dependencies that exist between the tasks. In year 3, the design

Table 3: REST API.

Operations	**Description**
GET /api/application POST /api/application GET /api/application/{application_id} DELETE /api/application/{application_id}	Query, deploy, undeploy a single application within the system
GET /api/lifecycleComponent POST /api/lifecycleComponent GET /api/lifecycleComponent/{lifecycleComponent_id} DELETE /api/lifecycleComponent/{lifecycleComponent_id} PUT /api/lifecycleComponent/{lifecycleComponent_id}	Query, deploy, undeploy, perform actions on components using lifecycle shell scripts to execute the lifecycle actions (*e.g.* start, stop instances)
POST /api/communication DELETE /api/communication/{communication_id} GET /api/communication/{communication_id}	Create, destroy, query communication between two ports. The communication has a direction going from provided port to required port.
GET /api/instance/{instance_id} POST /api/instance DELETE /api/instance/{instance_id} GET /api/instance/{instance_id}	Query, add, delete, query, perform actions on an instance of an application component on a virtual machine
GET /api/virtualMachine	Lists all virtual machines available in the system

of the `Plan Generator` was re-iterated and the key changes are described in the next section.

Implementation

Figure 34 shows the class diagram for the Plan Generator. There are two key variations from the year 2 design:

1. The input models are now based on Paasage CAMEL.
2. The output plan contains a list of ConfigurationTasks as opposed to Action objects.

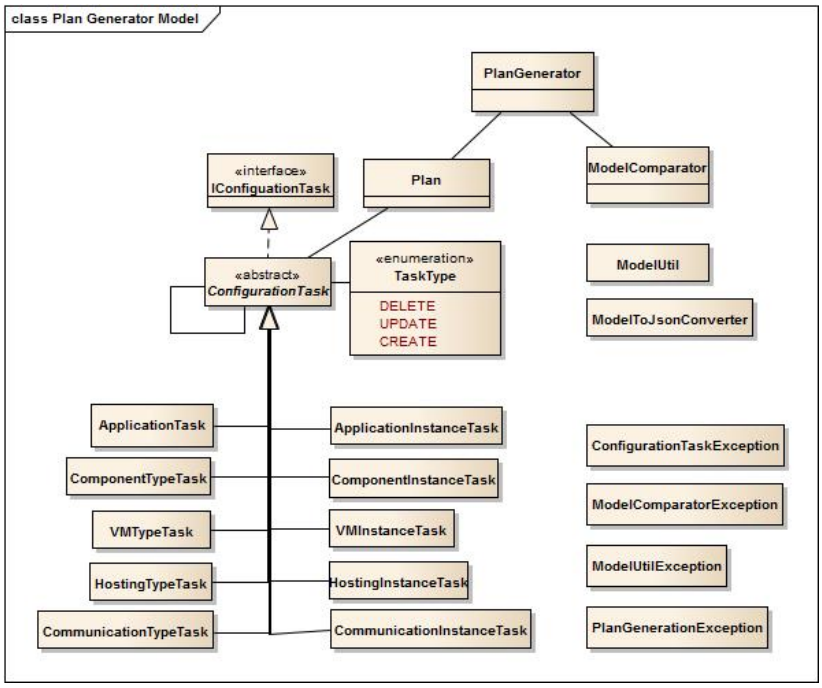

Figure 34: Plan Generator class diagram.

The adoption of CAMEL to cover the necessary aspects of modelling and provisioning of cloud application is intended to decouple the Plan Generator from the concrete enactment of the deployment by the `Adapter` using specific `ExecutionWare` API. As a result, the `Adapter` needs to map the ConfigurationTasks contained in the deployment Plan into specific deployment Actions for submission to the `ExecutionWare` (see Section 6.1 for a description of the process). To facilitate the mapping and orchestration, the *Plan Generator* provides each ConfigurationTask with a plain text Json key-value map of information describing the target configuration (see Appendix C) and computes the dependencies between tasks according to the logical dependencies between the CAMEL concepts (see Figure 35). Other than these changes, the overall

model comparison process remains the same except for the addition of functions to identify update to key attributes of an already deployed object.

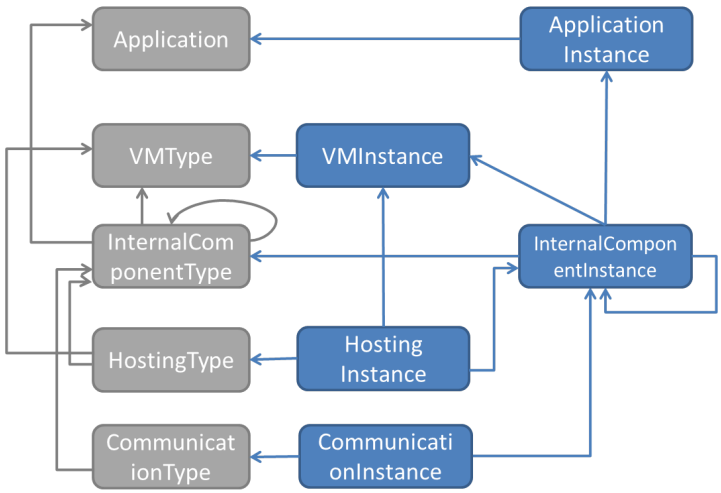

Figure 35: Logical dependencies between the configuration tasks.

During a model comparison, the *Plan Generator* first compares the type definition of each object instance. For example, the VMType objects are first compared followed by the VMInstance objects. If a target object is not found in the current model, a Create ConfigurationTask is generated. Conversely, if a current object is not found in the target model, a Delete ConfigurationTask is generated. And if an object is found in both the current and target models, the object's key attributes, e.g. port number, OS/Image, location, etc., are then compared to identify if an Update ConfigurationTask is needed to re-configure the deployed object.

6.3 Application Controller

Overview

The `Application Controller` component monitors the running application and its execution context in order to detect changes that make the current deployment unacceptable. Monitored information includes parameters necessary for evaluating application requirements (e.g., application response time), the status of previously-triggered reconfiguration actions (e.g., component deployment), and any deployment changes outside the Upperware control (e.g., VM failures, Executionware-triggered auto-scaling, or administrator actions). Based on this information, the component decides whether to trigger the execution of

the full reconfiguration process, based on the best available target deployment model.

Implementation

The `Application Controller` receives as input the CAMEL model, including application requirements, and uses Executionware APIs to configure metric collection and to receive notifications related to the current deployment. If the incoming information invalidates the current application model, then the application controller informs the `Application Manager`, which executes the full reconfiguration process involving generating, validating, and applying a new reconfiguration plan. This happens, for example, when a hard requirement is violated, such as a service-level objective specifying a maximum response time. To handle delays and transient failures, which are unavoidable in cloud infrastructures, the `Application Controller` applies simple fault-tolerance tactics, adding resiliency and avoiding costly reconfiguration. For example, it may retry reconfiguration actions or issue extra reconfiguration requests and cancel the late ones. If such tactics fail to address the situation, the component informs the `Application Manager` to launch a new reconfiguration process.

6.4 SRL Adapter

Overview

The `SRL Adapter` has two main functionalities: *(i)* the creation of the MetricInstances based on the MetricContexts and the actual available VirtualMachine resources and *(ii)* the translation of these metric definitions to the Executionware.

The workflow as shown in Figure 36: *(i)* The Adaptation Manager transfers the application into the Executionware and updates the model with the actual values; *(ii)* After the deployment has finished and deployment details like IP adresses are stored in the CAMEL model, the `SRL Adapter` is called; *(iii)* The Instantiator of the `SRL Adapter` connects to the CDO server and creates the required MetricInstances; *(iv)* Already instantiated monitors in the Executionware, that might be raw or composed Metrics from an older CAMEL model, are deleted; *(v)* For each entity in the Scalability and Metric models of the CAMEL model, an corresponding Adapter is instantiated and transforms the CAMEL entity into an Executionware entity; *(vi)* The Adapters send the entities to the Executionware via the FrontendCommunicator.

This Adapter component is loosely coupled to the others, since it can be used independently from the deployment phase. First of all, because the engagement of the sensors require the virtual machines to be up and running. And secondly, since the monitoring requirements can change during execution, *e.g.* after the

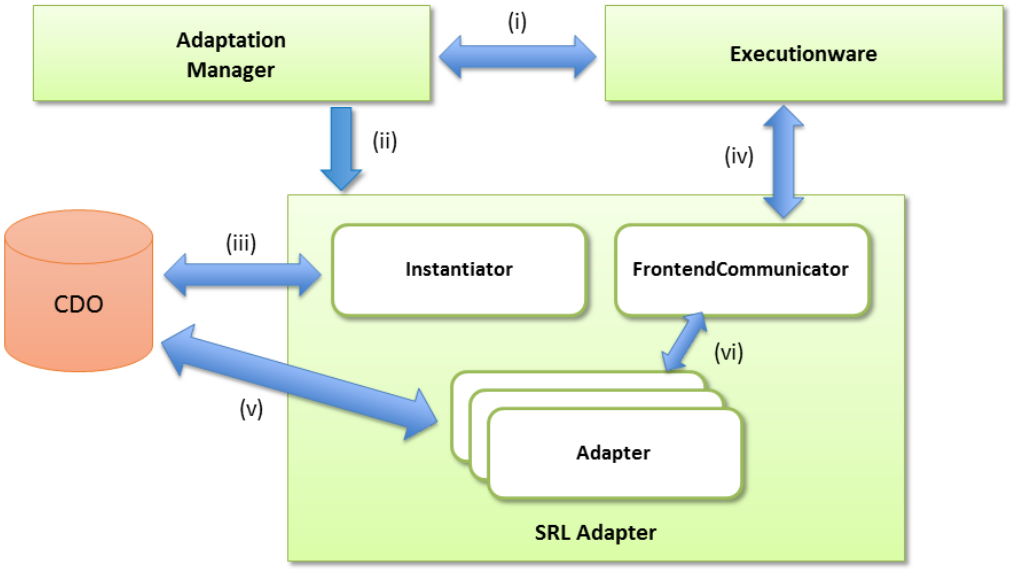

Figure 36: Workflow of the SRL Adapter.

Table 4: Structure of the SRL Adapter.

Package	**Description**
adapter	The actual adapters for each model entity.
communication	The communication manager towards the executionware.
config	Managing of the config parameters.
test	Holds some tests for the adapter and the executionware.
utils	Utilities for instantiating, finding and converting types, entities, etc. from the CAMEL model to the Executionware.

application manager identifies better-suited analysis methods for the monitoring based on workload behaviour.

Implementation

The Java package structure of SRL Adapter is shown in Table 4.

There is an Adapter implemented for each CAMEL entity of the Metrics and Scalability packages to translate it.

For this task it needs access to the CDO server in order to read and update the current CAMEL model. Therefore it needs the CDO endpoint and credentials as

configuration parameters, as well as the resource and model name. To be able to distinguish between several possible parallel deployments, an additional input parameter is the ExecutionContext of the deployment to be adapted.

In order to access the Executionware and therefore install probes and aggregators, also these credentials have to be passed as parameters. The monitoring agent *Visor* was forked to implement the connection to the *Metric Collector*. This fork is called "visor-bridge". Its endpoint is used to send the values of observed Monitors to the *metrics-collector-accessor* - a tool that has the sole task to instantiate the *Metric Collector* and therefore provide a ZeroMQ server to publish those measurements and events. Also the creation of MetricInstances can be deactivated. Example application arguments may look like this:

```
-cdoUser="sa"
-cdoPass=""
-modelName="BewanCamelModel"
-resourceName="enterprise-service-application.xmi_1442232824"
-executionContextName="ExecutionContext"
-colUser="john.doe@example.com"
-colTen="admin"
-colPass="admin"
-colUrl="http://localhost:9000/api"
-visEndpoint="localhost"
-createMetricInstances="true"
```

The implementation is solely done in Java and uses the REST interface via the `Executionware Client`.

6.5 Executionware Client

Overview

The `ExecutionwareClient` (or `Colosseum Client`[19]) is a Java-based client for the REST interface of the Executionware (Colosseum).

The `colosseum-client-test` application shows the intended usage with test deployments, like the be.wan use case.

Implementation

The client is instantiated with credentials and the Executionware endpoint. A Client Controller per entity of the Executionware model provides the functions for resource manipulation as predetermined by the REST priniciple of HTTP.

[19] `https://github.com/cloudiator/colosseum-client`

Table 5: Methods of the generic Client Controller of type T.

Method	**Meaning (and HTTP method)**
T get(long id)	Retrieves the object with the identifier *id*. (GET)
List<T> getList()	Retrieves all objects of the type of the specific Client Controller. (GET)
T create(T t)	Create the object *t* in Colosseum. (POST)
T update(obj)	Updates the object *t* with the current values in Colosseum. (PUT)
T delete(obj)	Deleting the object *t*. (DELETE)
T getSingle(Predicate filter)	Special form of the *get* method with a Predicate to be used as filter. (GET)
List<T> getList(Predicate filter)	Special form of the *list* method with a Predicate to be used as filter. (GET)

Listing 6: Example code for client usage.

```
// This example shows the manipulation of
// resources stored in Colosseum.

// example credentials and endpoint
String url = "http://localhost:9000/api";
String username = "john.doe@example.com";
String password = "admin";
String tenant = "admin";

// the builder to create a client
ClientBuilder clientBuilder = ClientBuilder.getNew()
    // the base url
    .url(url)
    // the login credentials
    .credentials(username, tenant, password);

Client client = clientBuilder.build();

//get the controller for the cloud entity
final ClientController<Cloud> controller = client.controller(Cloud.
    class);

//get the controller for the api entity
final ClientController<Api> apiController = client.controller(Api.
    class);

//create a new API
Api api = apiController.create(new Api("ApiName-" + random.nextInt
    (10000),
    "InternalProviderName-" + random.nextInt(100)));

//create a new Cloud
controller
    .create(new Cloud("MyCloud-" + random.nextInt(10000), "
        endpointTest.com", api.getId()));
```

```
//fetch all clouds
List<Cloud> clouds = controller.getList();

//fetch the first cloud of this list
Cloud cloud = clouds.get(0);

//create a another Cloud
cloud = controller
    .create(new Cloud("MyCloud-" + random.nextInt(1000), "endpointTest
        .com", api.getId()));

//update a cloud
cloud.setName("MyNewName-" + random.nextInt(100));
controller.update(cloud);

//delete a cloud
controller.delete(cloud);
```

The functions are presented in Table 5 and an example usage is shown in Listing 6.

7 Conclusion

The Upperware is made of three major components: the Profiler, the Reasoner, and the Adapter. Interactions with other PAASAGE elements (CAMEL, MD-DB/CDO, Executionware) have been well identified and the proposed Upperware product architecture and aim to minimize them to few elements.

The Profiler is mainly made of two sub-components, with distinct objectives. The `CP Generator Model-to-Solver` produces a constraint problem description that is improved by the `Rule Processor` by removing redundancies and verifying the list of Cloud providers candidates.

The Reasoner currently supports several kinds of solvers, so as to propose various strategies to efficiently compute a deployment depending on the circumstances. Hence, we have defined an architecture where new solvers could be integrated at low cost. Current efforts are geared towards developing a Learning Automata based solver, making use of existing solvers (MILP Solver and CP Solver), developing some greedy heuristics as well as simulator based heuristics, and developing a Meta Solver for handling complex situations. Reasoner also contains

The Adapter is made of three major sub-components. The Plan Generator transforms the target deployment computed by the Reasoner into an ordered set of Configuration Tasks which the Adapter maps to deployment Actions. These deployment Actions are executed parallely, if possible. The Adaptation Manager is responsible for driving the (re)configuration process, and in particular interacting with the Reasoner and the Executionware. The Application Controller component monitors the running application and its execution context in order to detect changes that make the current deployment unacceptable. Auxiliary functionnalities are handled in well-separated components to ease PaaSage evolution: the SRL Adapter for helping with configuring the Executionware and the Executionware client for communicating with the Executionware.

This document describes our views of the Upperware product at M36. When gaining experience by supporting more and more use cases, the Upperware architecture may evolve. Therefore, interactions with other PAASAGE partners (in particular from other work-package) may have some impacts on well identified sub-components. Interactions with the use case partners in PAASAGE will provide feedbacks on the actual usage of the Reasoner, and its relevance to fulfil requirements.

All these interactions and evaluations will contribute to the evolution of the Upperware.

References

[Bsi+13] Amin Bsila et al. *Deliverable D3.1.1 - Upperware Prototype*. PaaSage project deliverable. PaaSage Project, 2013.

[Jef+13] Keith Jeffery, Geir Horn, Lutz Schubert, Philippe Massonet, Kostas Magoutis, Brian Matthews, Tom Kirkham, Christian Perez and Alessandro Rossini. *Deliverable D1.6.1 - Initial Architecture Design*. PaaSage project deliverable. PaaSage Project, 2013.

[RP15] Alessandro Rossini and the PaaSage consortium. *D2.1.3 – CAMEL Documentation*. PaaSage project deliverable. Oct. 2015.

[Hop+15] Dennis Hoppe et al. *Deliverable D5.2.1 - Product ExecutionWare*. PaaSage project deliverable. PaaSage Project, 2015.

[Ros+15] Alessandro Rossini, Kiriakos Kritikos, Nikolay Nikolov, Jörg Domaschka, Frank Griesinger and Daniel Romero. *CAMEL Documentation v2015.06t*. 2015.

[QRD13] Clément Quinton, Daniel Romero and Laurence Duchien. "Cardinality-Based Feature Models With Constraints: A Pragmatic Approach". In: *SPLC - 17th International Software Product Line Conference - 2013*. Tokyo, Japan, Aug. 2013, pp. 162–166.

[LH13] Livia Predoiu and Heiner Stuckenschmidt. "Probabilistic Models for the Semantic Web: A Survey". In: *Web Technologies: Concepts, Methodologies, Tools, and Applications*. Ed. by Arthur Tatnall. Chapter 102. IGI Global, 5th July 2013, pp. 1896–1928. ISBN: 9781605669823. URL: doi:10.4018/978-1-60566-982-3.

[Ben+08] Ben Goertzel, Matthew Iklé, Izabela Freire Goertzel and Ari Heljakka. *Probabilistic Logic Networks: A Comprehensive Framework for Uncertain Inference*. DOI: 10.1007/978-0-387-76872-4. Springer, 2008. 336 pp. ISBN: 978-0-387-76872-4. URL: http://www.springer.com/computer/ai/book/978-0-387-76871-7.

[Pei13] Pei Wang. *Non-Axiomatic Logic: A Model of Intelligent Reasoning*. World Scientific, July 2013. 276 pp. ISBN: 978-981-4440-29-5. URL: http://www.worldscientific.com/worldscibooks/10.1142/8665.

[CC05] Costas P. Pappis and Constantinos I. Siettos. “Fuzzy Reasoning”. In: *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Ed. by Edmund K. Burke and Graham Kendall. Chapter 15. Springer, 2005, pp. 437–474. ISBN: 978-0-387-23460-1, 978-0-387-28356-2. URL: http://link.springer.com/chapter/10.1007/0-387-28356-0_15 (visited on 05/07/2013).

[DY08] David G. Luenberger and Yinyu Ye. *Linear and Nonlinear Programming*. 3rd. Springer, 2008. 546 pp. ISBN: 978-0387745022.

[Lau08] Laurence A. Wolsey. “Mixed Integer Programming”. In: *Wiley Encyclopedia of Computer Science and Engineering*. DOI: 10.1002/9780470050118.ecse244. John Wiley & Sons, Inc., 2008, pp. 1–10. ISBN: 9780470050118. URL: http://onlinelibrary.wiley.com/doi/10.1002/9780470050118.ecse244/abstract (visited on 05/07/2013).

[BJ08] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. 4th. Vol. 21. Algorithms and Combinatorics. Berlin Heidelberg: Springer, 2008. 627 pp. ISBN: 978-3-540-71843-7. URL: http://www.springer.com/new+&+forthcoming+titles+(default)/book/978-3-540-71843-7.

[PR09] Pascal Van Hentenryck and Russell Bent. *Online Stochastic Combinatorial Optimization*. The MIT Press, 2009. ISBN: 0262513471, 9780262513470.

[ALA09] Aharon Ben-Tal, Laurent El Ghaoui and Arkadii Semenovich Nemirovskii. *Robust optimization*. Princeton Series in Applied Mathematics. Princeton: Princeton University Press, 2009. ISBN: 9781400831050 1400831059 9780691143682 0691143684. URL: http://public.eblib.com/EBLPublic/PublicView.do?ptiID=457706 (visited on 07/07/2013).

[DM03] Dimitris Bertsimas and Melvyn Sim. “Robust discrete optimization and network flows”. In: *Mathematical Programming* 98.1 (Sept. 2003), pp. 49–71. ISSN: 0025-5610, 1436-4646. DOI: 10.1007/s10107-003-0396-4. URL: http://link.springer.com/article/10.1007/s10107-003-0396-4 (visited on 07/07/2013).

[Jam03] James C. Spall. *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. Wiley, Apr. 2003. 618 pp. ISBN: 978-0-471-33052-3. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471330523.html.

[RA98] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning*. Vol. 9. Boston, MA, USA: MIT Press, 1998. ISBN: 0-262-19398-1.

[KM89] Kumpati S. Narendra and Mandayam A. L. Thathachar. *Learning Automata: An Introduction*. Prentice Hall, May 1989. ISBN: 0134855582.

[AK97] Alexander Semenovich Poznyak and Kaddour Najim. *Learning Automata and Stochastic Optimization*. Vol. 225. Lecture Notes in Control and Information Sciences. DOI: 10.1007/BFb0015102. Springer Berlin Heidelberg, 1997. ISBN: 978-3-540-76154-9, 978-3-540-40938-0. URL: http://link.springer.com/book/10.1007/BFb0015102/page/1.

[Nor83] Norio Baba. "On the Learning Behaviors of Variable-Structure Stochastic Automaton in the General N-Teacher Environment". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.3 (Mar. 1983), pp. 224–231.

[MP04] Mandayam A. L. Thathachar and P. S. Sastry. *Networks of Learning Automata: Techniques for Online Stochastic Optimization*. 1st ed. Boston, MA, USA: Kluwer Academic, 2004. ISBN: 1-4020-7691-6.

[GB10] Geir Horn and B. John Oommen. "Solving Multiconstraint Assignment Problems Using Learning Automata". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 40.1 (Feb. 2010), pp. 6–18. ISSN: 1083-4419. DOI: 10.1109/TSMCB.2009.2032528.

[CPR73] Carl Hewitt, Peter Bishop and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. Conference location: San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804 (visited on 25/03/2014).

[Mik61] Mikhail L'vovich Tsetlin. "On the Behavior of Finite Automata in Random Media". In: *Automation and Remote Control* 22.10 (1961), pp. 1210–1219. ISSN: 1608-3032.

[VI63] V. I. Varshavskii and I. P. Vorontsova. “On the Behaviour of Stochastic Automata with a Variable Structure”. In: *Automation and remote control* 24 (Mar. 1963), pp. 327–333.

[GK66] George James McMurtry and K. S. Fu. “A variable structure automaton used as a multimodal searching technique”. In: *IEEE Transactions on Automatic Control* AC-11.3 (July 1966), pp. 379–387. ISSN: 0018-9286. DOI: 10.1109/TAC.1966.1098374.

[BD68] B. Chandrasekaran and David W. C. Shen. “On Expediency and Convergence in Variable-Structure Automata”. In: *IEEE Transactions on Systems Science and Cybernetics* SSC-4.1 (Mar. 1968), pp. 52–60. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300188.

[RK71] R. Viswanathan and Kumpati S. Narendra. *Application of stochastic automata models to learning systems with multimodal performance criteria*. Technical Report CT-40. No copies are available at Yale, but a copy can be obtained from the author of this work. New Haven, Connecticut, USA: Becton Center, Yale University, June 1971.

[Mal+13] Maciej Malawski, Bartosz Baliś, Dariusz Król and Achilleas Achilleos. *Deliverable D6.1.3 - Initial Requirements*. 2013.

[Ste14] Mike Steglich. *CMPL (Coin Mathematical Programming Language): https://projects.coin-or.org/Cmpl*. 2014.

[KP15] Kyriakos Kritikos and Dimitris Plexousakis. “Multi-Cloud Application Design through Cloud Service Composition”. In: *CLOUD*. New York City, NY, USA, 2015, pp. 686–693.

[HY81] C. Hwang and K. Yoon. “Multiple Criteria Decision Making”. In: *Lect. Notes Econ. Math.* (1981).

[Bar+13] George Baryannis, Panagiotis Garefalakis, Kyriakos Kritikos, Kostas Magoutis, Antonis Papaioannou, Dimitris Plexousakis and Chrysostomos Zeginis. “Lifecycle management of service-based applications on multi-clouds: a research roadmap.” In: *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds (MultiCloud ’13)*. Prague, Czech Republic, 2013, pp. 13–20.

[Kri+14] Kyriakos Kritikos et al. *D4.1.1 – Prototype Metadata Database and Social Network / Prototype of Metadata Integration Extension*. PaaSage project deliverable. Apr. 2014.

[Cha14] Drona Pratap Chandu. “A Parallel Genetic Algorithm for Three Dimensional Bin Packing with Heterogeneous Bins”. In: *CoRR* abs/1411.4565 (2014). URL: http://arxiv.org/abs/1411.4565.

[LKK99] W. Leinberger, G. Karypis and V. Kumar. “Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints”. In: *Proc of International Conference on Parallel Processing*. Sept. 1999, pp. 404–412. DOI: 10.1109/ICPP.1999.797428.

[GZ13] Michaël Gabay and Sofia Zaourar. *Variable size vector bin packing heuristics - Application to the machine reassignment problem*. Tech. rep. INRIA, Sept. 2013.

[Jea+13] E. Jeannot, E. Meneses, G. Mercier, F. Tessier and Gengbin Zheng. *Process Placement in Multicore Clusters: Algorithmic Issues and Practival Techiniques*. Tech. rep. INRIA, 2013.

[Kum+11] Karthik Kumar, Jing Feng, Yamini Nimmagadda and Yung-Hsiang Lu. “Resource Allocation for Real-Time Tasks Using Cloud Computing”. In: *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)* (July 2011), pp. 1–7. DOI: 10.1109/ICCCN.2011.6006077. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6006077.

[MNC11] Ching Chuen Teck Mark, Dusit Niyato and Tham Chen-Khong. “Evolutionary Optimal Virtual Machine Placement and Demand Forecaster for Cloud Computing”. In: *2011 IEEE International Conference on Advanced Information Networking and Applications* (Mar. 2011), pp. 348–355. DOI: 10.1109/AINA.2011.50. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5763426.

[Pan+11] R. Panigrahy, K. Talwar, L. Uyeda and U. Wieder. *Heuristics for Vector Bin Packing*. Tech. rep. Microsoft Research, 2011.

[Sha+11] Upendra Sharma, Prashant Shenoy, Sambit Sahu and Anees Shaikh. “Kingfisher: Cost-aware elasticity in the cloud”. In: *2011 Proceedings IEEE INFOCOM* (Apr. 2011), pp. 206–210. DOI: 10.1109/INFCOM.2011.5935016. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5935016.

[DR13] Frédéric Desprez and Jonathan Rouzaud-Cornabas. *SimGrid Cloud Broker: Simulating the Amazon AWS Cloud*. Anglais. Rapport de recherche RR-8380. INRIA, Nov. 2013, p. 30. URL: http://hal.inria.fr/hal-00909120.

[Ger+06] Gerald Tesauro, Nicholas K. Jong, Rajarshi Das and Mohamed N. Bennani. "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation". In: *Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC '06)*. IEEE International Conference on Autonomic Computing, 2006. ICAC '06. Ed. by Manish Parashar, Jeffrey O. Kephart, Omer Rana and Mazin Yousif. Conference location: Dublin, Ireland: IEEE Computer Society, 12th June 2006, pp. 65–73. DOI: 10.1109/ICAC.2006.1662383.

[Pet70] Peter C Fishburn. *Utility theory for decision making*. New York: Wiley, 1970. ISBN: 0471260606 9780471260608. URL: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=AD0708563 (visited on 30/03/2014).

[FJ04] Francis C. Chu and Joseph Y. Halpern. "Great expectations. Part II: generalized expected utility as a universal decision rule". In: *Artificial Intelligence* 159.1 (Nov. 2004), pp. 207–229. ISSN: 0004-3702. DOI: 10.1016/j.artint.2004.05.007. URL: http://www.sciencedirect.com/science/article/pii/S0004370204000979 (visited on 01/04/2014).

[I E68] I. E. Sutherland. "A Futures Market in Computer Time". In: *Communications of the ACM* 11.6 (June 1968), pp. 449–451. ISSN: 0001-0782. DOI: 10.1145/363347.363396. URL: http://doi.acm.org/10.1145/363347.363396 (visited on 30/03/2014).

[JD03] Jeffrey O. Kephart and David M. Chess. "The vision of autonomic computing". In: *Computer* 36.1 (2003), pp. 41–50. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1160055.

[Ter03] Terence Kelly. "Utility-Directed Allocation". In: *Proceedings of the First Workshop on Algorithms and Architectures for Self-Managing Systems*. San Diego, California, USA: ACM, 11th June 2003. URL: http://tesla.hpl.hp.com/self-manage03/.

[JR07] Jeffrey O. Kephart and Rajarshi Das. “Achieving Self-Management via Utility Functions”. In: *IEEE Internet Computing* 11.1 (2007), pp. 40–48. ISSN: 1089-7801. DOI: 10.1109/MIC.2007.2.

[Wil+04] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart and Rajarshi Das. “Utility functions in autonomic systems”. In: *Proceedings of the International Conference on Autonomic Computing*. IEEE, 17th May 2004, pp. 70–77. ISBN: 0-7695-2114-2. DOI: 10.1109/ICAC.2004.1301349.

[Kur+09] Kurt Geihs et al. “A comprehensive solution for application-level adaptation”. In: *Software: Practice and Experience* 39.4 (2009), pp. 385–422. ISSN: 1097-024X. DOI: 10.1002/spe.900. URL: http://onlinelibrary.wiley.com/doi/10.1002/spe.900/abstract (visited on 07/07/2013).

[Sve+12] Svein Hallsteinsen, Kurt Geihs, Nearchos Paspallis, Frank Eliassen, Geir Horn, Jorge Lorenzo, Alessandro Mamelli and George A. Papadopoulos. “A development framework and methodology for self-adapting applications in ubiquitous computing environments”. In: *Journal of Systems and Software* 85.12 (Dec. 2012), pp. 2840–2859. ISSN: 0164-1212. DOI: 10.1016/j.jss.2012.07.052. URL: http://www.sciencedirect.com/science/article/pii/S0164121212002245 (visited on 17/12/2012).

[Jac+06] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund and Eli Gjørven. “Using architecture models for runtime adaptability”. In: *IEEE Software* 23.2 (2006), pp. 62–70. ISSN: 0740-7459. DOI: 10.1109/MS.2006.61.

[Jac+13] Jacqueline Floch et al. “Playing MUSIC — building context-aware and self-adaptive mobile applications”. In: *Software: Practice and Experience* 43.3 (Mar. 2013), pp. 359–388. ISSN: 1097-024X. DOI: 10.1002/spe.2116. URL: http://onlinelibrary.wiley.com/doi/10.1002/spe.2116/abstract (visited on 30/03/2014).

[FA09] Franck Fleurey and Arnor Solberg. “A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems”. In: *Model Driven Engineering Languages and Systems: Proceedings of the 12 International conference (MODELS 2009)*. Ed. by Andy Schürr and Bran Selic. Vol. 5795. Lecture Notes in Computer Science. Conference location: Denver, Colorado, USA: Springer, 4th Oct. 2009, pp. 606–

621. ISBN: 978-3-642-04424-3, 978-3-642-04425-0. DOI: 10.1007/978-3-642-04425-0_47. URL: http://link.springer.com/chapter/10.1007/978-3-642-04425-0_47 (visited on 07/08/2013).

[SDB06] Shang-Wen Cheng, David Garlan and Bradley Schmerl. "Architecture-based Self-adaptation in the Presence of Multiple Objectives". In: *Proceedings of the 2006 International Workshop on Self-adaptation and Self-managing Systems (SEAMS'06)*. Ed. by Betty H. C. Cheng, Rogério de Lemos, Stephen Fickas, David Garlan, Jeff Magee, Hausi Müller and Richard Taylor. SEAMS '06. Conference Location: Shanghai, China: ACM, 20th May 2006, pp. 2–8. ISBN: 1-59593-403-0. DOI: 10.1145/1137677.1137679. URL: http://doi.acm.org/10.1145/1137677.1137679 (visited on 01/04/2014).

[GPD11] Giuseppe Valetto, Paul deGrandis and Dale Seybold Jr. "Synthesis of application-level utility functions for autonomic self-assessment". In: *Cluster Computing* 14.3 (Sept. 2011), pp. 275–293. ISSN: 1386-7857, 1573-7543. DOI: 10.1007/s10586-010-0130-y. URL: http://link.springer.com/article/10.1007/s10586-010-0130-y (visited on 30/03/2014).

[Ric58] Richard E. Bellman. "On a Routing Problem". In: *Quarterly of Applied Mathematics* 16 (1958), pp. 87–90.

[Mou+06] Mourad Alia, Geir Horn, Frank Eliassen, Mohammad Ullah Khan, Rolf Fricke and Roland Reichle. "A Component-Based Planning Framework for Adaptive Systems". In: *On the Move to Meaningful Internet Systems 2006: Proceedings of the OTM Confederated International Conferences CoopIS, DOA, GADA, and ODBASE*. Ed. by Robert Meersman and Zahir Tari. Vol. Part II. Lecture Notes in Computer Science. Montpellier, France: Springer Berlin Heidelberg, 29th Nov. 2006, pp. 1686–1704. ISBN: 978-3-540-48274-1, 978-3-540-48283-3. DOI: 10.1007/11914952_45. URL: http://link.springer.com/chapter/10.1007/11914952_45 (visited on 12/02/2014).

[JE02] James J. Buckley and Esfandiar Eslami. *An introduction to fuzzy logic and fuzzy sets*. Berlin Heidelberg: Physica-Verlag, 2002. 285 pp. ISBN: 3790814474 9783790814477. URL: http://www.springer.com/computer/ai/book/978-3-7908-1447-7?otherVersion=978-3-7908-1799-7.

A Common Meta-Models

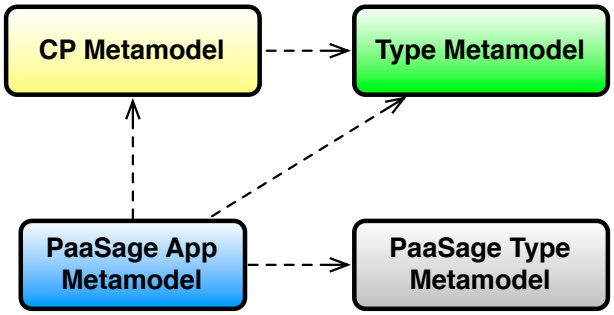

Figure 37: Meta-Models overview.

Figure 37 provides an overview of the Upperware meta-models and their relationships.

The *Constraint Problem Meta-Model* (CP Meta-Model) and *Types Meta-Model* enable the definition of the Cloud provider selection problem as a constraint problem. They are fully describes on Page 101.

The *PaaSage Application Meta-Model* (PaaSage App Meta-Model) and *PaaSage Type Meta-Model* establish the relationship between concepts from the Cloud and constraint problem worlds. They are fully described on Page 102.

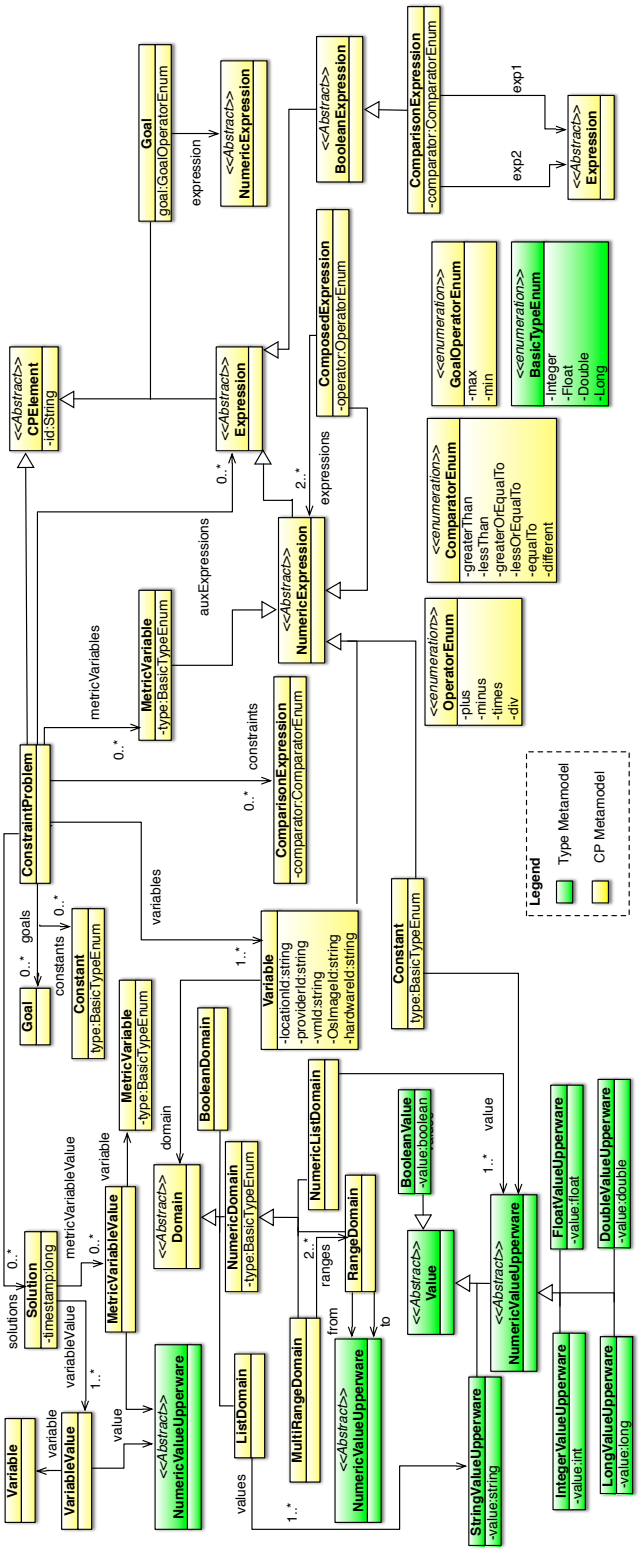

Figure 38: CP and Type Meta-Models.

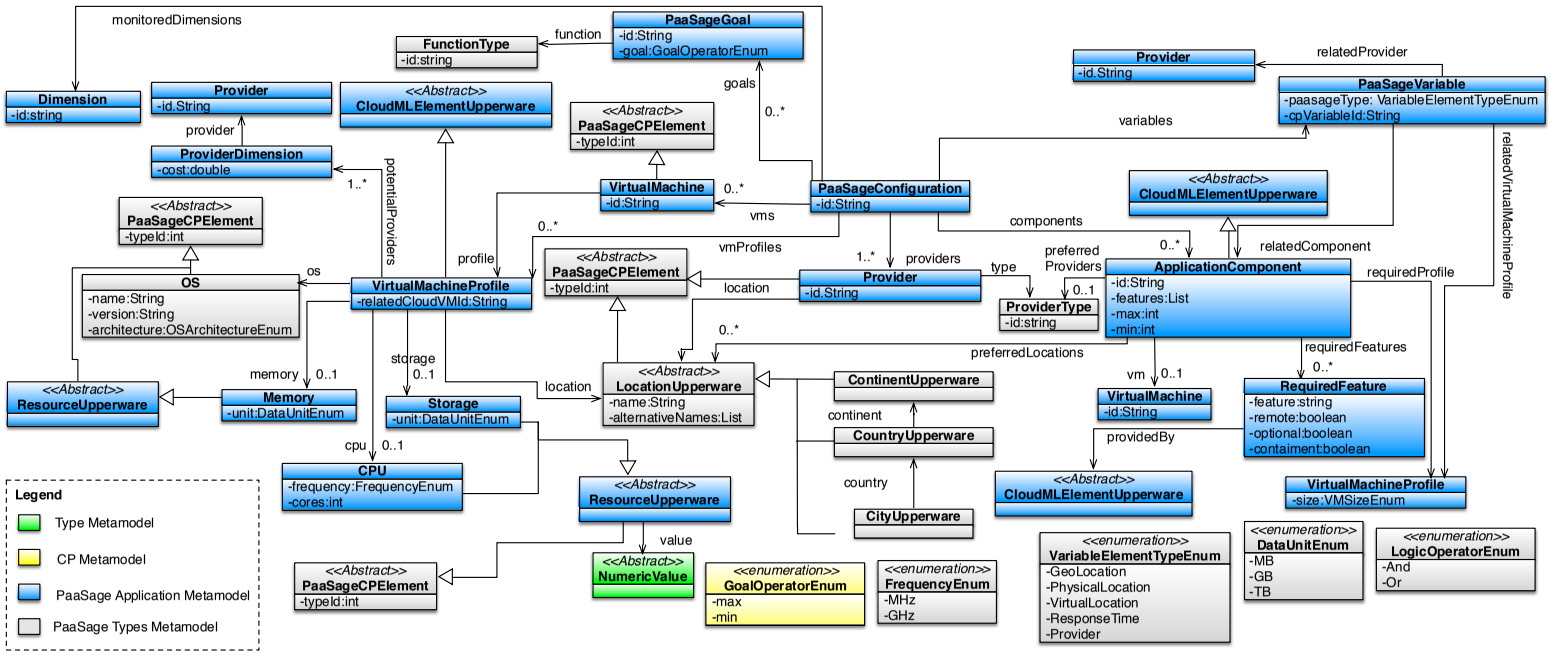

Figure 39: PaaSage Type and Application Meta-Model.

B Saloon Ontology

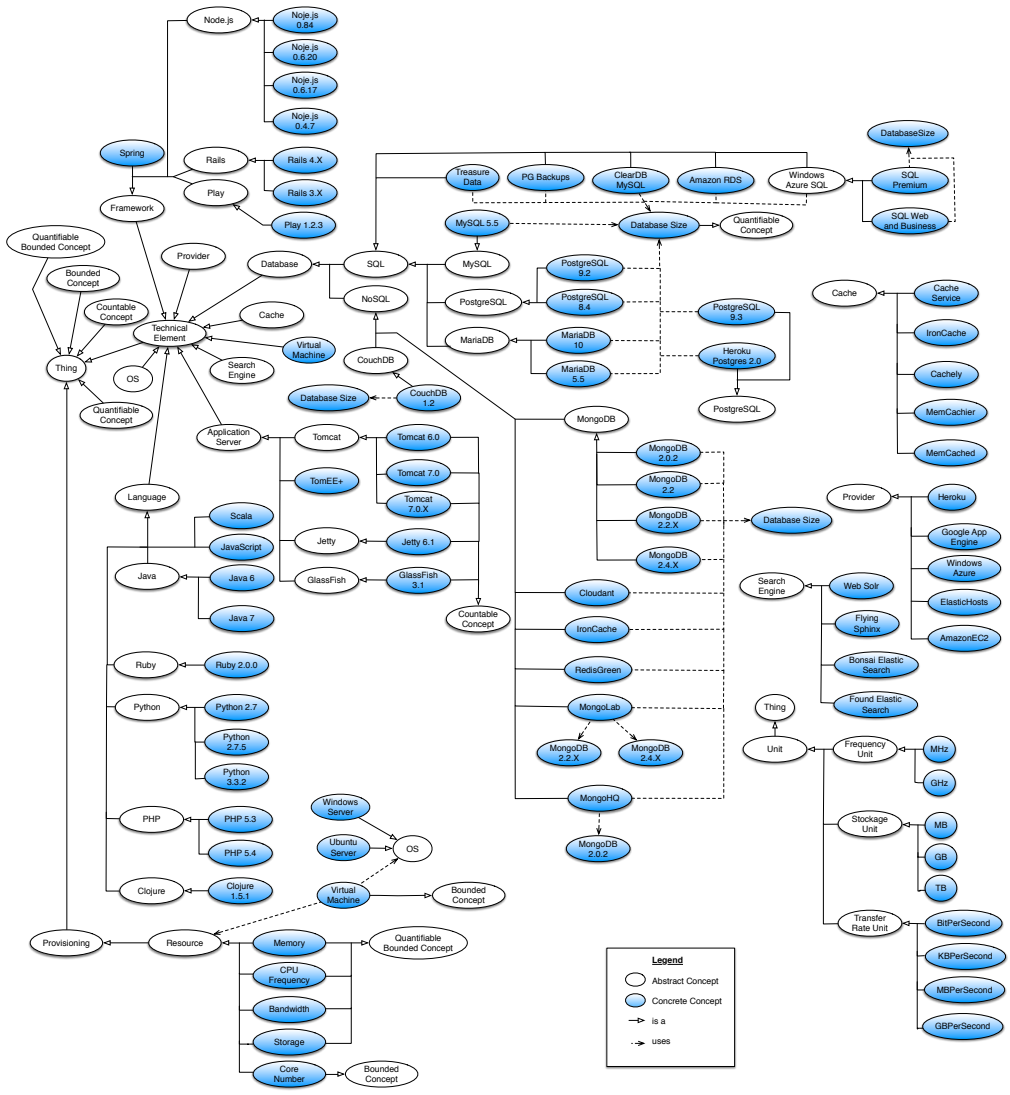

Figure 40: Saloon Ontology.

C Plan Generator Output Data Dictionary

Table 6: Data Dictionary

Object	Attribute	Type	Opt.	Description
ApplicationTask	name	String		Application name as in CAMEL Model, e.g. Scalarm
	objType	String		hardcoded to "application"
	description	String	yes	Description of the application.
	version	String		Application version.
	owner	String		Organisation that owns this application.
ApplicationInstance Task	name	String		Application name as in CAMEL Model appended with Instance, e.g. ScalarmInstance
	objType	String		hardcoded to "applicationInstance"
	type	String		Application type name, i.e. the Application name
VMInstanceTask	name	String		VM Instance name
	objType	String		hardcoded to "vmInstance"
	type	String		VM/Component type name
	cloud	String		Cloud provider name
	locations	String Array	yes	Cloud provider location.
	driver	String		Driver class for a particular cloud provider. Note that the value may be an empty String.
	endpoint	String		Endpoint for the cloud provider
	credential	Json object		This currently contains two key-value pairs : username and password Strings
	vmType	String		VM flavour name
	providedHostInstances	String Array		Array of provided host instance names
	providedCommunicationInstances	String Array	yes	Array of provided communication instance names
VMTypeTask	name	String		VM Type name
	objType	String		hardcoded to "VM"
	osImage	String	yes	OS image name. An object only contains either osImage or os. Not both.

Table 6: Data Dictionary

Object	Attribute	Type	Opt.	Description
	os	String	yes	OS name. An object only contains either osImage or os. Not both.
	os64bit	boolean	yes	True if IS is 64bit, else false. This is an attribute of os.
	configName	String	yes	(Configuration) name
	downloadCmd	String	yes	(Configuration) download command
	configureCmd	String	yes	(Configuration) configure command
	installCmd	String	yes	(Configuration) install command
	startCmd	String	yes	(Configuration) start command
	StopCmd	String	yes	(Configuration) stop command
	uploadCmd	String	yes	(Configuration) upload command
	providedHosts	String Array		Array of provided host names
	providedCommunications	String Array	yes	Array of provided communication names
ComponentInstanceTask	name	String		Internal component instance name
	objType	String		hardcoded to "internalComponentInstance"
	type	String		Internal component type name
	providedHostInstances	String Array	yes	Array of provided host instance names
	requiredHostInstance	String		Required communication instance name
	providedCommunicationInstances	String Array	yes	Array of provided communication instance names
	requiredCommunicationInstances	String Array		Array of required communication instance names
ComponentTypeTask	name	String		Internal component type name
	objType	String		hardcoded to "internalComponent"
	configName	String	yes	(Configuration) name
	downloadCmd	String	yes	(Configuration) download command
	configureCmd	String	yes	(Configuration) configure command

Table 6: Data Dictionary

Object	Attribute	Type	Opt.	Description
	installCmd	String	yes	(Configuration) install command
	startCmd	String	yes	(Configuration) start command
	StopCmd	String	yes	(Configuration) stop command
	uploadCmd	String	yes	(Configuration) upload command
	providedHosts	String Array	yes	Array of provided host names
	requiredHost	String		Required host name
	providedCommunications	String Array	yes	Array of provided host names
	requiredCommunications	String Array		Array of required communication names
CommunicationInstanceTask	name	String		Communication instance name
	objType	String		hardcoded to "communicationInstance"
	type	String		Communication type name
	providerInstance	String		Communication provider instance name
	providerCompInstanceTask	String	yes	ComponentInstanceTask name - owner of the providerInstance
	consumerInstance	String		Communication consumer instance name
	consumerCompInstanceTask	String	yes	ComponentInstanceTask name - owner of the consumerInstance
CommunicationTypeTask	name	String		Communication type name
	objType	String		hardcoded to "communication"
	communicationType	String		LOCAL, REMOTE, ANY
	provider	String		Provider name
	providerCompTypeTask	String	yes	ComponentTypeTask name - owner of the Provider
	providerPort	int		Provider port number
	providedPortconfigName	String	yes	(Configuration) name
	providedPortdownloadCmd	String	yes	(Configuration) download command

Table 6: Data Dictionary

Object	Attribute	Type	Opt.	Description
	providedPortconfigure-Cmd	String	yes	(Configuration) configure command
	providedPortinstallCmd	String	yes	(Configuration) install command
	providedPortstartCmd	String	yes	(Configuration) start command
	providedPortStopCmd	String	yes	(Configuration) stop command
	providedPortupload-Cmd	String	yes	(Configuration) upload command
	consumer	String		Consumer name
	consumerCompType-Task	String	yes	ComponntTypeTask name - owner of the Consumer
	isMandatory	boolean		True if consumer depends on this communication, else false
	consumerPort	int		Consumer port number
	requiredPortconfig-Name	String	yes	(Configuration) name
	requiredPortdownload-Cmd	String	yes	(Configuration) download command
	requiredPortconfigure-Cmd	String	yes	(Configuration) configure command
	requiredPortinstallCmd	String	yes	(Configuration) install command
	requiredPortstartCmd	String	yes	(Configuration) start command
	requiredPortStopCmd	String	yes	(Configuration) stop command
	requiredPortupload-Cmd	String	yes	(Configuration) upload command
HostingInstanceTask	name	String		Hosting instance name
	objType	String		hardcoded to "hostingInstance"
	type	String		Hosting type name
	providerInstance	String		Hosting provider instance name. The provider could be of type VM or an Internal Component.
	providerCompInstance-Task	String	yes	ComponentInstanceTask name - owner of the providerInstance
	consumerInstance	String		Hosting consumer instance name

Table 6: Data Dictionary

Object	Attribute	Type	Opt.	Description
	consumerCompInstanceTask	String	yes	ComponentInstanceTask name - owner of the consumer-Instance
HostingTypeTask	name	String		Hosting name
	objType	String		hardcoded to "hosting"
	provider	String		Provider name
	providerCompTypeTask	String	yes	ComponentTypeTask name - owner of the Provider
	providedHostconfigName	String	yes	(Configuration) name
	providedHostdownloadCmd	String	yes	(Configuration) download command
	providedHostconfigureCmd	String	yes	(Configuration) configure command
	providedHostinstallCmd	String	yes	(Configuration) install command
	providedHoststartCmd	String	yes	(Configuration) start command
	providedHostStopCmd	String	yes	(Configuration) stop command
	providedHostuploadCmd	String	yes	(Configuration) upload command
	consumer	String		Consumer name
	consumerCompTypeTask	String	yes	ComponentTypeTask name - owner of the Consumer
	requiredHostconfigName	String	yes	(Configuration) name
	requiredHostdownloadCmd	String	yes	(Configuration) download command
	requiredHostconfigureCmd	String	yes	(Configuration) configure command
	requiredHostinstallCmd	String	yes	(Configuration) install command
	requiredHoststartCmd	String	yes	(Configuration) start command
	requiredHostStopCmd	String	yes	(Configuration) stop command
	requiredHostuploadCmd	String	yes	(Configuration) upload command