PaaSage

Model Based Cloud Platform Upperware

Deliverable D3.1.1 & D3.1.3

Upperware Prototype Report

Version: 1.0

D3.1.1

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

Mr Tom Williamson Tel: +33 4 9238 5072 Fax: +33 4 92385011 E-mail: tom.williamson@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:	29th August 2012
Document	
Period covered:	
Deliverable number:	D3.1.1 & D3.1.1E
Deliverable title	Upperware Prototype
Contractual Date of Delivery:	31th March 2013 (M18)
Actual Date of Delivery:	
Editor (s):	Christian Perez
Author (s):	Amin Bsila, Nicolas Ferry, Kamil Figiela, Geir Horn, Tom Kirkham, Maciej Malawski, Nikos Parlavantzas, Christian Perez, Jonathan Rouzaud-Cornabas, Daniel Romero, Alessandro Rossini, Arnor Solberg, Hui Song
Reviewer (s):	Benjamin Depardon, Philippe Massonet
Participant(s):	
Work package no.:	3
Work package title:	Upperware
Work package leader:	Christian Perez
Distribution:	
Version/Revision:	1.0
Draft/Final:	
Total number of pages (including cover):	

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.

Contents

List of Figures

List of Tables

Executive Summary

This document gives an overview of the architecture of the prototype of the Upperware layer of PAASAGE. The Upperware contains three main entities, known as the Profiler, the Reasoner, and the Adapter. This deliverable describes the initial implementation of each of them. It also describes four metamodels 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 M18. As long as experience will be gained using the developed software, the implementation of the Upperware layer can be updated. The next deliverable about final version of the Upperware is due at Month 36 (Product 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 [1].

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

1 Introduction

The Upperware is a collection of tools and components used to assist the application development or porting at design time, and then to integrate with the Executionware at runtime to facilitate the optimisation of the running application and execution platform. It is made of three main elements: the Profiler, the Reasoner, and the Adapter.

Deliverable D1.6.1 [1] has provided a high level view of these three elements. In particular, it gives some insights on the major sub-components, 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 metamodels private to the Upperware to ease separation of concerns, in particular with respect to the Profiler and the Adapter.

1.1 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, and its relationships with other PAASAGE elements. Next, Section 3 motivates and presents the four metamodels 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 three appendices. Appendix A fully describes the Upperware metamodels. Appendix B lists the full Cloud Provider-Independent Model (CPIM) of the Simple Application example introduced in Section 4.1. Appendix C fully represents the Saloon Ontology.

2 Upperware Architecture Overview

2.1 Overview

As defined in Deliverable [1], the first objective of the Upperware is to compute which commands to send to the Executionware from a CAMEL configuration model instance (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 maintain an actual deployment respecting the deployment constraints.

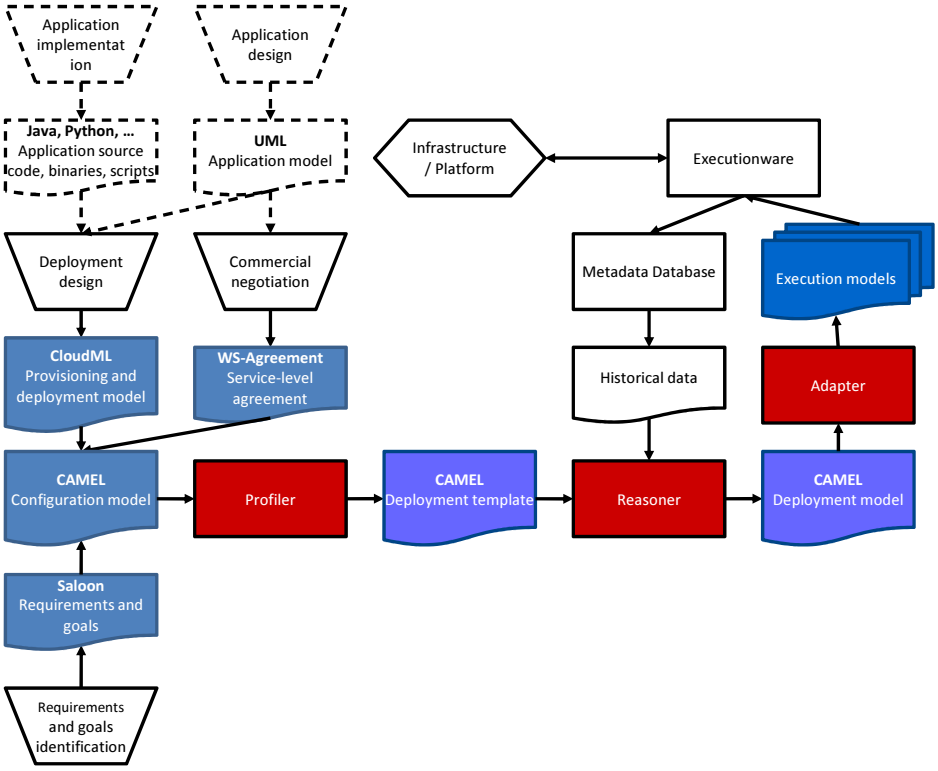

Figure 1: PAASAGE Workflow.

Figure 1 describes the PAASAGE workflow in more details. At the end of deployment description phase, that involves the deployment design, the commercial negotiation, and the identification of requirements and goals, all information is gathered into a CAMEL configuration model instance. This document

instance is the initial input of the Upperware. First, the Profiler analyses it to produce a CAMEL deployment template, that contains a list of potential candidate providers that satisfy the constraints. Second, the Reasoner computes a CAMEL deployment model instance, that is the chosen deployment solution. Third, the Adapter is responsible for transforming the output of the Reasoner into the target configuration in an efficient and consistent way, by issuing 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. However, as the Upperware Prototype at M18 only focuses on the initial deployment, this document does not cover application redeployment.

2.2 Upperware Metamodels

PAASAGE is a model based project. The Profiler and most Reasoner elements are based on the usage of some models. Therefore, we have defined four metamodels that aims at capturing elements important for Upperware components. Section 3 presents these metamodels 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 metamodel 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 metamodels 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 2.2, four metamodels have been defined to minimise model transformations and to minimise Reasoner dependencies to CAMEL. Figure 2 provides an overview of these metamodels and their relationships. The *Constraint Problem Metamodel* (CP Metamodel) and *Types Metamodel* enable the definition of the Cloud provider selection problem as a constraint problem. The *PaaSage Application Metamodel* (PaaSage App Metamodel) and *PaaSage Type Metamodel* establish the relationship between concepts from the Cloud and constraint problem worlds. These metamodels are presented in the following sections.

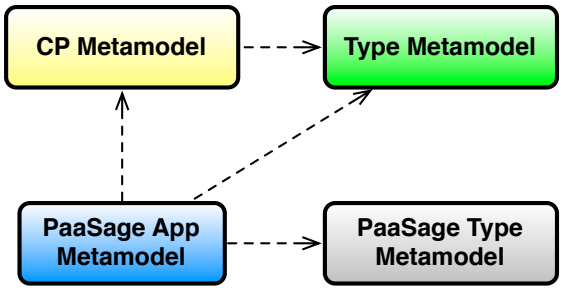

Figure 2: Metamodels overview.

3.1 Types and Constraint Problem Metamodel

Overview

The CP metamodel contains the different concepts needed to define a constraint problem: variables, constants, constraints and objective functions. Figure 3 presents an overview of this metamodel. 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.

Expressions

The *Expressions* are mainly *NumericExpressions* and *BooleanExpressions*. *Variables*, *Constants* and *ComposedExpressions* are numeric expressions as depicted in Figure 4. A *ComposedExpression* contains a set of numeric expressions related through an operation, *i.e.*, addition, subtraction, multiplication, division (*cf. OperatorEnum* in Figure 4).

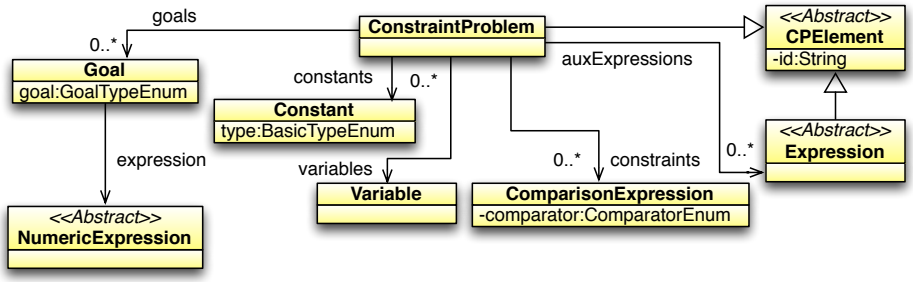

Figure 3: CP Metamodel overview.

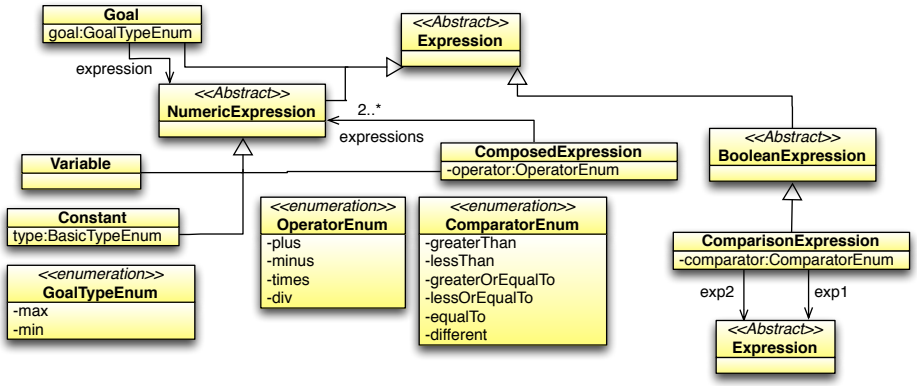

Figure 4: Expressions in the CP Metamodel.

A boolean expression is a *ComparisonExpression* that relates two expressions by using a comparator such as >,<, $\geq$, $\leq$, $=$ and $\neq$ (*cf. ComparatorEnum* in Figure 4).

Types, Variables and Constants

Figure 5 shows the *Types Metamodel* concepts in green, and the concepts in yellow are the ones related to CP Metamodel. As observed, this metamodel defines the basic values types, *i.e.*, integer, long integer, float, double, boolean and string, which are used to characterise variables and constants in a constraint problem.

A *Variable*, besides a *Value*, has a *Domain* that can be numeric or a list of strings. A *NumericDomain* is defined by basic types (*cf. BasicTypeEnum* in Figure 5). However, this kind of domain can be specialised for only considering a

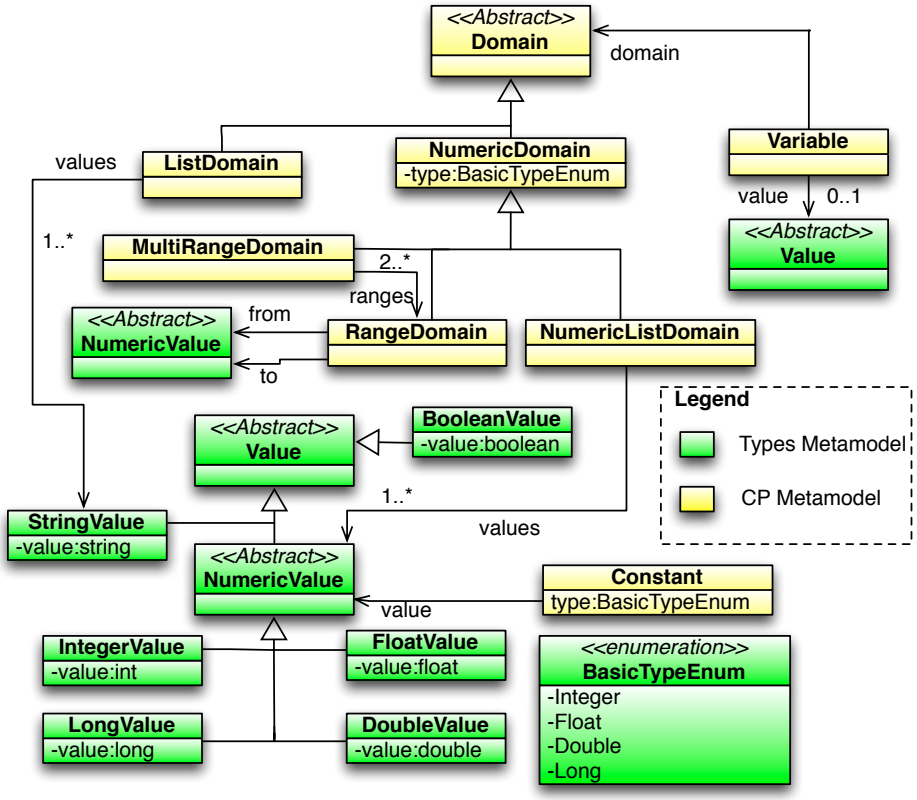

Figure 5: Type Metamodel: types, variables, and constants in the CP Metamodel.

subset of values through *NumericListRange*, *RangeDomain* and *MultiRangeDomain*. A *MultiRangeDomain* contains 2 or more ranges. On the other hand, a *Constant* also has a value but its domain is defined only by basic types.

Figure 37 in Appendix A depicts the whole *Types Metamodel* and *CP Metamodel* and their relationships.

3.2 PaaSage Type and Application Metamodels

Overview

The *PaaSage Application Metamodel* (or *PaaSage App Metamodel*) combined with the *PaaSage Type Metamodels* contain the required concepts to characterise an application to be deployed by using the PaaSage platform. Figure 6 provides an overview of these models. As observed, an application is described with a *PaaSageConfiguration* containing *VirtualMachineProfiles*, *ElasticityRules*, a set

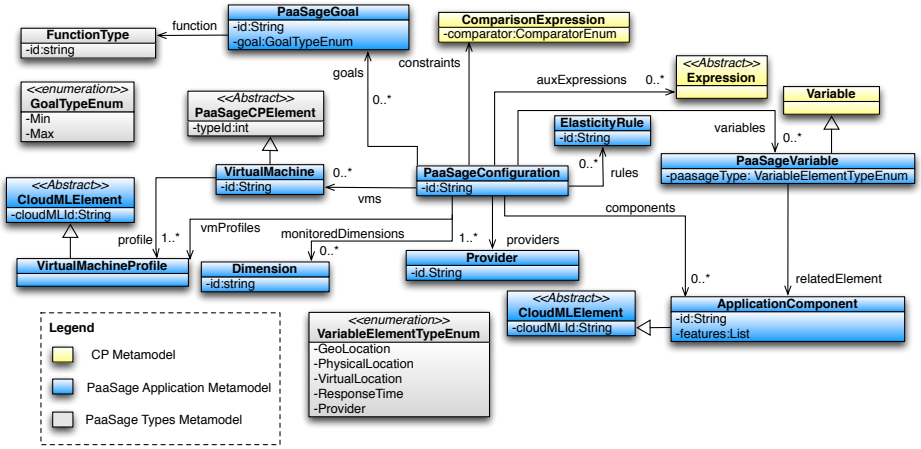

Figure 6: PaaSage Type and Application Metamodels overview.

of *Providers* and *Dimensions* to be monitored during the execution (*e.g.*, the application response time or the price evolution). 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 Metamodels* is to allow the derivation of the CP Model used by the `Reasoner` (*cf.* Section 5) and the generation of a *Cloud Platform Specific Model* (CPSM) for the `Adapter` component (*cf.* Section 6).

Virtual Machines and Providers

The Cloud providers are reified by the *ProviderType* class in the *PaaSage Type Metamodel* (*cf.* Figure 7). 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 Metamodel* 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 metamodel (*cf.* Figure 7): *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 7. These profiles are also related to a *ProviderCost* that depends on specific Cloud provider rates.

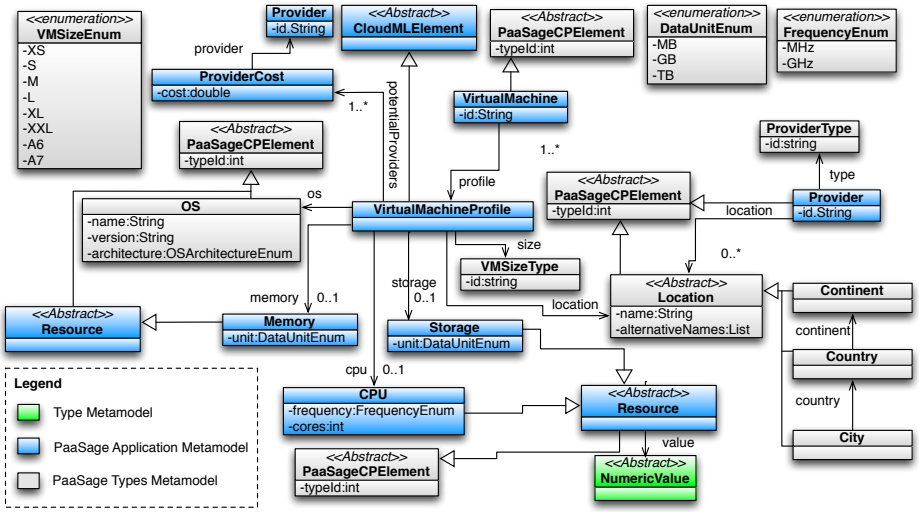

Figure 7: Virtual Machines and Providers in the PaaSage Type and App Metamodels.

Variables and Application Components

ApplicationComponents in the *PaaSage App Metamodel* represents the node types in CloudML [2]. An *ApplicationComponent* has a location, *RequiredFeatures* and an *ApplicationComponentProfile*. *RequiredFeatures* are dependencies to other application components and therefore they are provided by *ApplicationComponents*. *ApplicationComponentProfiles* characterises the application components in terms of name, version and *ApplicationComponentType*. An *ApplicationComponentProfile* example is Tomcat 7.0 with application server as type. An *ApplicationComponent* also has a list of preferred providers to be deployed and a VM that could contain it. Figure 8 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 and providers.

Elasticity Rules

ElasticityRules in the *PaaSage App Metamodel* reifies the rules enabling the dynamic adaptation of the application in order to deal with the trade-off between performance and cost for example (*cf.* [2]). They are composed by *Conditions* and *Actions*. *Conditions* are boolean expressions from the *CP Metamodel* while

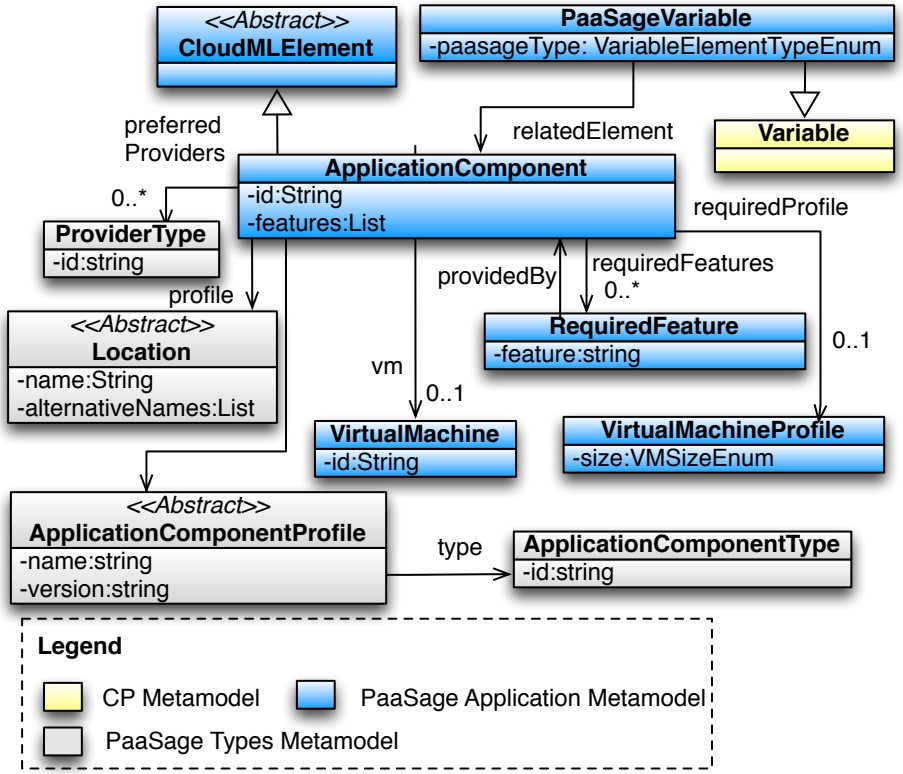

Figure 8: Application Components and Variables in the PaaSage Type and App Metamodel.

Actions represent typical actions on cloud providers such as "stop node" or "resume service". In this way, the rules are of the "*if...then*" form. Figure 9 depicts these *ElasticityRules*.

3.3 Example

In order to illustrate the use of the presented metamodels, 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.

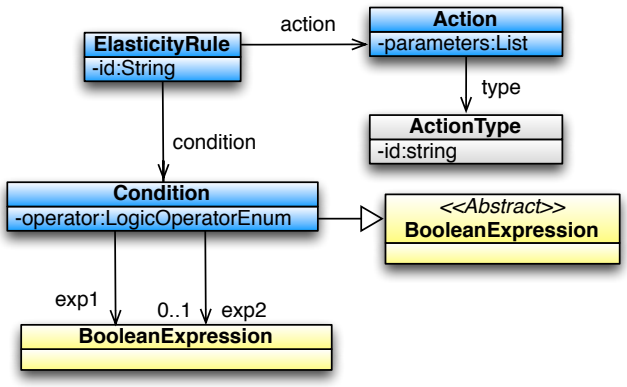

Figure 9: Elasticity Rules in the PaaSage Type and App Metamodels.

- *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.

$$\sum_{vm \in VMs} (number\_of_{vm}) \times Price_{vm} \equiv \tag{1}$$

$$(number\_of_{vm1\_amazon1}) \times Price_{vm1\_amazon1}) + (number\_of_{vm2\_elastichosts1}) \times Price_{vm2\_elastichosts1}) + \ldots \tag{2}$$

where

$VMs = \{vm1\_amazon1, vm2\_elastichosts1, vm3\_windowsAzure1\}$
$Price_{vm}$ = Cost of deploying Virtual Machine vm

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

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 each virtual machine, i.e., $vm1\_amazon1$, $vm2\_elastichosts1$ and $vm3\_windowaAzure1$, is related to an specific provider and satisfies the requirements of application in terms of memory, storage, CPU and operating system. The constraints to be satisfied in order to minimise this function are presented below.

- Each application component is only deployed in a virtual machine:

$$(\forall ac/ac \in COMPS : \sum_{vm \in VM} (appComponent_{ac} in_{vm}) = 1) \quad (3)$$

where

$$COMPS = \{simpleApplicationWar, tomcat\}$$

$$appComponent_{ac} in_{vm} = \text{Component } ac \text{ is deployed on } vm$$

- If a virtual machine is selected to deploy a component, at least there is one instance of the virtual machine:

$$(\forall vm/vm \in VM : \sum_{ac \in COMPS} (appComponent_{ac} in_{vm}) \leq number\_of_{vm}) \quad (4)$$

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

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

$$(\forall ac1, ac2/ac1, ac2 \in COMPS : (\forall vm/vm \in VM : appComponent_{ac1} in_{vm} = appComponent_{ac2} in_{vm})) \quad (5)$$

- At least one virtual machine instance is required:

$$\sum_{vm \in VMs} number\_of_{vm} > 0 \quad (6)$$

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, and the profiles of virtual machines (of these providers) that support the required resources.

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. Finally, Figure 12 *c)* indicates that Simple Application is a WAR that should be deployed in Europe on any of the three providers.

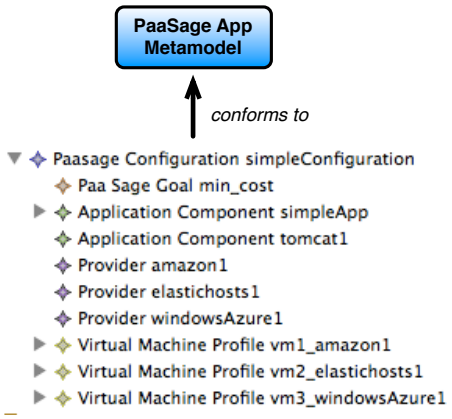

Figure 11: PaaSage Model of the Simple Application.

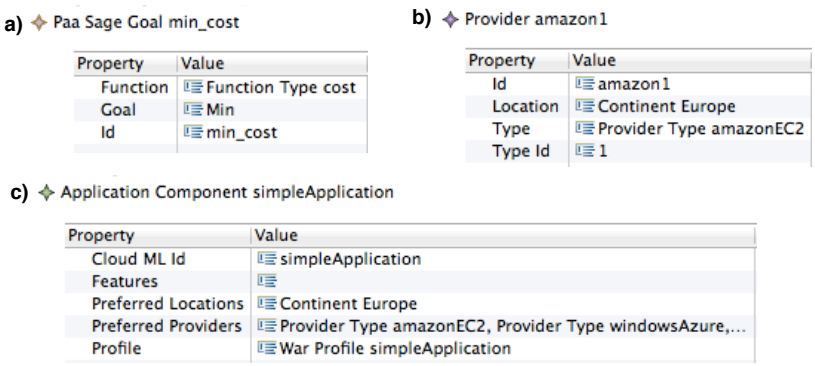

Figure 12: Properties of some elements of the Simple Application example.

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 are minimisation or maximisation of dimensions that have important business impact, such as cost or application response time.

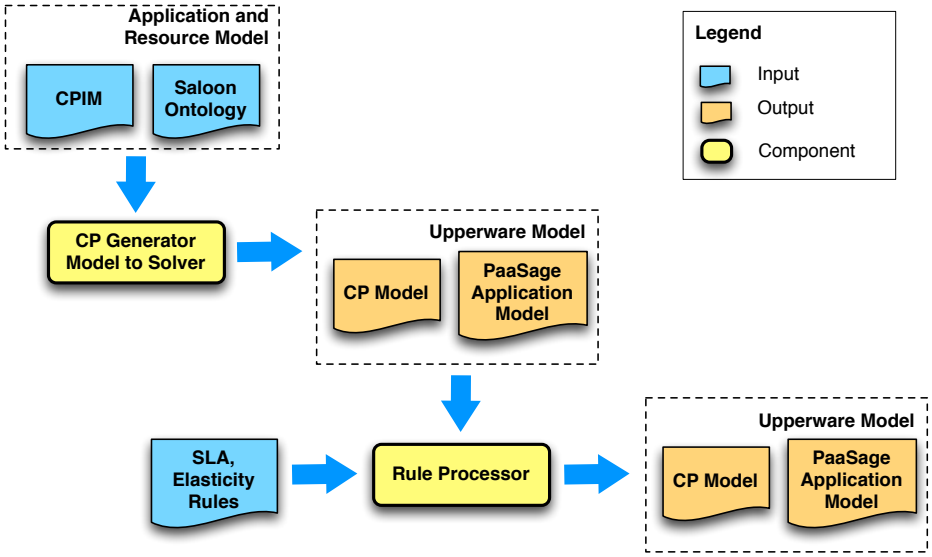

Figure 13: Profiler Architecture.

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 *Cloud Platform Independent Model* (CPIM) and a *Saloon ontology* (*cf.* [2]), which captures the requirements in terms of com-

Listing 1: CPIM for the Simple Application example (excerpt)

```
<net.cloudml:CloudMLModel xmi:version="2.0"...
                          xmlns:net.cloudml="http://cloudml.net"...
                          name="SimpleAppModel">

    <providers name="AmazonEC2"
               credentials="./credentials_aws"/>
    ...
    <components xsi:type="net.cloudml:VM"
                name="ML"
                provider="AmazonEC2"
                location="EU"
                minRam="4096"
                minCores="4"
                minStorage="102400"
                os="ubuntu"
                securityGroup="simpleApp"
                sshKey="simpleApp"
                privateKey="/simpleapp.pem"
                groupName="simpleApp">

            <providedContainmentPorts name="mlProvided"
                                      owner="ML"/>
    </components>
    ...
</net.cloudml:CloudMLModel>
```

putational resources and services related to the PAASAGE 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 CPIM. Listing 1 presents an excerpt of the CPIM for the Simple Application (*cf.* Section 3.3). In particular, it shows "amazonEC2" as the cloud provider and a virtual machine (named "ML") provided by "amazonEC2" and satisfying the application requirements in terms of resources. On the other hand, Figure 14 depicts an excerpt of the *Saloon ontology* used to specify some of the application requirements. As observed, the selected concepts and some of the related values correspond to the application requirements specified in Section 3.3. A complete version of the CPIM and *Saloon ontology* are defined in Appendix B and C, respectively.

Implementation

Figure 15 depicts the various components that compose the `CP Generator Model-to-Solver` component. The `Processor Factory` components load the models received as input and instantiate `Model Processors`, which encapsulate the functionality to parse and extract the elements required to build a *PaaSage Application Model* and a *CP Model* related to the application to be deployed.

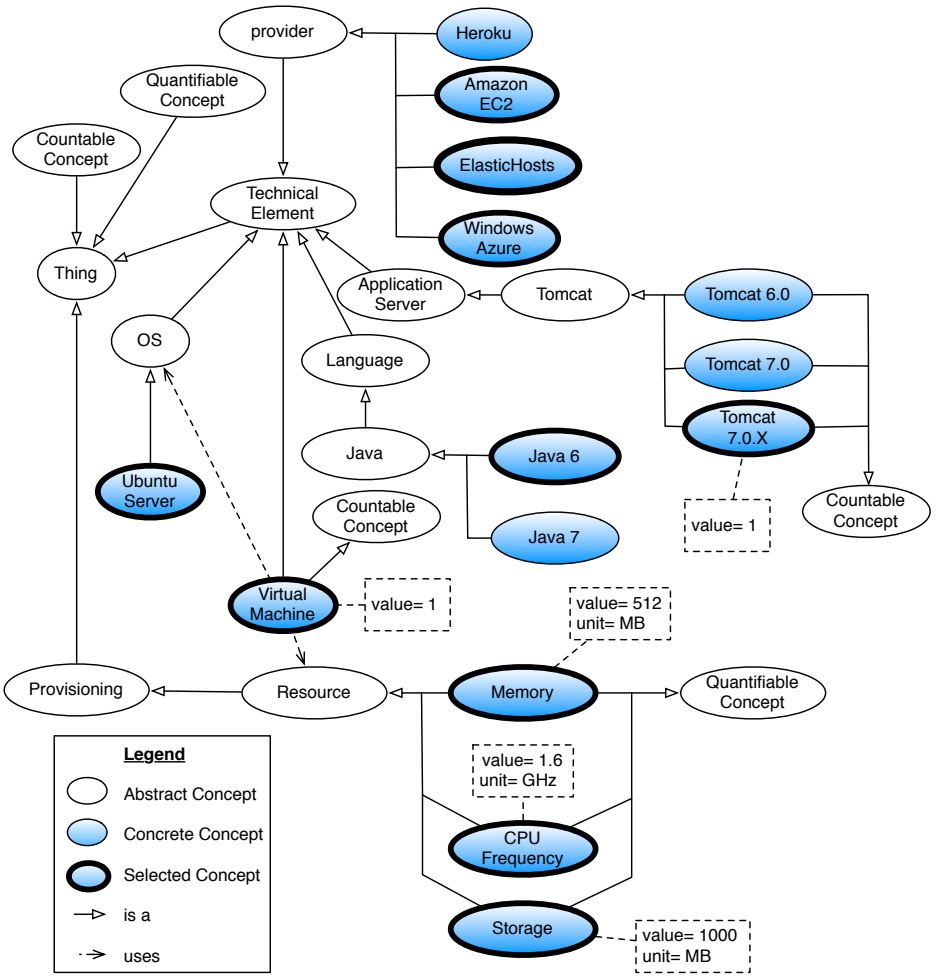

Figure 14: Saloon Ontology (excerpt) with the selected concepts for the Simple Application example.

The current implementation of the `CP Generator Model-to-Solver` has a `Processor Factory` and a `Model Processors` for both the *Domain Specific Language* (DSL) used for the input description, *i.e.*, CloudML and Saalon (*cf.* [2]). The CloudML `Processor Factory` and the Saloon `Processor Factory` exploit EMF to load an object representation of the CPIM model and Saloon Ontology, respectively. The `Model Processor` for CloudML 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 Saloon `Model`

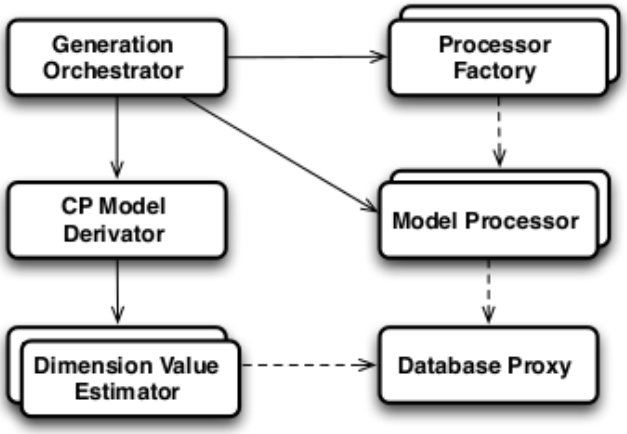

Figure 15: CP Generator - Model to Solver Architecture.

`Processor` benefits from the core of the Saloon Framework [3], which is based on the CHOCO library[5] for constraint programming, to filter the Cloud providers defined by the CPIM according to the application requirements reified by Saloon Ontology. To do that, the `Model Processor` retrieves the Feature Models (FMs) capturing the characteristics of the involved providers (*cf.* [2]) through the `Database Proxy`, which encapsulates the access to the PAASAGE Database (*cf.* [2]). Each FM is translated to a set of constraints and variables that are passed to the CHOCO Solver in order to look for solutions or configurations that are supported by the Cloud providers. Only valid FM configurations, which satisfy the application requirements, are selected.

The `CP Model Derivator` component processes the *PaaSage Application Model* produced by the `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 or time, the `CP Model Derivator` will use the suitable `Estimator` component. For example, the `Price Estimator` computes the cost of using the Cloud provider candidates selected by the `PaaSage Model Processors`. The information related to provider rates is retrieved through the `Database Proxy`. Finally, the `Generator Orchestrator` coordinates the whole model processes that leads to the generation of the output models, which are sent to the `Rule Processor` via RabbitMQ[6], a robust message

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

[6]RabbitMQ:`http://www.rabbitmq.com/`

broker that runs on all major operating systems. The use of RabbitMQ enables a potential distribution of `Profiler` components on different machines. Figure 10 and Figure 11 present a screenshot of the output models in the EMF editor.

Summary

The `CP Generator Model-to-Solver` component has a CPIM and a *Saloon Ontology* as inputs. The outputs are a *CP Model* and a *PaaSage Application Model*. Considering the different subcomponents of the `CP Generator Model-to-Solver` (*cf.* Figure 15) the following process is executed in order to generate these outputs:

1. The `Generation Orchestrator` creates the `Model Processor` for CloudML and Saloon by using the `Processor Factories` as well as an empty *PaaSage App Model*.
2. The `Generation Orchestrator` loads the CPIM and *Saloon Ontology* through the `Model Processors`.
3. The CloudML `Model Processor` extracts from the CPIM information related to virtual machines, providers and application components and fills the *PaaSage App Model* with them.
4. The Saloon `Model Processor` filters the providers in the *PaaSage App Model* according to the requirements defined in the ontology.
5. The `Derivator` generates a *CP Model* from the *PaaSage App Model* and uses the `Dimension Value Estimator` to retrieve values of the required dimensions.
6. The generated models are sent to the `Rule Processor` component.

The previous process is also depicted in Figure 16.

4.2 Rule Processor

Overview

The `Rule Processor` in PAASAGE receives a list of potential Cloud providers provided by the `Profiler` based on requirements specified by the application designer via the IDE. The `Rule Processor` checks these providers against implementation specific rules in the *Metadata Database* (MDDB). These

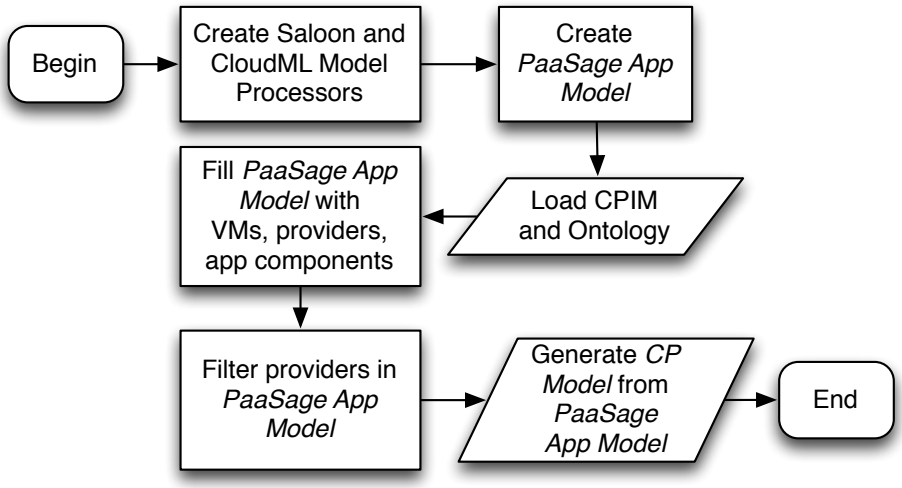

Figure 16: Process executed by the CP Generator Model-to-Solver component.

rules are the base rules of the system and are expressed in terms of performance and data processing constraints associated with the specific instance of the PAASAGE platform.

The `Rule Processor` verifies that the list of possible deployments or cloud providers satisfy all the given constraints from the data in the MDDB. Deployments formed by the `Rule Processor` take into account the constraints and details of the implementation in the MDDB. For example, the `Rule Processor` could have a requirement that data is processed in the EU. The MDDB could contain SLA specific data from associated service providers that fulfil this rule, and thus it is added to the list of deployments. If during this phase the `Rule Processor` encounters a requirement that can not be fulfilled by the PAASAGE instance such as the application should not use Amazon servers, the `Rule Processor` returns to a error message containing this detail to be fed back to the application designer.

Implementation

The `Rule Processor` receives as an input the CP Description from the `CP Generator` that defines a list of input constraints for the future deployment. The content of the CP Description is a `CP Model` and `PaaSage Application Model` describing characteristics for the application, such as service level objective, CPU and more generic provider goals. Then, the CP Description is passed into the `Rule Processor` as a RabbitMQ call.

RabbitMQ is used by the Profiler in order to handle the communication between the `CP Generator Model-to-Solver` and the `Rule Processor`. Since RabbitMQ calls or messages are asynchronous and stored in a queue, it provides reliability against failures and high scalability, where there could be many instances of Profiler and RabbitMQ's queue depending on the load demand.

The `Rule Processor` runs as a Java servlet calling the MDDB to extract platform information. The result is a CP Description that lists possible and feasible deployments - wrapped into 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 a CAMEL data and its associated Upperware metamodel intances onto the `Reasoner`.

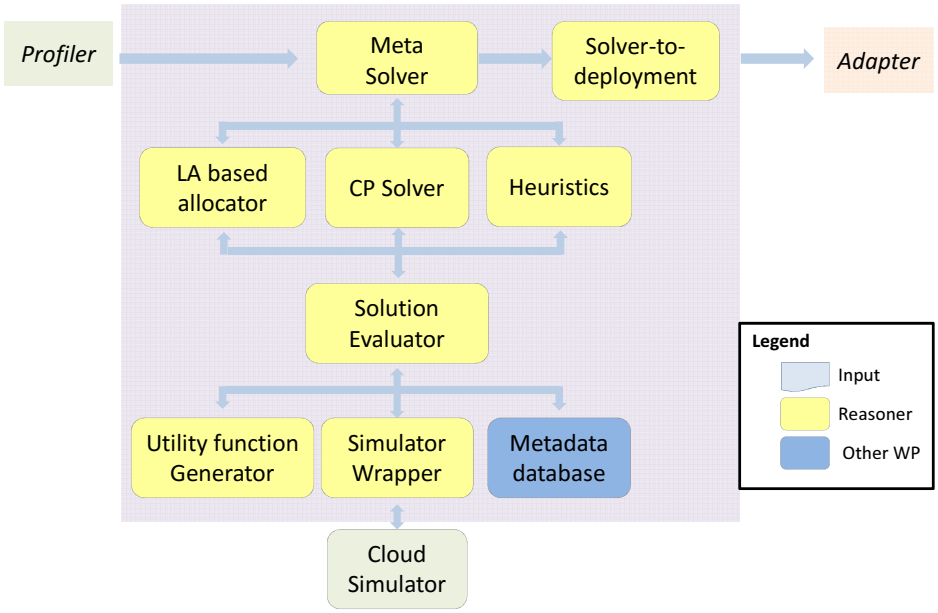

Figure 17: Reasoner: Architecture and main Components.

5 Reasoner

In a nutshell the Reasoner receives application and context models (from the Profiler) in CAMEL format and outputs Deployment Models in CAMEL. This process relies on the Reasoner extracting requirements from the CAMEL and using the current state of Cloud Infrastructure and knowledge from the metadata database to conduct reasoning.

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, and Meta-Solvers. These solvers have access to various external methods to evaluate solutions if needed: i) Utility Function Generator provides a quick evaluation of a solution based on a utility function. ii) Simulation Wrapper makes use of an external Cloud simulator, such as SimGrid, to provide a more accurate solution evaluation but a higher execution cost. iii) The MDDB provides access to historical data.

When a solution has been computed, the Solver-to-deployment component translates the solution in a CAMEL Deployment Model format.

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 [4]. There have rightly been attempts on logic based stochastic reasoning like the Probabilistic Logic Network [5] or, more recently, the Non-Axiomatic Logic [6], 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 [7] 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.

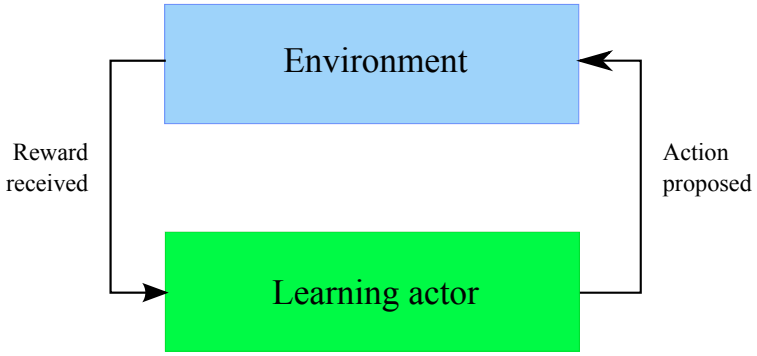

Figure 18: The fundamental learning loop: The learning actor proposes an action to the environment. In this case, the action is a particular deployment configuration. The environment then provides feedback on the quality of this configuration in terms of a reward to the learning agent.

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 5.8.

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[7]. 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 [8]:

$$\text{maximize } U(\mathbf{x}) \tag{7}$$

subject to

$$\mathbf{g}(\mathbf{x}) \leq \mathbf{0} \tag{8}$$

$$\mathbf{h}(\mathbf{x}) = \mathbf{0} \tag{9}$$

$$x_i \in \mathbb{X}_i \tag{10}$$

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 [9], otherwise it is a *combinatorial optimisation* problem [10]. 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 each discrete

[7]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.

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 (7) 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 (8) and (9) 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:

$$\text{maximise } E\left\{U(\mathbf{x})\right\} \tag{11}$$

subject to

$$E\left\{\mathbf{g}(\mathbf{x})\right\} \leq \mathbf{0} \tag{12}$$

$$E\left\{\mathbf{h}(\mathbf{x})\right\} = \mathbf{0} \tag{13}$$

$$x_i \in \mathbb{X}_i \tag{14}$$

If the distributions for the parameters x_i were known, the problem could be approached with *stochastic programming* [11]. 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* [12], and it is attractive that efficient methods exist

for robust integer programming [13]. 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 [14]. For the parameters with discrete domains, *i.e.* the combinatorial optimisation part, we will adopt a *reinforcement learning* [15] approach based on *Learning Automata* [16]. 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 [17] developed a theory for stochastic optimisation using learning automata based on Baba's multi-teacher approach [18] where the utility function (7) and the constraints (8) and (9) 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* [19]. 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 [8], where each iteration requires sampling the objective function (7). 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 exploring the configuration space quickly, by sampling many distant configurations. However, it may have a negative impact on the convergence of the learning al-

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** $x_i \in \mathbb{X}_{\textit{Discrete}}$ **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** $x_i \in \mathbb{X}_{\textit{Discrete}}$ **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** $x_i \in \mathbb{X}_{\textit{Discrete}}$ **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*

gorithms [20]. 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* [21] 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[8], released as open source under the MIT licence. The Theron library is robust, efficient, and complete compared with other alternatives: *libcppa*[9] 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*[10] implementation seems to be a minimal fragment not actively maintained. *libprocess*[11] adopts the view that each actor is a *process*. It is poorly documented with the best information source being a presentation[12] 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 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

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

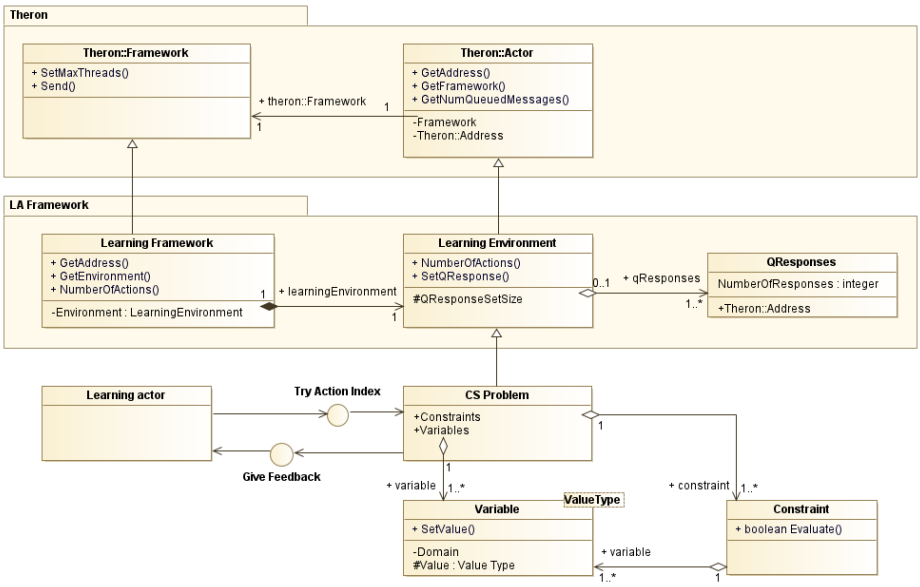

Figure 19: The learning environment controlling the problem variables and constraints is an actor that interacts with a learning actors through messages.

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*[13] linear algebra library to

[13] `http://arma.sourceforge.net/`

represent matrices. It was selected over the *Eigen*[14] 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*[15] machine learning library.

Only a few years after the first paper on FSSA [22], Varshavskii and Vorontsova introduced the family of Variable Structure Stochastic Automata (VSSA) [23]. 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 [24]. 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.

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

[14] `http://eigen.tuxfamily.org`
[15] `http://mlpack.org/`

its views on the better choices for a particular variable. Chandrasekaran and Shen [25] 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 [26].

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 metamodel 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.[16]

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 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

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

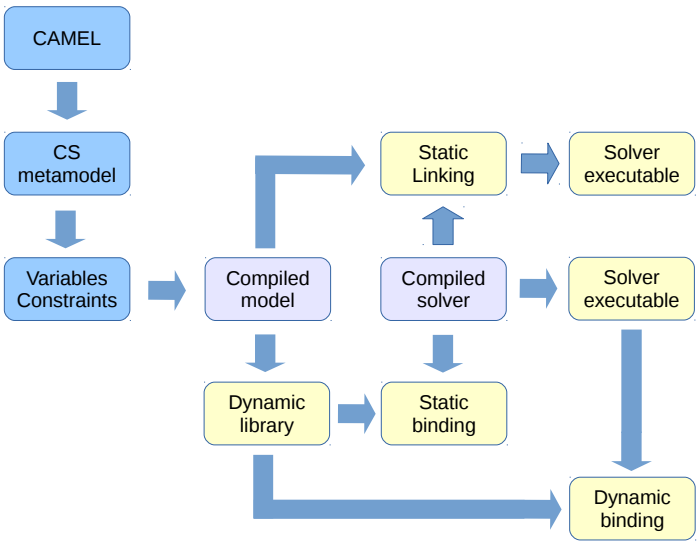

Figure 20: The various options of binding the solver code with the compiled variables and constraints of the problem at hand.

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 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

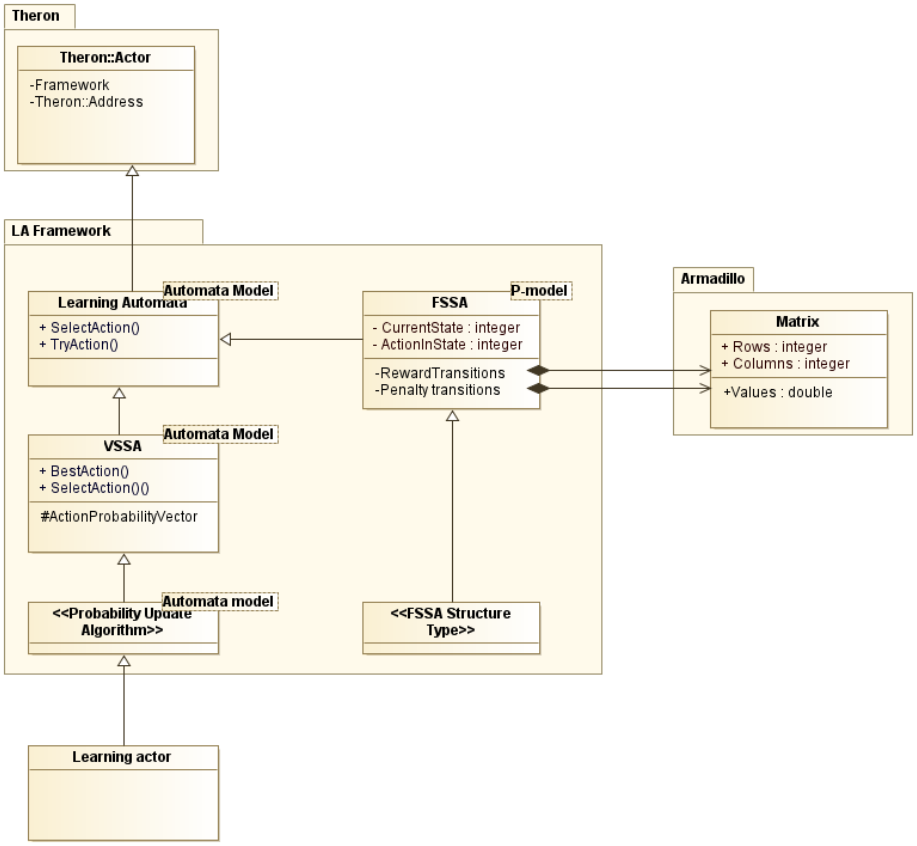

Figure 21: The hierarchy of a learning actor implementing Variable Structure Stochastic Learning. Alternatively the Learning Actor could have inherited the class implementing a Fixed Structure Stochastic Learning type.

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 CP Solver

Overview

One of the Solver components being developed for PAASAGE [27] exploits Constraint Logic Programming (CLP) techniques and is based on the following rationale:

- Use logic rules to map high-level (application or component) requirements to low-level (VM or PaaS) requirements.
- 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.
- Express optimisation formulas involving utility functions on quality metrics and attributes as well as cost.

The general formation of a CLP problem that will have to be solved in order to produce a particular deployment solution has the following form:

1. maximise weighted sum of utility functions application on the metric values considered for each high-level metric (plus cost).
2. constraints on how low metric values can be propagated to higher metric values.
3. constraints on low and high level metric values for VMs as well as applications and application components, respectively.
4. constraints concerning the co-location of application components.
5. constraints dictating that only one VM should be selected for the deployment for each application component.

One of the challenges with CLP 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 consider two 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: 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.

An interesting aspect of this performance problem is what happens when a particular application is not equivalent but similar with other ones (*e.g.* might use a percentage of identical or equivalent components) and how this similarity could be exploited to reduce the solution space. This is an interesting research direction that will be explored to further enhance the performance and accuracy of the reasoner that it develops.

Implementation

The current CLP solver prototype has been written in Prolog within the ECLiPSe environment[17] and has been tested according to a particular use case (traffic management) [27] which includes the description of requirements at a high level and their mapping to low-level ones through rules as well as co-locality constraints. The interface of the prototype is not yet available and thus can be used in a standalone manner for the time being. For integration into the Upperware, the output

[17] `www.eclipseclp.org`

of the Profiler (Upperware metamodel instance) provided as input to the reasoner will be transformed into a CLP form that is appropriate for the ECLiPSe Prolog-based environment. The output produced by the CLP reasoner will be stored back into instances of Camel's deployment metamodel. The integration and co-operation with the Knowledge Base will be implemented after M18.

5.3 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 [28]. The `MILP solver` is designed to be generic, so that it can be used for two purposes:

- as a generic solver within the `Reasoner` component,
- as an application-specific solver for `workflow planner` component of `HyperFlow` workflow execution system.

Moreover, the solver should also be able to operate in a standalone mode, so that it can be used for development and testing of optimisation models of various application and infrastructure configurations and their parameters [29].

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 [30] 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, solution evaluator and MDDB). It translates data between PaaSage CP model and internal problem description and binds the different technologies used.

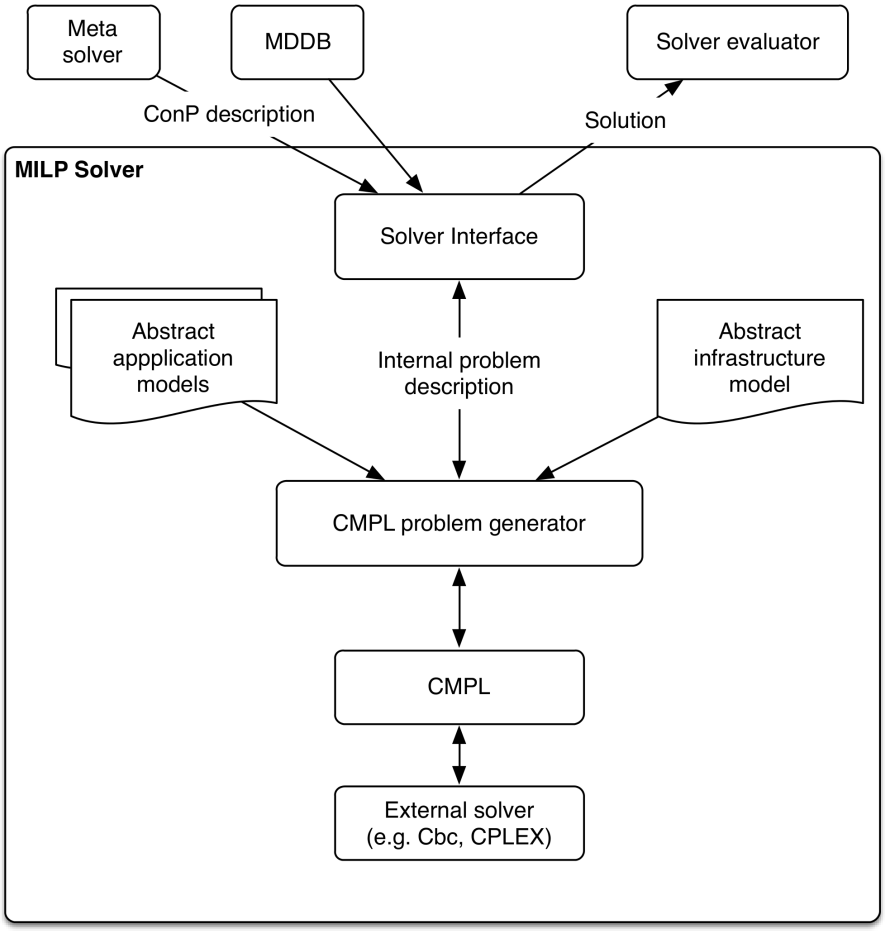

Figure 22: Architecture of MILP solver.

CMPL problem generator The goal of this component is to generate an instance of MILP problem that is subject to optimisation by CMPL. For this purpose it combines data provided by PAASAGE with optional predefined abstract application and infrastructure models, to create a complete MILP problem description. For example, the application model of a workflow needs to be provided by the user when operating within the workflow planner component.

Abstract application models The solver embeds a set of predefined models of supported application classes represented in CMPL notation. The model

represents the performance model of the application, *i.e.* it describes the expected performance and scalability of application components. It also includes constraints of application deployment. We will provide a model for scientific workflow applications, but the solver can be extended to support other application classes.

Abstract infrastructure model The solver also embeds an optional model of cloud infrastructure which is also represented in CMPL notation. It includes constraints of cloud infrastructure that affect application deployment.

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).

It is possible to use MILP solver in a standalone mode by interfacing directly to **CMPL problem generator**. This feature enables model developers to create and test optimisation models without the need of having a complete PAASAGE platform installed. It also makes it possible to use solver as an application-specific component for *e.g.* workflow execution planning.

The **internal problem description** uses a lightweight data format such as YAML in order to improve solver interoperability with different technologies. The input for the problem generator includes application description (*e.g.* workflow structure, historical performance data), cloud infrastructure description and user defined constraints and objective. As a result of optimisation the best deployment plan will be generated.

Model example

The Simple Application similar to the one given in Section 3.3, represented in CMPL is shown in Figure 23. Since CMPL allows defining constants and variables as sets or arrays, the model has a concise notation. The objectives and constraints are represented as equations directly corresponding to the mathematical notation of Equation 1.

Implementation

The current version of the prototype includes the CMPL problem generator developed in Ruby and a set of abstract application and infrastructure models in CMPL for layered workflows, as well as simplified sample applications. The solver interface is not available yet, so the solver can operate in standalone model only.

Figure 23: Example of CMPL model for simple cloud application.

```
%allowed_vms set[2] <
   vm1 amazon1
   vm2 amazon1
   vm1 elastichosts1
   vm2 rackspace
   vm3 private
>
%cost[allowed_vms] < 1 2 1.5 0.4 7 >

variables:
   {[vm, p] in allowed_vms: vm_in[vm,p]: binary; }

objectives:
   sum {[vm, p] in allowed_vms:
          cost[vm, p] * vm_in[vm,p]
       } -> min;

constraints:
   sum {[vm, p] in allowed_vms: vm_in[vm,p] } = 1;
```

5.4 Heuristics

Overview

Exact solution solvers (*e.g.* CP and MILP Solver) have to evaluate all the possible solutions to find the optimal one. But when the search space increases, the time to compute the optimal solution is growing exponentially. At some point, the solver will take too much time or will be impossible to process all the solutions. When using a multi-Cloud platform and a medium to large application, CP and MILP solvers might not be able to scale and give a solution within a reasonable time. Using a heuristic approach, it is possible to find a good enough solution within a smaller amount of time. However, such a heuristic is only valid for some specific cases. For example, recurrent business use cases may deserve a heuristic, letting general solver for unsupported use cases. Heuristics shall be typically controlled by a `Meta Solver` component.

Our main focus for the M18 prototype is to show how such an algorithm can be integrated into the PAASAGE platform.

Design

The internal of the heuristic reasoner is shown in the Figure 24. The heuristic reasoner takes as input instances of the `PaaSage application` and `CP metamodels`, but also a connection to the MDDB. Using them, it generates the problem for the heuristic, *i.e.* it instantiates heuristic specific structures.

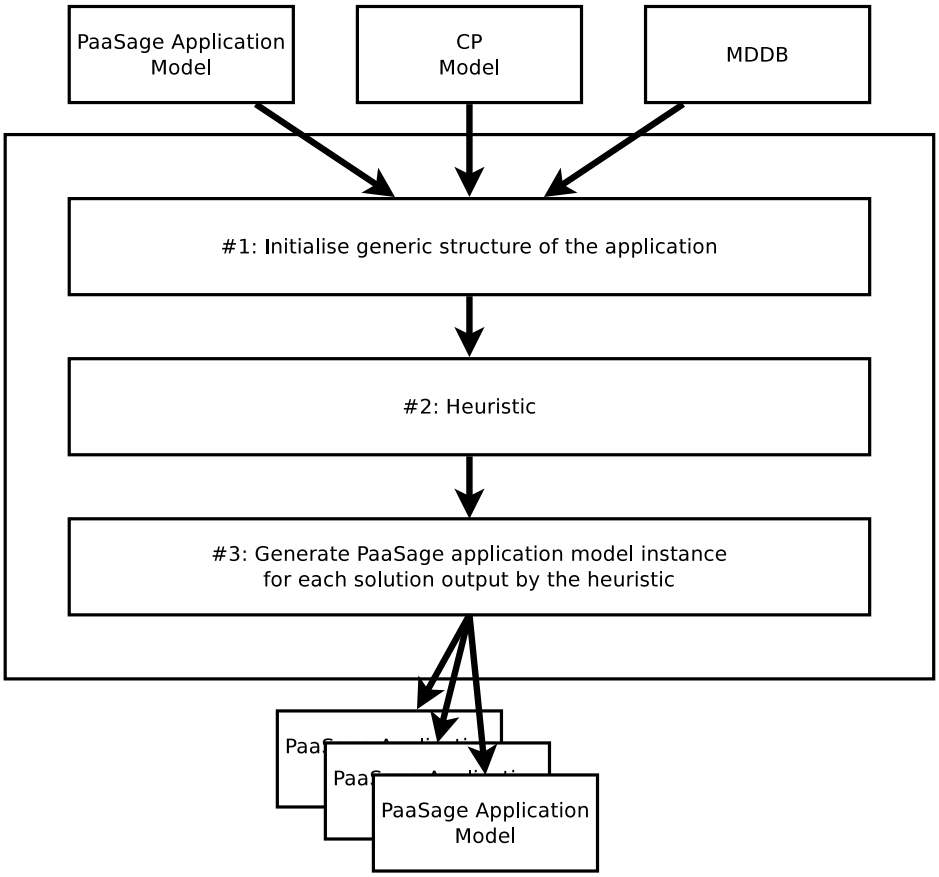

Figure 24: Architecture of Heuristic reasoner.

The heuristic returns one (or multiple) solution(s). Each of these solutions are injected into an instance of the `PaaSage application` metamodel with each parameter set.

Even if we only present one heuristic here, the heuristic interface (block #2 in Figure 24) can be used by any heuristic.

Horizontal scalability heuristic

For illustration purposes, this section focuses on a horizontal scalability heuristic presented in Algorithm 2. The algorithm takes as input the list of application components ($components$) and a set of constraints ($Constraints$). Each component contains the list of available VM profiles and their costs. Furthermore, if historical data or performance models are available, each VM profile contains

Algorithm 2: Horizontal scalability heuristics with constraints.

Require: $components, Constraints$
Ensure: $solution$
if historicalData(applicationComponent) OR performanceModel(applicationComponent) **then**
for all applicationComponent : components **do**
for vmProfile: $applicationComponent_{VirtualMachineProfiles}$ **do**
if $Constraints_{Cost} > 0$ **then**
if $vmProfile_{Cost} \leq \frac{Constraints_{Cost}}{|components|}$ **then**
if $Constraints_{RTT} > 0$ **then**
if $vmProfile_{RTT} \leq \frac{Constraints_{RTT}}{|components|}$ **then**
if $Constraints_{Throughput} > 0$ **then**
if $vmProfile_{Throughput} \geq \frac{Constraints_{Throughput}}{|components|}$ **then**
$normalizedThroughput = \frac{vmProfile_{Troughput}}{max(applicationComponent_{VirtualMachineProfiles_{Throughput}})}$
$normalizedCost = \frac{vmProfile_{Cost}}{max(applicationComponent_{VirtualMachineProfiles_{Cost}})}$
$tradeOff_{vmProfile} = \frac{normalizedThroughput}{normalizedCost}$
end if
end if
end if
end if
end if
end if
end for
$solution_{applicationComponent} = max(tradeOff)_{vmProfile}$
end for
else
for all applicationComponent : components **do**
for vmProfile : $applicationComponent_{VirtualMachineProfiles}$ **do**
$normalizedFLOPS = \frac{vmProfile_{FLOPS}}{max(applicationComponent_{VirtualMachineProfiles_{FLOPS}})}$
$normalizedCost = \frac{vmProfile_{Cost}}{max(applicationComponent_{VirtualMachineProfiles_{Cost}})}$
$tradeOff_{vmProfile} = \frac{normalizedFLOPS}{normalizedCost}$
end for
$solution_{applicationComponent} = max(tradeOff)_{vmProfile}$
end for
end if

the predicted throughput and round-trip time (RTT) for each component. If no historical data and performance models are available, each VM profile contains the available FLOPS based on the processor architecture. The heuristic outputs a solution, *i.e.* a mapping between each component and a selected VM profile.

This heuristic only focuses on selecting the good enough set of VM profiles for a set of application components. Furthermore, it only enforces 3 constraints: maximum price, maximum end-to-end RTT and minimum throughput. Moreover, the enforcing is straightforward: the constraints is divided by the number of components and then checked for each of them. For example, if the maximum cost is set to 8 € and the application has 4 components, the heuristic only searches for sub 2 € VM profiles. Each fitting solution is compared and

ranked based on a trade-off between throughput and monetary cost. The monetary cost only includes the price of the VMs and excludes the cost of storage and network transfers. If it is not possible to predict the throughput and RTT for a given component and VM profile, *i.e.* no historical data and performance model, the heuristic falls back to a simple trade-off between the FLOPS available on each VM profile and their cost.

Implementation

Implementing Algorithm 2 is straightforward in any programming language. As an exercise for the M18 prototype, we have implemented in Java a version without MDDB access, *i.e.* without historical data, nor performance model. The implementation exploits the Eclipse Modelling Framework (EMF) to extract information from the metamodel instances.

5.5 Meta-Solver

Overview

The Meta-Solver receives the deployment requirements from the Profiler and extracts from this a set of variables to express the requirements as a deployment problem. In the Reasoning architecture there are many algorithms, or solvers, that can be used to assign values to the Deployment problem. When breaking down the Profiler deployment requirements into the deployment problem the Meta-Solver has to create a series of variables that the Solvers can process.

Variables created by the Meta-Solver have significant characteristics which determine the choice of reasoning Solver. An initial factor is the nature of the relationship between variables either linear or non-linear. In addition various domain related characteristics has to be applied to the variables, such as intervals over real numbers or as integers, including binary decision variables. The Meta-Solver will select one or more solvers appropriate for the Deployment problem based on the individual and available Solver characteristics.

The Meta-Solver when sending the deployment problem to one or more Solvers will take into account the choice of Solver and break the variables into a single problem or as parts of a problem to different Solvers. Deployment problems can be dispatched by the Meta-Solver as part of a workflow using results from previous Solvers as input.

Finding the optimal configuration is normally only possible for certain restricted problems, and in general one will have to evaluate every possible feasible configurations in order to assess a posteriori the best configuration. This is impractical for all but the smallest problems. In reality one will therefore need to run the solvers for as many iterations as allowed by the time budget available

for finding a configuration. Thus, it is a task of the Meta-Solver to control the execution of the individual solvers, and stop or pause them when a solution must be returned.

Implementation

The Meta Solver will act as an initial point at which requirements are broken into problems and sent to specific solvers for solutions. The duration at which the solvers are run depends on other requirements taken into account by the Meta Solver such as resource constraints or the output from the individual solvers. The Meta Solver will communicate with other components using the Upperware meta-model developed in WP3. It will not pass other data to Solvers and therefore does not parse or produce CAMEL. Implemented using Java the component can be used as a standalone jar or scaled up into a RESTFUL web service to allow distributed use.

5.6 Solution Evaluator

The `Solution Evaluator` component aims at offering a standardised function evaluation interface to all solvers. It interconnects solvers with the utility `Function Generator`, the `Simulation Wrapper`, and the Metadata Database (MDDB). `Solution Evaluator` gets the application model from solvers with the function to evaluate and forwards it to the `Utility Function Generator`, the `Simulation Wrapper`, or the Metadata Database (MDDB) depending on the form of the function to evaluate. It communicates with other PAASAGE components through interfaces provided by those components. In the initial prototype (M18), `Solution Evaluator` is optional as it proposes three distinct interfaces, one for every backend it supports. Therefore, it can be bypass by other components. The `Solution Evaluator` component is a simple dispatch component.

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 [31].

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 [32]. Chu and Halpern showed that essentially all decision rules can be represented as a generalised notion of expected utility [33]. 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 [34]. The concept of utility for system management and operation is closely linked with the vision of autonomic computing [35] and was first used to allocate resources in a data centre [36], 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 [37]. 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 [38].

Utility functions were used for self-adaptive applications in the MADAM project [39], seeding ideas that were carried forward for context aware ubiquitous computing systems in the MUSIC project [40]. Extensive experimentation with real applications revealed that even experienced software developers would find it difficult to develop utility functions [41], and that they often reduced the utility function to some kind of situation-action rule [42]. These experiments thus confirmed the conjecture of Walsh *et al.* [38] 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" [43]. 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"* [44].

Another approach to elicit a utility function was offered by Valetto *et al.* [45]. 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 [37]. The best configuration can then be found in polynomial time by an application of the Bellman-Ford algorithm [46] as explained in [47]. However, the experiments in MUSIC showed that decomposable utility functions cannot be expected in general [42].

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" [43]. 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* [48] to establish a utility value in the unit interval, $[0, 1]$. This approach entails the following steps that are illustrated in Figure 25.

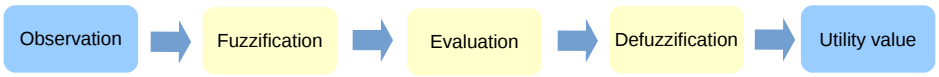

Figure 25: The steps needed in order to compute the fuzzy utility value.

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 Simulator Wrapper

Overview

The goal of this component is to provide to the PAASAGE platform the ability to evaluate different resources' allocations of an application prior to deploying it on Clouds. Based on the application model and the MDDB, an application mockup is generated to simulate the behaviour of the application. Moreover, based on historical data and/or analytic performance model, the resource consumption of the different workloads of the application are generated. The same is true for the platform based on information retrieved from the MDDB for the performance and behaviour of the platform and from the application model for the Cloud offering (VM type, price, etc.). Based on the application and CP models, `Simulator Wrapper` generates one or multiple applications to Cloud resources mapping. Each simulation is then run by the selected simulator. By analysing the simulator output, `Simulator Wrapper` computes a set of metrics and trade-offs between them. Using this information, it can rank the different mappings. Finally, it enhances the application and CP models with this information. `Simulator Wrapper` could also be used during the description of the application to evaluate different compositions of components but also the impact of different functional and non-functional goals on the metrics. Doing so will give finer insights of the predicted behaviour of the application to the Cloud user and/or the application developer.

Design

Figure 26 describes how `Simulator Wrapper` component works internally. First of all, based on the PAASAGE application and CP models but also using information retrieved from the MDDB, a simulator view of the application is generated. It contains all the information required to simulate an application:

- Application's components and their compositions.
- A (set of) mapping(s) between the components and the Cloud resources.
- Synthetic or predicted workloads, *i.e.* the different request types and their characteristics (request arrival, inter-request time, resource consumption for each component, etc.).
- Performance model of the application and its components and connectors.

Figure 27 shows the internal representation of an application. An example is presented in Section Example on page 63.

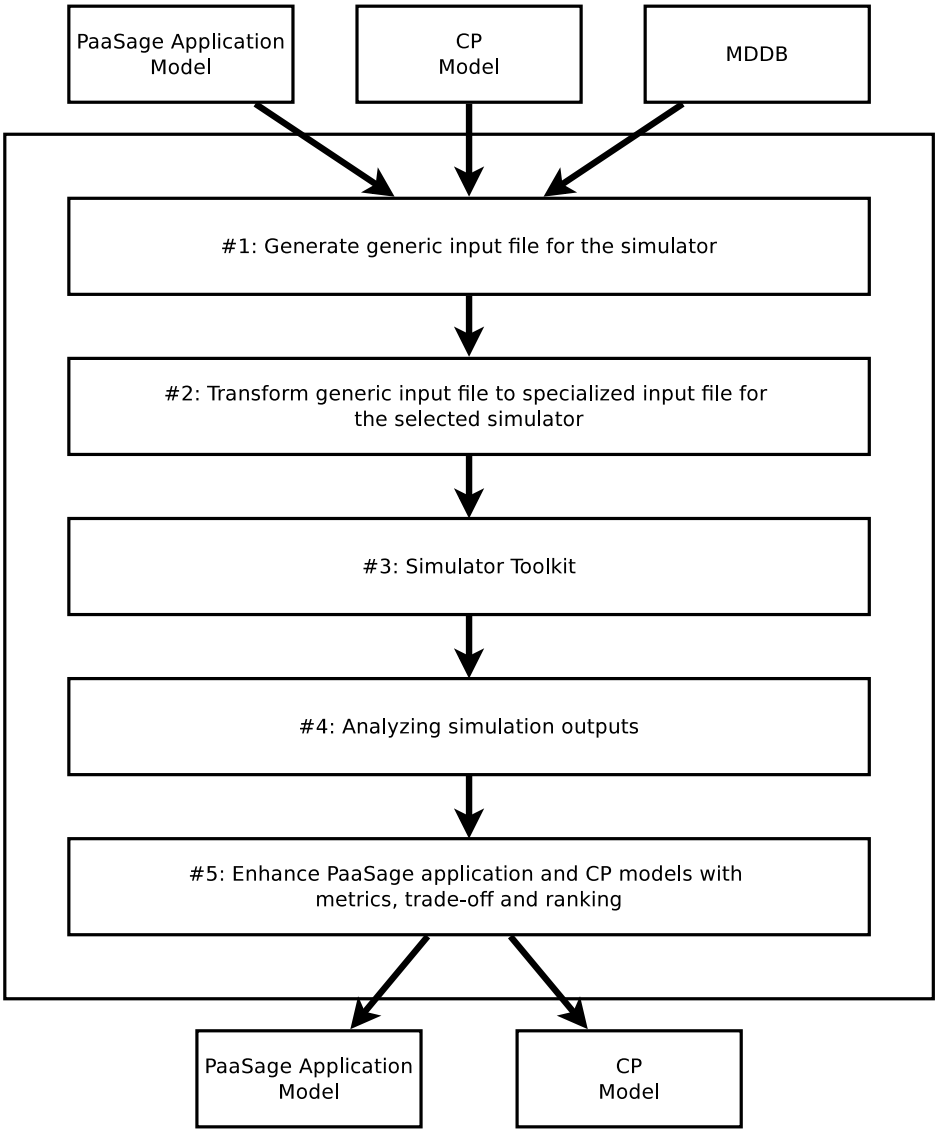

Figure 26: Internal architecture of the Simulator Wrapper.

Furthermore, it is also required to generate an internal representation of the considered platform. Based on information extracted from the MDDB, the infrastructure topology (network, physical machines, storage) of the platform and performance models for each of its resources are generated. Furthermore, using the MDDB and the PaaSage application model, the cost model of the platform is also generated, *i.e.* the Cloud offers (VM types, storage solutions, prices, etc.).

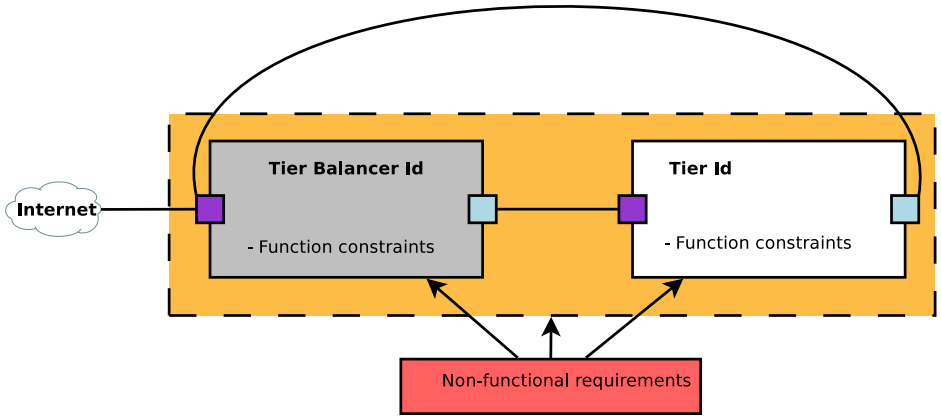

Figure 27: Generic application model.

By modifying the next layer (layer #2 in Figure 26), it is possible to support any simulator toolkit. Nonetheless, for the moment, we focus on the integration of the SimGrid Cloud Broker (SGCB) Cloud simulator toolkit [49]. Based on the internal representation of the platform and the application, the specific input for the simulator are generated. First of all, the application model is used to generate the code that will be used to simulate the application. This code also contains the performance model of each components and connectors. In our case, the EMF model is transformed into Java SGCB code.

Moreover, based on the workloads, the code for each request types and their characteristics is generated. A generic model for n-Tier application requests within SGCB is given in Figure 28. An example is presented in Section Example on page 63. This code describes the data flow between components and the consumption of resources on each components. Finally, it generates a (set of) `XML` configuration file(s) for the simulator, *i.e.* one per mapping between components and Cloud resources. An example of such file is given in Listing 2.

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

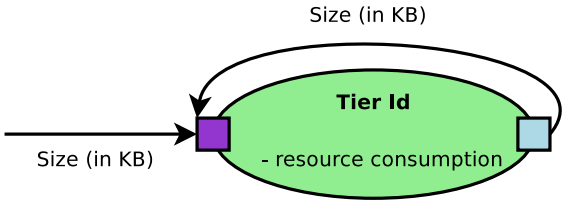

Figure 28: Generic request's dataflow model.

Listing 2: 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>
```

Listing 3: 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>
```

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 3. 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 4. 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 4: 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>
[...]
```

The simulator is invoked (layer #3) for each configuration file, *i.e.* for each component to resource mapping.

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[18]. The traces are converted to the CSV format to be 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. 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. All the traces are merged and analysed through a set of R scripts[19]. The R scripts generate a set of metrics (Throughput, RTT, Cost). Using these metrics, `Simulator Wrapper` also produces trade-off curves between the different metrics. An example of the performance analysis process is given in Section Example on page 63. Finally, based on this trade-off, the layer #4 of the

[18] `https://github.com/schnorr/pajeng`
[19] `http://www.r-project.org/`

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
```

`Simulator Wrapper` component ranks the different component to resource mappings.

Using the information generated by the layer #4, the PaaSage application and CP models are enhanced to contain the different performance metrics, trade-off between them and the ranking number of each individual mapping. These two enhanced models are the output of the `Simulator Wrapper` component.

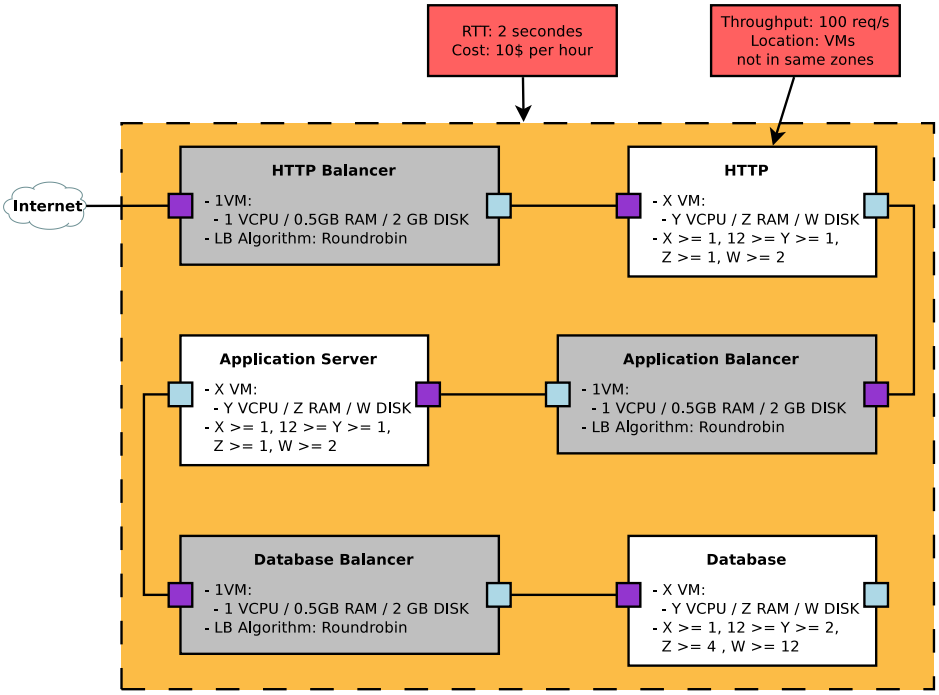

Figure 29: Generic application model for the RUBBoS application.

Implementation

At the moment, SGCB is available for the PAASAGE community[20] but also its extension[21] for n-Tier applications. The combination of these 2 software allows to simulate most of the use-cases selected in the PAASAGE project. Consequently, the layer #3 is ready. The layer #1 and #2 are not available yet but part of it is, *e.g.* the configuration file generator. The pre-analysing (paje traces to CSV) and the analysis itself (R scripts) are available. But the layer #5 is not available yet.

Example on a 3-tier application

This section presents a small example of the simulator wrapper for a 3-tier application. It uses RUBBoS[22] as test application, that is composed of an HTTP front-end, an application server back-end and a database. Moreover, for each

[20] `https://gforge.inria.fr/projects/sgcb/`
[21] `https://gforge.inria.fr/projects/sgcbntier/`
[22] `http://jmob.ow2.org/rubbos.html`

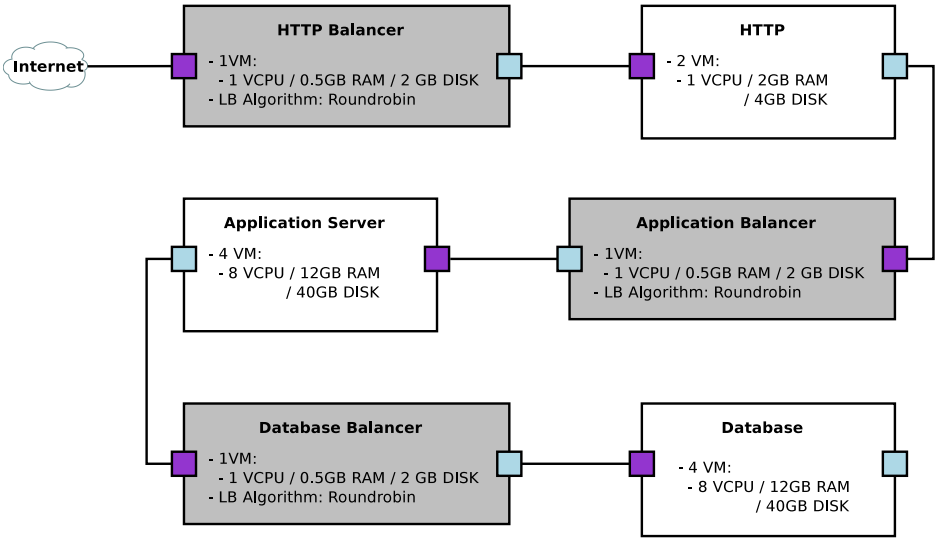

Figure 30: Example of an instance of the RUBBoS application model.

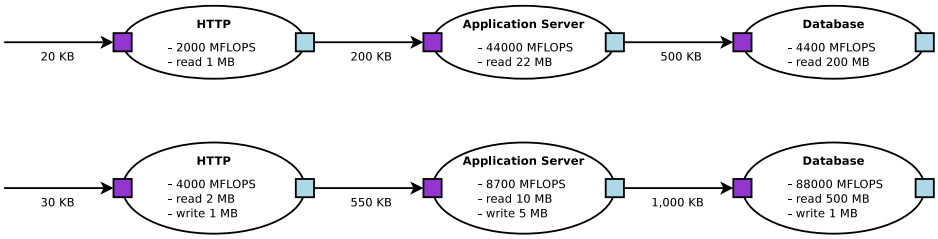

Figure 31: 2 request types' dataflow for the RUBBoS application.

tier, there is a tier load balancer that spreads the incoming requests between the different instances of each components. Figure 29 presents a generic description of the application. Furthermore, Figure 30 presents an instance of this generic description where the amount of resources for each components has been fixed. Accordingly, multiple instances of the generic description must be generated, *i.e.* one for each combination of resources.

However, it is also required to generate the request data-flow and their resource consumption. Figure 31 presents the modelling as data-flow of two types of RUBBoS requests.

First, the simulator requests resources, *e.g.* VMs in our case. Then, it starts the components to the selected VMs. Finally, all the components are initialised and run using the following workflow:

- For each Tier Server

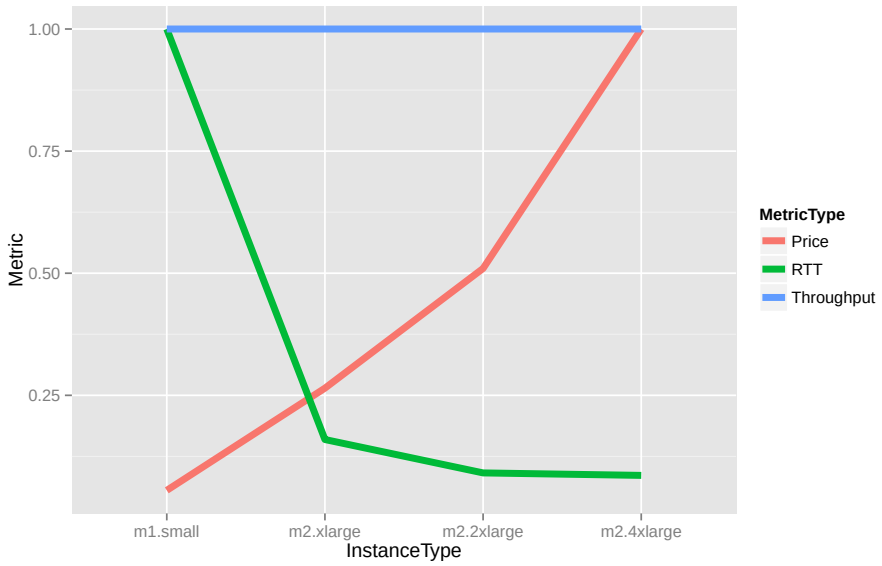

Figure 32: Trade-off between the metrics for the RUBBoS application and the horizontal scalability of the application tier.

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

- 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

Figure 32 shows the output of the performance analysis. The goal of this experiment was to evaluate the horizontal scalability of the application tier. Accordingly, we run a simulation for each VM type available. Then, we generate price, round-trip time (RTT) and throughput metrics. Finally, we produce Figure 32 that can be used to select the best trade-off by ranking the different mapping.

5.9 Solver-to-deployment

Overview

The *Reasoner* consumes the CPIM received as input by the Profiler (*cf.* Section 4) and produces a *Cloud Platform Specific Model* (CPSM) [2], that specifies the target deployment of applications. `Solver-to-deployer` 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 metamodels aim at enabling interactions between the *Profiler* and the *Reasoner* while lowering dependencies to CAMEL. Solvers produce solutions using these Upperware metamodels. The main objective of the `Solver-to-deployment` component is to translate the output of the Solvers into the Deployment Model CPSM. Figure 33 describes the various subcomponents that compose the *Solver-to-deployer* component.

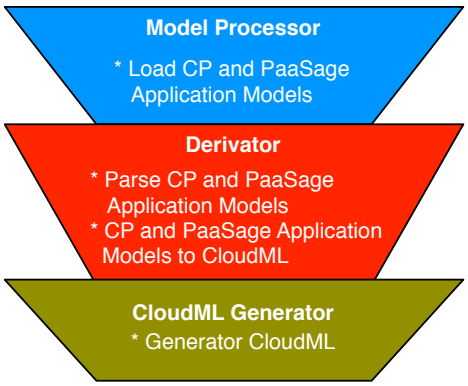

Figure 33: Solver-to-deployer-overview.

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 elements required to build the CPSM. The `Derivator` component is concerned with matching the CP and PaaSage Application Models of applications with the concepts in the CPSM. Finally, the

`CloudML Generator` component is concerned with serialising the *CPSM* models.

Matching Algorithm

The *Paasage Application Metamodel* (*cf.* Section 3.2) defines concepts to characterise an application to be deployed. This metamodel allows a solver to specify a target deployment model of the application. In particular, the following concepts can be expressed:

- *Providers:* It represents a cloud infrastructure provider. The *Provider* concept is defined with a specific position, *i.e.* continent, country or city.
- *Virtual Machines:* Virtual Machines (VM) are represented through two concepts in the metamodel, *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.
- *Application Components:* Application Components are represented through two concepts in the metamodel, the *ApplicationComponentProfiles* and *ApplicationComponent*. The former represents the application components in terms of name, version and application type. The latter represents concrete instances of the *ApplicationComponentProfiles*.

On the other side, *CloudML* allows expressing the following concepts: Resource, Providers, Component, Communication, and containment.

- *Resource:* A Resource represents an artefact (e.g., scripts, binaries, configuration files, etc.) adopted to manage the deployment life-cycle (e.g., download, configure, install, and start, stop).
- *Provider:* A Provider represents a collection of virtual machines on a particular cloud provider with a specific position, *i.e.* continent, country or city.
- *Component:* A Component represents a reusable type of component of a cloud-based application. A Component can be an ExternalComponent, meaning that it is managed by an external Provider, or an InternalComponent, meaning that it is managed by the PaaSage platform. An ExternalComponent can be a VM and an InternalComponent can be an application component.

Paasage App. Metamodel Concepts	CloudML Concepts
Provider	*Provider*
VirtualMachineProfiles	*ExternalComponent*
VirtualMachines	*VMInstance*
ApplicationComponentProfiles	*InternalComponent*
ApplicationComponent	*ComponentInstance*

Table 1: Matching table.

- *Communication:* Communication represents a reusable type of communication binding between Components, ExternalComponent or InternalComponent.
- *Containment:* A Containment represents a reusable type of containment binding between Required- and a ProvidedContainmentPort. A Containment can be associated to Resources specifying how to configure the components so that the contained component can be deployed on the container component.

The correspondence between the two models (`Paasage Application` and `CloudML` models) is performed using the matching in Table 1. When parsing the loaded models of the target deployment of the application, a corresponding `CloudML` instance is created for every `Paasage Application Metamodel` instance.

Implementation

The current implementation of the `Solver-to-deployment` has a `Model Processor` for the *Paasage Application and CP Metamodels* used for the input description. This `Model Processor` exploits the *Eclipse Modeling Framework*[23] (EMF) to load an object representation of the target CP and Application models. The `Derivator` component for *PaaSage Application* and *CP Models* fills the *CPSM* with information related to virtual machine profiles and their related Cloud providers as well as to the different elements that compose the application. The *CloudML* Generator is based on the *CloudML* library[24].

[23]EMF: `http://www.eclipse.org/modeling/emf/`
[24]The CloudML library: `https://github.com/SINTEF-9012/cloudml`

6 Adapter

The purpose of the *Adapter* is twofold: (1) It transforms the currently running application configuration into the target configuration in an efficient and safe way, and (2) it implements high-level management policies involving multiple clouds. The *Adapter* is composed of three components, the `Adaptation Manager`, the `Plan Generator`, and the `Application Controller`, as shown in Figure 34. These three components are discussed in the following sections.

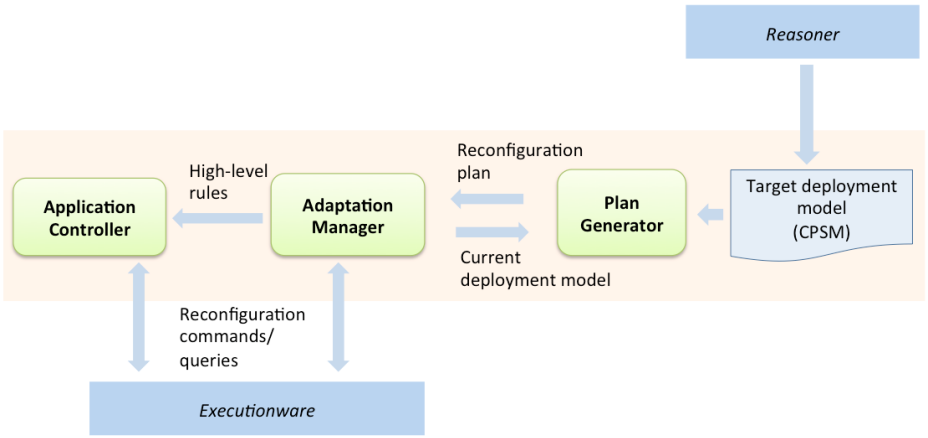

Figure 34: Adapter Architecture.

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 provide high-level rules, and with the *Executionware* to collect information and to execute reconfiguration actions.

Implementation

The Adaptation Manager is decomposed into four main components shown in Figure 35. The `Reasoner Interfacer` loads the Cloud Platform Specific

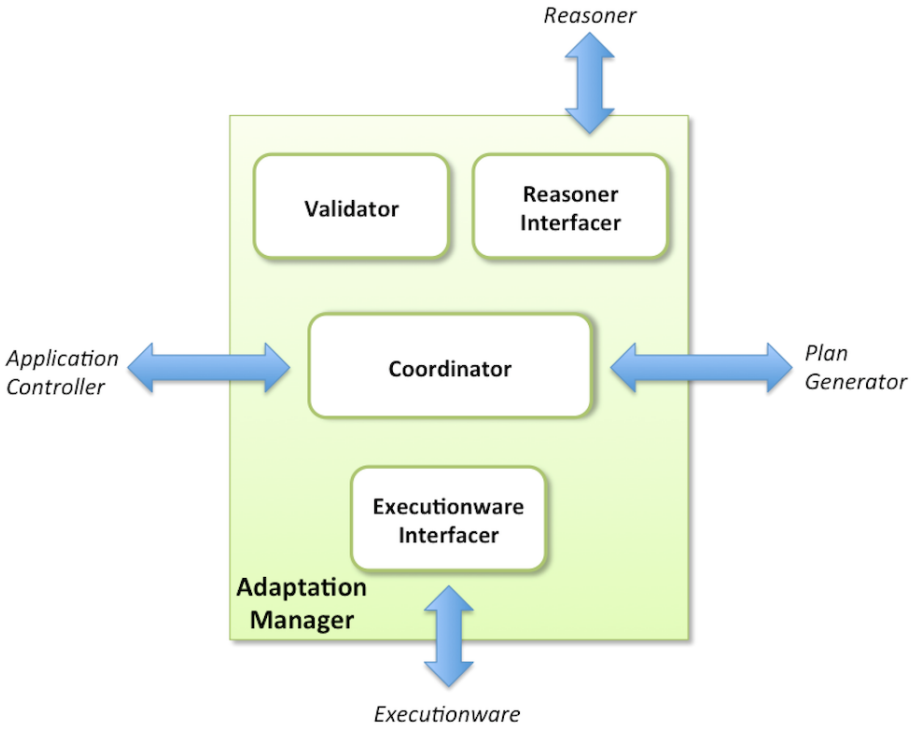

Figure 35: Adaptation Manager structure.

Model (CPSM) 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` is a pluggable component that decides whether a reconfiguration plan is allowed. `Validator` decisions are based on analysing the reconfiguration benefits (*e.g.* increases in application performance) and costs (*e.g.* disruption of application operation, risk for reconfiguration failures). This analysis uses information about the current system state, past application executions as well as past application reconfigurations. Finally, the `Coordinator` directs the other components and maintains a CPSM model that reflects the state of the current system.

The following workflow illustrates the reconfiguration process.

1. The `Reasoner Interfacer` loads 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` extracts from the plan any rules that involve multiple clouds and passes them to the `Adaptation Controller`.

5. The `Coordinator` invokes the `ExecutionWare Interfacer` to apply the actions in the plan and to verify their correct execution.

6. The `Coordinator` updates the current deployment model.

At Step 3, if validation fails, the `Coordinator` obtains a new target deployment model through the `Reasoner Interfacer`. At Step 5, the `Coordinator` must handle failures and timeouts, a challenging but common problem in distributed systems. The simple approach that is initially adopted is to discontinue the adaptation and ask the `Plan Generator` for a new reconfiguration plan starting from the updated state.

The M18 prototype of the `Adaptation Manager` focuses on initial deployment rather than full-scale dynamic reconfiguration. It runs as a JAVA process that uses the *CloudML* library[25] and the Jersey REST library. The `Validator` component is currently based on performing simple syntactical checks.

Upperware-Executionware REST API

Interactions between the *Upperware* and the *Executionware* rely on a REST API. The API is based on the main concepts of CloudML, 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 an HTTP POST request with a JSON representation corresponding to the CPSM model and including information on components, component instances, relationships, and scalability rules. Successful deployment produces a set of linked resources that enables the client to monitor and manipulate the application through HTTP requests.

Table 2 provides an overview of the API, which is currently being elaborated in collaboration with WP5. Listing 6 presents an excerpt of the HTTP POST request for deploying the Simple Application.

[25]The CloudML library: `https://github.com/SINTEF-9012/cloudml`

Table 2: REST API.

Operations	**Description**
GET applications POST applications GET applications/{appId} DELETE applications/{appId}	Query, deploy, undeploy applications
GET applications/{appId}/components POST applications/{appId}/components GET components/{compId} DELETE components/{compId} POST components/{compId}	Query, deploy, undeploy, perform actions on components (*e.g.* start, stop instances)
POST relationships DELETE relationships/{relId} GET relationships/{relId}	Create, destroy, query relationships (communication or containment)
GET components/{compId}/instances POST components/{compId}/instances DELETE instances/{instanceId} GET instances/{instanceId} POST instances/{instanceId}	Query, add, delete, query, perform actions on component instances (*e.g.* start, stop)
GET virtualMachines/{vmId}	Query virtual machines

Listing 6: Example of HTTP POST request for the Simple Application.

```
POST /applications HTTP/1.1
Content-Type: application/json

{
    "name": "simpleApp",
    "description": "My simple application",
    "components": [
        {
            "name": "simpleAppBundle",
            "lifecycle": {
                "init": "myinit.sh",
                "start": "myrun.sh",
                "stop": "...",
                [...]
            },
```

```
            "requiredVM": {
                "provider": "AmazonEC2",
                "location": "EU",
                "minRam": 4096,
                "minCores": 4,
                "os": "ubuntu",
                [...]
            },
            [...]
        }
    ]
}
```

6.2 Plan Generator

Overview

The `Plan Generator` component generates a reconfiguration plan for modifying the configuration of the running application. The inputs of `Plan Generator` are two CloudML deployment models, namely `current` and `target`. The `current` model describes the actual system deployment at the time when the comparison is launched, and the `target` model represents the deployment desired by the *Reasoner*. The output is a list of actions representing the required changes from the current to the target model. The types of potential actions are listed in Table 3.

Implementation

The comparison process follows the order from virtual machine, application component, to communications, according to the logical dependencies between these concepts.

The `Plan Generator` first compares the set of virtual machine instances from the current model, and the set of virtual machine instances in the target. It compares every pair of virtual machine instances from the Cartesian product of these two sets, and identifies the matched paired based on the virtual machine names. For each un-matched virtual machine in the current set, it creates a `removeVM` action with this VM as the argument, and for each un-matched VM in the target set, it creates a `addVM` for it.

After finishing the virtual machine comparison, `Plan Generator` compares the application component instances on each VM. On each pair of matched virtual machine, it extracts the set of components deployed on the virtual machine in the current model, and also the set of components on the VM in the target model. The comparison of these two sets are similar to the VM comparison: `Plan Generator` identifies the matching pairs of components, generates `addComponent` actions for the unmatched components in the target set,

Table 3: Action types output by `Plan Generator`.

Action	Parameter	Effect
addVM	VM	deploy a new virtual machine
removeVM	VM	terminate a virtual machine
addComponent	Component, VM	deploy the application component on the target virtual machine
removeComponent	Component	remove the application component instance from its current host
addCommunication	Communication	create a communication instance
setRequired	Communication, Port	set and configure the required port of the communication
setProvided	Communication, Port	set and configure the provided port of the communication
removeCommunication	Communication	remove the communication

and generates `removeComponent` actions for the unmatched ones in the current set. To decide if a pair of application components are matched, it utilises both the names of application components, and also their types. Beside the comparison of components on the matched pair of virtual machines, `Plan Generator` also creates `addComponent` actions for all the components on the newly added virtual machine instances, and similarly creates `removeComponent` actions for all the components on the removed virtual machine instances.

`Plan Generator` compares the communications in the last step. It also firsts identifies the matched pairs of communications between current and target models, based on the communication name and type. For the matched pairs, it goes on to compare the required and provided ports, to generate the `setRequired` and `setProvided` actions. For the un-matched ports, it creates the `addCommunication`, `removeCommunication` actions, as well as the corresponding actions on their port.

It is worth noting that at this stage, `Plan Generator` always gives a higher priority to the target model, which means that, for example, any virtual machine instance in the target model that does exist in the current one will be

regarded as one that needs to be created, and any virtual machine that does not exist in the target model any more will be removed. This works well when there is no change on the current system during the time of reasoning, otherwise the system changes happened during the reasoning time will be flushed. This is kept as future work.

6.3 Application Controller

Overview

The `Application Controller` component implements high-level management policies that need global knowledge or involve multiple clouds. An example is a policy that bursts an application from a private to a public cloud in response to increased demand. Another example is a policy that migrates an application across public clouds in response to relative price or availability variations. Such policies cannot be enforced by the *Executionware*, which only supports low-level scalability operations on a per-component and per-cloud basis.

Implementation

`Application Controller` receives as input a set of high-level rules and is responsible for enforcing them. The rules are expressed in the DSL for scalability rules and are based on the Event-Condition-Action format. Events may originate from monitored or historical information and actions are commands of the *Upperware-Executionware* REST API. One possible action taken by `Application Controller` is to inform *Application Manager* that a new deployment model is needed. This enables the implementation of rules that trigger a change in the deployment model if the system behaviour moves outside acceptable bounds (*e.g.* maximum response time).

The implementation of the `Application Controller` component is out of scope for the M18 prototype. The plan is to use a rule engine that supports the scalability rules DSL, which is being defined in Task 2.3. This engine will be integrated with components that interface with the *Executionware* and *Metadata Database* in order to collect the necessary information and to send reconfiguration commands.

7 Conclusion

The Prototype Upperware is made of three major components: the Profiler, the Adapter, and the Adapter. Interactions with other PAASAGE elements (CAMEL, MDDB, Executionware) have been well identified and the proposed Upperware prototype architecture tries to localise 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. These are the sub-components that would have to be adapted in case of evolution of CAMEL.

The Reasoner currently supports several kinds of solvers, so as to let us evaluate various strategies to efficiently compute a deployment. 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 exiting solvers (CP Solver and MILP Solver), developing some simple heuristics as proof of concepts, and developing a Meta Solver for anticipated complex situations. The Reasoner also contains several helper components to evaluate solutions. Currently, it provides three mechanisms: a utility function evaluation, an access to historical data stored in the MDDB, and a cloud simulator wrapper of SimGrid.

The Adapter is made of three sub-components, only two being integrated into the M18 prototype. The Plan Generator transforms the target deployment computed by the Reasoner in a set of actions. The Adaptation Manager is responsible for driving the (re)configuration process, and in particular interacting with the Reasoner and the Executionware.

This document describes our views of the Upperware Prototype at M18. During the integration phase, as well as when gaining experience by actually supporting more and more use cases, the Upperware architecture may evolve. Therefore, interactions with other PAASAGE research partners (in particular from other work-package) may have some impacts on well identified sub-components. Interactions with the industrial 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. The M36 deliverable will describe the final version of the Upperware, *i.e.* the Product Upperware.

References

[1] 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*. 2013.

[2] Alessandro Rossini, Arnor Solberg, Daniel Romero, Jörg Domaschka, Kostas Magoutis, Nicolas Ferry and Tom Kirkham. *Deliverable D2.1.1 - CloudML Guide and Assessment Report*. 2013.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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).

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

[9] 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).

[10] 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+%26+forthcoming+titles+%28default%29/book/978-3-540-71843-7.

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

[12] 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).

[13] 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).

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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, 235–245. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804 (visited on 25/03/2014).

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] Maciej Malawski, Kamil Figiela and Jarek Nabrzyski. “Cost minimization for computational applications on hybrid cloud infrastructures”. In: *Future Generation Computer Systems* 29.7 (2013), pp. 1786 –1794. ISSN: 0167-739X. DOI: http://dx.doi.org/10.1016/j.future.2013.01.004.

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

[31] 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.

[32] 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).

[33] 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).

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

[35] 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.

[36] 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/.

[37] 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.

[38] 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.

[39] Kurt Geihs et al. "A comprehensive solution for application-level adaptation". In: *Software: Practice and Experience* 39.4 (2009), 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).

[40] 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).

[41] 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.

[42] 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).

[43] 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).

[44] 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, 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).

[45] 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).

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

[47] 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).

[48] 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.

[49] 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.

A Common Metamodels

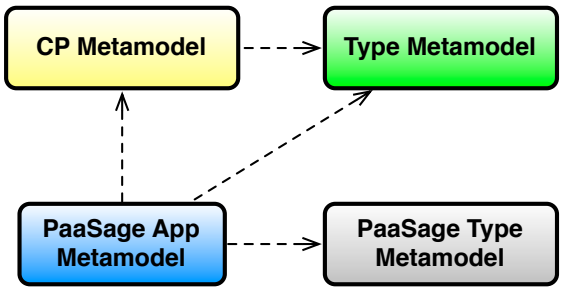

Figure 36: Metamodels overview.

Figure 36 provides an overview of the Upperware metamodels and their relationships.

The *Constraint Problem Metamodel* (CP Metamodel) and *Types Metamodel* enable the definition of the Cloud provider selection problem as a constraint problem. They are fully describes on Page 84.

The *PaaSage Application Metamodel* (PaaSage App Metamodel) and *PaaSage Type Metamodel* establish the relationship between concepts from the Cloud and constraint problem worlds. They are fully described on Page 85.

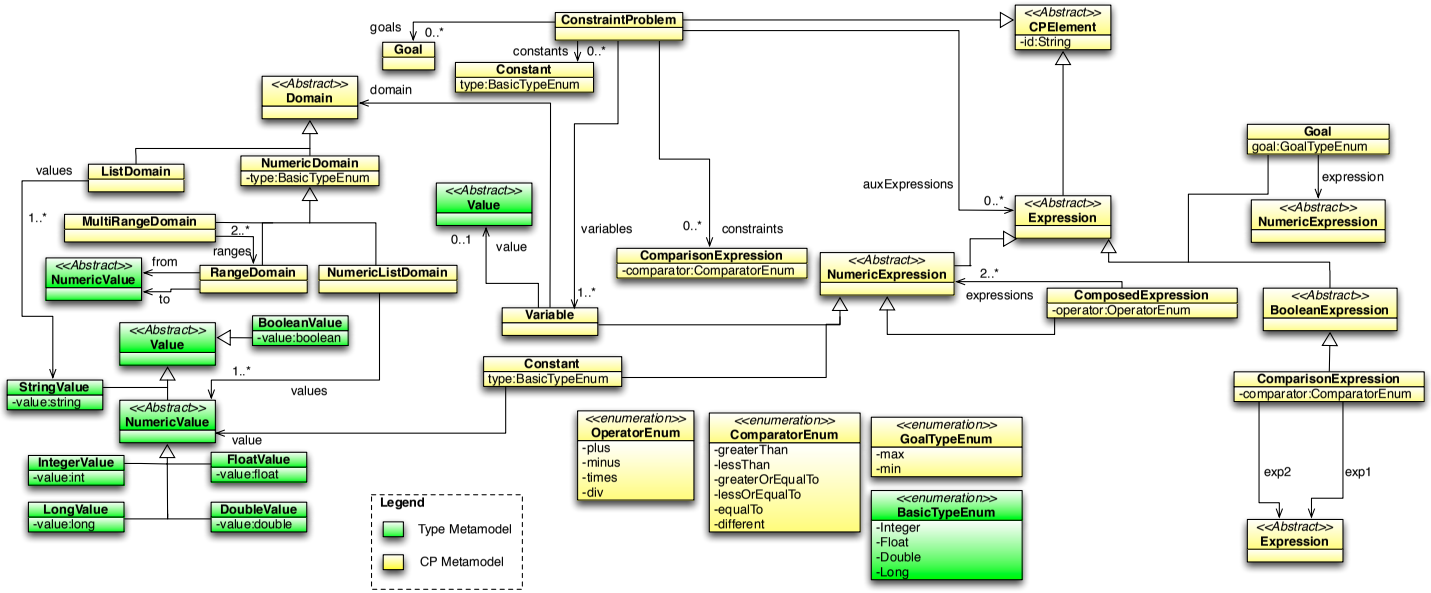

Figure 37: CP and Type Metamodels.

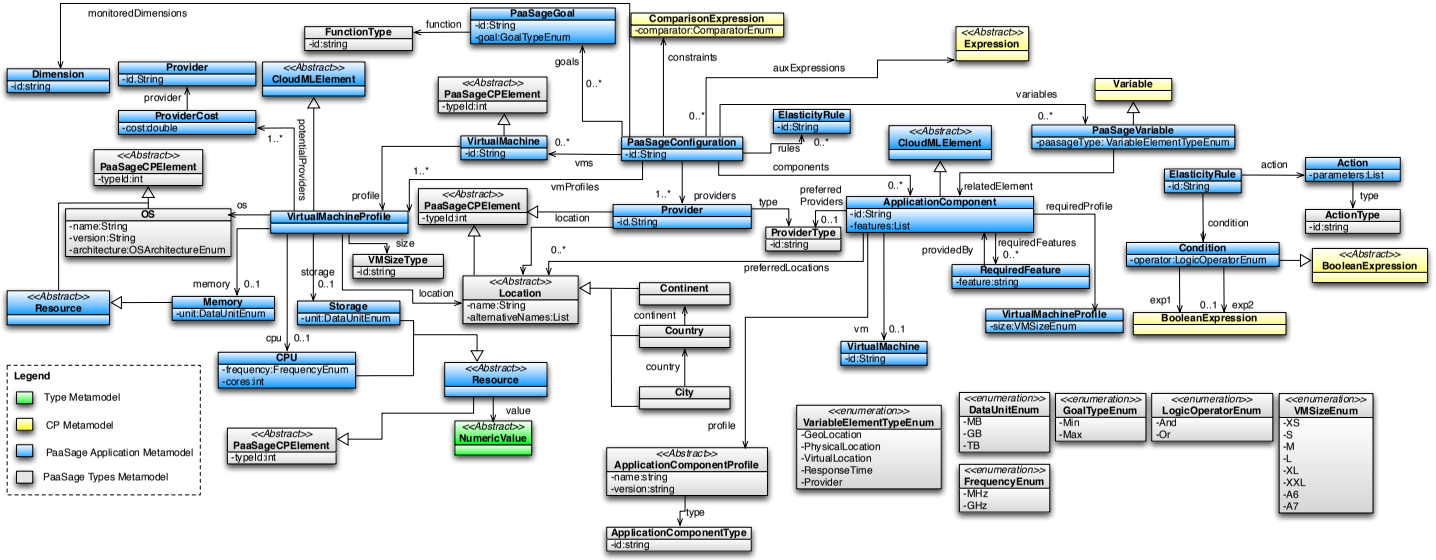

Figure 38: PaaSage Type and Application Metamodel.

B CPIM of Simple Application Example

```
<?xml version="1.0" encoding="UTF-8"?>
<net.cloudml:CloudMLModel xmi:version="2.0" xmlns:xmi="http://www.omg.
    org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:net.cloudml="http://cloudml.net" name="SimpleAppModel" xsi:
    schemaLocation="http://cloudml.net ../../metamodel/CloudML-2.0.
    ecore">
  <providers name="AmazonEC2" credentials="./credentials_aws"/>
  <providers name="ElasticHosts" credentials="./credentials_eh"/>
  <components xsi:type="net.cloudml:VM" name="ML" provider="AmazonEC2"
       location="EU" minRam="4096" minCores="4" minStorage="102400" os
      ="ubuntu" securityGroup="simpleApp" sshKey="simpleApp" privateKey
      ="/simpleapp.pem" groupName="simpleApp">
    <providedContainmentPorts name="mlProvided" owner="ML"/>
  </components>
  <components xsi:type="net.cloudml:VM" name="MW" provider="
      ElasticHosts" location="EU" minRam="4096" minCores="4" minStorage
      ="102400" os="windows" securityGroup="simpleApp" sshKey="
      simpleApp" privateKey="/simpleapp.pem" groupName="simpleApp">
    <providedContainmentPorts name="mwProvided" owner="MW"/>
  </components>
  <components xsi:type="net.cloudml:InternalComponent" name="SimpleApp
      ">
    <requiredContainmentPort name="tomcatRequired" owner="SimpleApp"/>
  </components>
  <components xsi:type="net.cloudml:InternalComponent" name="tomcat">
    <providedContainmentPorts name="tomcatProvided" owner="tomcat"/>
    <requiredContainmentPort name="mRequired" owner="tomcat"/>
  </components>
</net.cloudml:CloudMLModel>
```

C Saloon Ontology

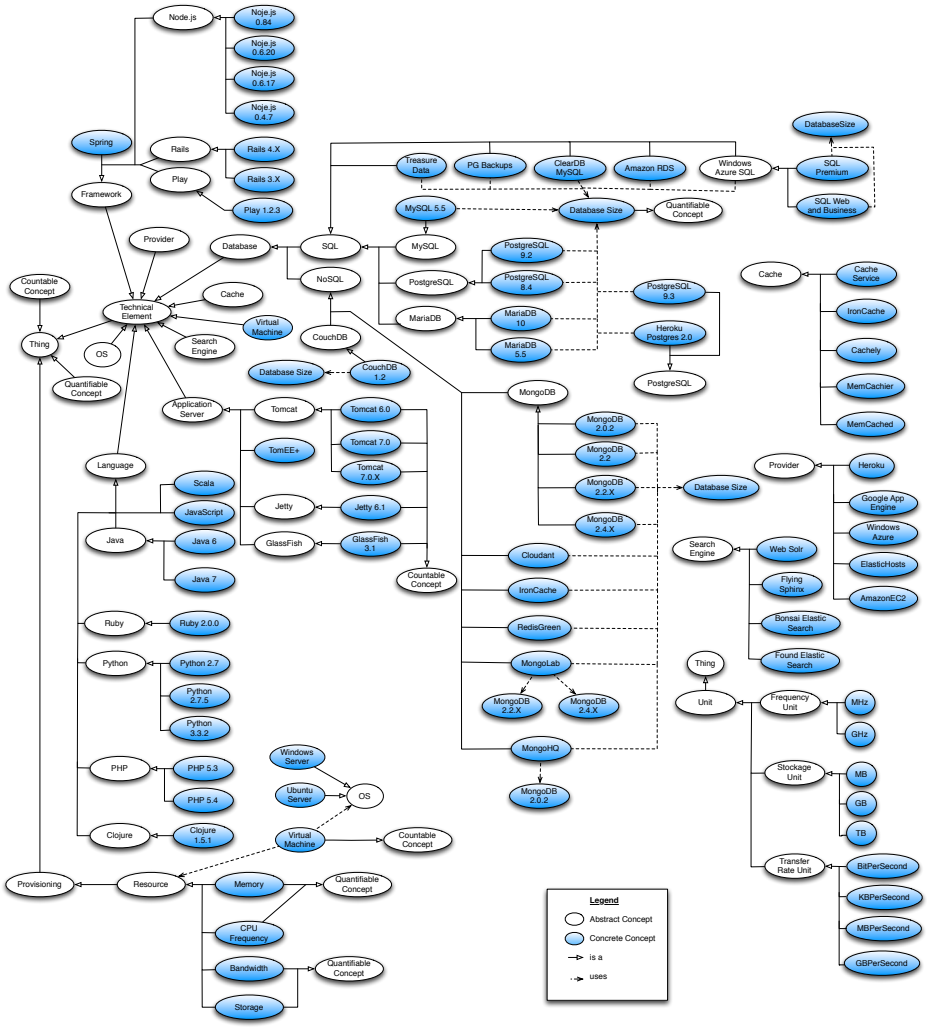

Figure 39: Saloon Ontology.