SALOON: A Platform for Selecting and Configuring Cloud Environments

Clément Quinton, Daniel Romero, Laurence Duchien

INRIA Lille – Nord Europe & University Lille 1, LIFL UMR CNRS 8022, France

SUMMARY

Migrating legacy systems or deploying a new application to a cloud environment has recently become very trendy, since the number of cloud providers available is still increasing. These cloud environments provide a wide range of resources at different levels of functionality, which must be appropriately configured by stakeholders for the application to run properly. Handling this variability during the configuration and deployment stages is known as a complex and error-prone process, usually made in an *ad hoc* manner. In this paper, we propose SALOON, a *Software Product Lines*-based platform to face these issues. We describe the architecture of the SALOON platform, which relies on feature models combined with a domain model used to select among cloud environments a well-suited one. SALOON supports stakeholders while configuring the selected cloud environment in a consistent way, and automates the deployment of such configurations through the generation of executable configuration scripts. This paper also reports on some experiments showing that using SALOON significantly reduces time to configure a cloud environment compared to a manual approach and provides a reliable way to find a correct and suitable configuration. Moreover, our empirical evaluation shows that our approach is effective and scalable to properly deal with a significant number of cloud environments.

Received . . .

KEY WORDS: Software Product Line, Cloud Computing, Feature Modeling, Model Driven Engineering

1. INTRODUCTION

Cloud computing has recently emerged as a major trend in distributed computing, and deploying an application to a cloud environment has become very trendy, since the number of cloud provider offers is still increasing. In the cloud computing paradigm, computing resources are delivered as services. Such a model is usually described as *Anything as a Service* (XaaS or *aaS), where *anything* is divided into layers from *Infrastructure* to *Software* including *Platform* [1, 2]. At the IaaS level, the entire software stack running inside the virtual machine must be configured as well as the infrastructure concerns: number of virtual machines, amount of resources, number of nodes, SSH access or database configuration. Regarding platforms provided by PaaS environments, the configuration part only focuses on software that compose this platform: which database(s), application server(s), compilation tool or libraries. The software stack management process is entirely handled by the PaaS provider. Furthermore, IaaS and PaaS define different deployment modes (*e.g.*, public, private and hybrid) and price rates to be considered when selecting an environment. This layered model therefore offers many configuration and dimension choices, for the application to be deployed as well as the configurable runtime environments [3]. Thus,

*Correspondence to: clement.quinton@inria.fr

Contract/grant sponsor: EU; contract/grant number: FP7 IP PaaSage project.

when deploying an application to the cloud, companies or developers have to cope with clouds variability due to a wide range of resources at different levels of functionality among available cloud environments. Dealing with this variability leads to complex and error-prone configuration choices that are usually made in an *ad hoc* manner.

In this article, we propose SALOON, a platform for selecting and configuring cloud environments, an improved version of our work presented in a previous paper [4]. The main contribution of SALOON is to address the above issues in an automated way, taking as input the applications *functional* and *non-functional* requirements [5]. In particular, SALOON supports the selection and configuration of cloud environments which are well-suited to handle these requirements. This is achieved through the use of *Software Product Lines* (SPLs) [6, 7]. SPLs are dedicated to the configuration of software products with high variability using variability models, in particular *Feature Models* (FMs) [8]. In SALOON, FMs are extended with cardinalities [9, 10, 11] and attributes [11, 12, 13]. These extensions are required when modeling cloud environments, to define the quantification of cloud resources together with dependencies among them. Although SALOON is not the first approach to use cardinalities and attributes as FM extensions, it provides additional support for the management of complex constraints involving attributes and cardinalities required when modeling cloud environments.

The remainder of this paper is organized as follows. In SEC. 2, we identify the key challenges in cloud environment selection and configuration that motivate this work. SEC. 3 introduces the background information about software product lines and feature modeling. We then present in SEC. 4 SALOON, our SPL-based platform. SEC. 5 reports on the implementation of the platform and describes the evaluation we lead to assess our approach. Finally, we discuss in SEC. 6 close-related work while SEC. 7 concludes the paper and gives some perspectives for future work.

2. KEY CHALLENGES

When deploying an application to the cloud, developers have to cope with a wide range of configurable resources among available cloud environments. Our goal, by proposing SALOON, is to provide a support to deal with this variability and to help those developers selecting and configuring a suitable cloud environment. To achieve these objectives, we identify the following challenges SALOON has to help developers to deal with:

$\mathcal{C}_1$: *Selecting a suitable environment.* Among the plethora of cloud providers, developers have to *(i)* find the ones that provide all functionalities required by the application to run properly, *e.g.*, the correct type of application server or database, and *(ii)* select one that is suitable regarding non-functional requirements for these functionalities, *e.g.*, the less expensive solution with at least 4 GB of RAM. The first challenge is therefore to provide a support to help the developer make such a selection.

$\mathcal{C}_2$: *Defining a proper configuration.* Dealing with clouds variability leads to complex and error-prone configuration choices that are usually made in an *ad hoc* manner. Moreover, developers' knowledge is not exhaustive and the way a cloud environment is configured can lead to inconsistencies between cloud services when running the application. The second challenge is thus to provide a mean to find a valid cloud configuration with respect to the required functionalities.

$\mathcal{C}_3$: *Deploying in a reliable way.* Once a cloud environment is selected and there exists a configuration for this environment that suits the required functionalities, developers have to avoid errors in the configuration process, in particular when defining cloud environment configuration files and scripts, to ensure the cloud environment to be properly configured. The third challenge is thus to provide a reliable support to handle such cloud configurations.

3. BACKGROUND

In this section, we present a brief introduction to the main elements the SALOON platform relies on, *i.e.*, Software Product Lines engineering and Feature Modeling.

SPL engineering aims at building software while ensuring quality, reliability and reduction of cost, efforts and time-to-market. It begins with the description, management and implementation of the commonalities and variabilities existing among the members of the same family of software products [6, 7]. The building process relies on the definition and the composition of a set of software artifacts, *e.g.*, piece of code, model, component or aspect, defined as *assets*. Some of these assets are mandatory and will be part of all the built software (commonalities), while other assets define the way software differs from each other (variabilities). The definition of variabilities and commonalities, known as variability modeling, is a central activity in SPL engineering and relies on variability models. A well-known approach to variability modeling is by means of *Feature Models* (FMs) introduced as part of *Feature Oriented Domain Analysis* (FODA) back in 1990 [8]. FMs are an abstraction to define software systems, where software artifacts are reified as features. FMs describe the way features are configured and reused to get a configuration that satisfies a set of defined constraints. The selection or deselection of features is known as the feature selection process, and a product is a valid combination of features. In these FMs, known as boolean FMs, a feature is either present or absent in the final product according to the configuration and the involved constraints. This valid product configuration is then given as input together with the related assets to a *derivation* tool that yields the software product, *e.g.*, generates code, merges model fragments or weaves aspects.

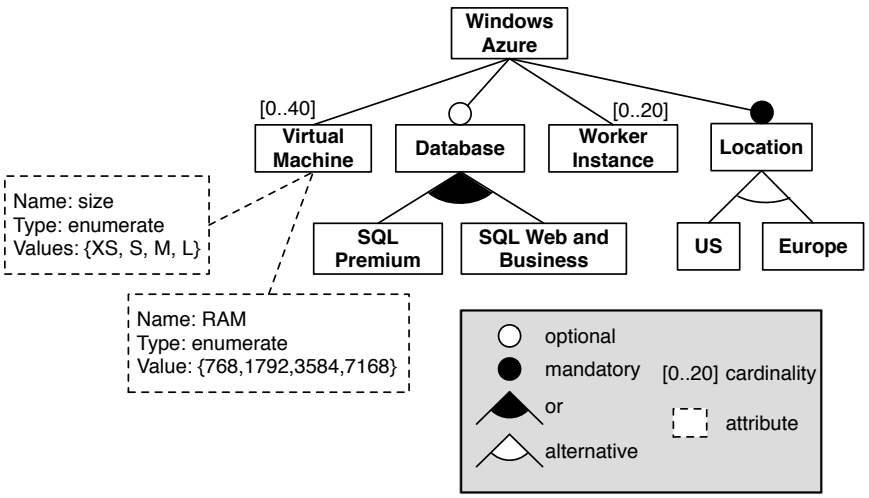


Figure 1. The Windows Azure Feature Model (excerpt)

In this paper, the FODA notation is extended with cardinalities and attributes. These extensions are required to define a cloud environment FM, *e.g.*, to express resources amount or available number of instances. A cardinality-based FM supports cardinalities [10, 11], first introduced as UML-like multiplicities [9], in extension to the original FODA notation. In this paper, we focus on feature cardinalities, in opposition to group cardinalities, used to define the amount of sub-features a parent feature may have. Thus, a feature cardinality specifies how many clones of a feature and its subtree can be included in a product configuration. A feature cardinality is defined as an interval [*m..n*] with *m* as lower bound and *n* as upper bound of this interval. FIG. 1 depicts such a situation, with an excerpt of the Windows Azure cloud environment FM [14]. Some configurations of this

FM may involve respectively up to 40 and 20 instances of the `Virtual Machine` and `Worker Instance` features.

On the other hand, attribute-based FMs (also known as *extended* FM*s*) are FMs whose additional information is added in terms of feature *attributes* [11, 12, 13]. Those attributes, mostly used to specify non-functional properties *e.g.*, a size or a quantity, can be either boolean, integer, real or an enumeration. In FIG. 1, the `Virtual Machine` feature holds an attribute named *size*, which enumerates the different sizes one can configure for a `Virtual Machine` instance.

4. THE SALOON PLATFORM

To tackle the challenges related to cloud selection and configuration definition issues described in SEC. 2, we propose the SALOON platform. FIG. 2 depicts an overview of the platform, which provides the following four contributions:

- (i) the description of cloud environment variability, *i.e.*, commonalities and variabilities, as FMs extended with cardinalities, attributes and constraints over them. One FM is used to describe one cloud environment, there is thus one software product line per cloud environment.
- (ii) the reification and gathering of cloud environments provided functionalities into a *Cloud Knowledge Model*, mapped to each cloud FM to automate the feature selection process.
- (iii) the configuration analysis of these FMs, including complex constraints over attributes and cardinalities, as well as the derivation of the related software products as configuration files and scripts, both processes being automated.
- (iv) an estimation of the cost for a given FM configuration, based on a cloud pricing model related to the FMs and a dedicated cost estimation engine.

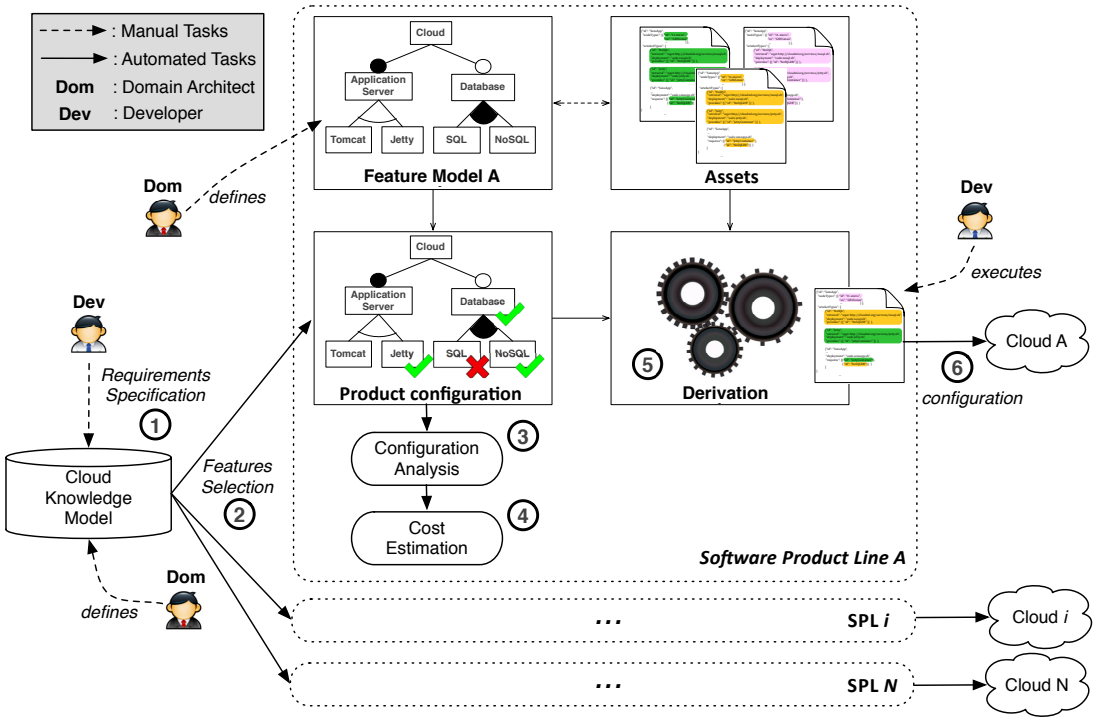


Figure 2. SALOON Platform Overview

Our approach distinguishes between two roles, *domain architects* and *developers*. The domain architect is expert in the particular domain targeted by the software product lines, the cloud computing one in SALOON. He/She is responsible for defining both the cloud feature models and the `Cloud Knowledge Model`. A domain architect has all information about commonalities and variabilities of one particular cloud environment, and thus defines the related cloud feature model. Then, all domain architects gather their knowledge to build the Cloud Knowledge Model, which is a reification of cloud services and functionalities provided by all cloud environments and defined in all SALOON feature models. The developer is the final user of SALOON. She/He interacts with the platform through the Cloud Knowledge Model, which is the entry-point of SALOON. Then, after several automated stages, she/he selects a cloud environment among suitable ones to automatically retrieve the related configuration files and scripts and executes them to configure the cloud environment.

Configuring a cloud environment using SALOON relies on several phases. First, the developer specifies her/his requirements using the Cloud Knowledge Model (FIG. 2 ①). Then, features and attributes of each FM are selected regarding the mapping between this cloud model and the FMs ②. Each FM `Configuration` is then checked ③ and given as input, if valid, to the `Cost Estimation` engine to help the user making her/his choice ④. Finally, the `Derivation` tool yields the related configuration files and/or deployment scripts ⑤, executed by the developer ⑥. We describe in details in the following sections the different concerns of the SALOON platform.

4.1. SALOON Core

Our recent work in the European PaaSage project [15] regarding deployment of cloud applications and configuration of cloud environments provides evidence that support for expressing constraints in terms of cardinality or attribute values and reasoning about these values has become necessary [16, 4]. SALOON Core provides such a support.

4.1.1. Abstract Syntax. SALOON Core enables the definition, the configuration and the analysis of FMs extended with cardinalities and attributes. In particular, it provides a support for new means of expression for feature modeling approaches regarding cardinality and attributes in constraints. FIG. 3 depicts the abstract syntax of the FMs supported by SALOON Core. Metaclasses drawn in dotted line are well-known in the variability modeling community, but may have different names. We refer to this part of the metamodel as FM_{MM}. This FM_{MM} metamodel remains valid for most feature modeling languages and tools that handle boolean FMs as well as cardinality-based FMs. More precisely for the latter, FM_{MM} is used in the literature [17, 10, 18] to define a graphical notation for features with cardinality. Even though the definition of cardinality is well-known, to the best of our knowledge, there is no available tool able to handle cardinalities properly during the reasoning and verification stages, in particular regarding constraints over them.

We extend this abstract syntax to support both modeling and configuration of FMs extended with cardinalities and attributes, depicted as solid line metaclasses in FIG. 3. Thus, we refer to the SALOON metamodel as SALOON_{MM}, where SALOON_{MM} = FM_{MM} + EXT_{MM}. These extensions can be plugged in any existing FM metamodel, *e.g.*, [17, 18], and relies on the `CardExConstraint` which allows variability modeler to define cardinalities and attributes for both features and constraints. A `CardExConstraint` is written $condition \rightarrow action$. For example, such a syntax provides a support to express the following constraints:

$$A' \quad \rightarrow \quad 3\ B \tag{1}$$

$$[3,6]\ A \quad \rightarrow \quad B.size \geq 2 \tag{2}$$

Constraints (1) and (2) illustrate the extensions the SALOON_{MM} provides for modeling FMs with cardinalities, attributes and constraints over them. Constraint (1) depicts an example of a `CardExConstraint` meaning that for each instance of `A` configured (*for each* is depicted by the

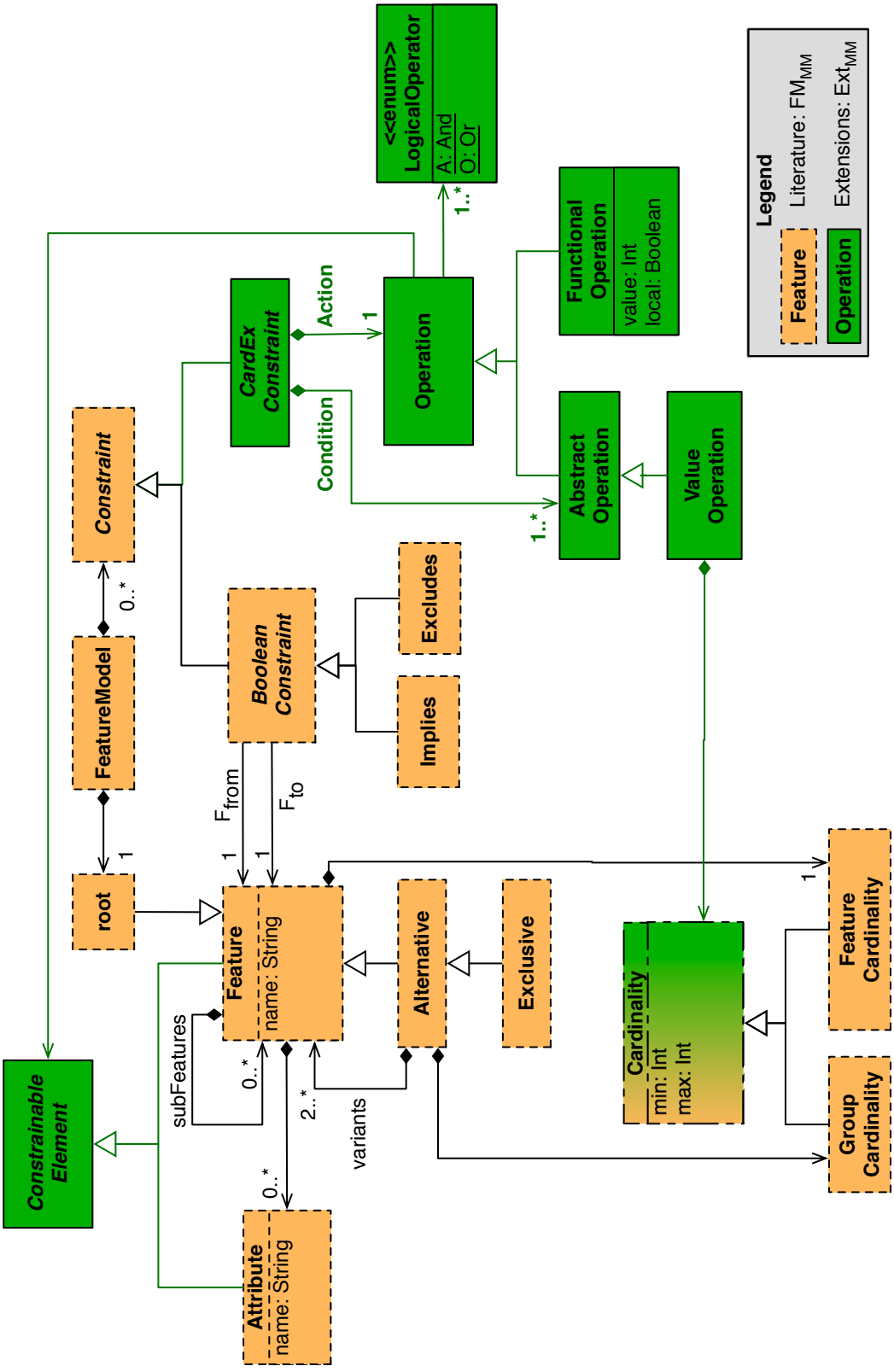


Figure 3. SALOON Metamodel

apostrophe), then there must be at least three more instances of `B` in the configuration. The condition of this constraint is set through an `AbstractOperation` to the `ConstrainableElement` feature A, while the action is a `FunctionalOperation`, whose *value* attribute is set to 3 and the *local* one to true. The *local* attribute is used to express that a constraint must hold for each instance of a feature or for the feature itself. If the same constraint is written without assigning the *local* attribute to true, then it means that three instances of `B` must be added in the configuration if `A` is selected, whatever the number of (potentially) configured instances. Constraint (2) describes that if there is at least three and at most six instances of `A`, then the value of the *size* attribute of feature `B` must be greater or equal than two. It relies on two `ValueOperations` as condition and action of the constraint. For both of them, the `Cardinality` meta-class is used to define the value range, *e.g.*, [3,6] for the condition and [2,*] for the action, where the "*" multiplicity is used to define that the given bound is not a fixed value and will not be taken into consideration.

Although we introduce a new notation to illustrate these examples, the contribution of the SALOON core is not to provide such a notation, but an extension of the existing expression means, to define constraints over attribute and cardinality values, required in particular when modeling cloud environments.

4.1.2. Semantics. The above-described cardinality and attribute-based expressions offer a support to precisely define how constraints are expressed in terms of feature instances and their value. We describe below a formal semantics for such constraints. Considering that:

- $\mathcal{M} = (\mathcal{F}, \varphi)$ is a FM extended with attributes and cardinalities, with $\mathcal{F}$ its non empty set of features and φ its set of constraints;
- $\omega : \mathcal{F} \rightarrow \mathbb{N} \times \mathbb{N}$ indicates the range of cardinality of each feature, *i.e.*, $\forall f \in \mathcal{F},\ \omega(f) = [n, m]$;
- $\texttt{card}$: $\mathcal{F} \rightarrow \mathbb{N}$ indicates the number of instances for a feature, *i.e.*, $\forall f \in \mathcal{F},\ \texttt{card}(f) = n$ with $n \in \mathbb{N}$;
- $\texttt{attr} : \mathcal{F} \rightarrow \mathcal{A}$ returns the set of attributes of f, *i.e.*, $\forall f \in \mathcal{F},\ \texttt{attr}(f) = \alpha$ with $\alpha \subseteq \mathcal{A}$, the set of all the attributes in $\mathcal{M}$;
- $\texttt{val} : \mathcal{A} \rightarrow \mathcal{E}$ returns the value of each attribute, *i.e.*, $\forall a \in \mathcal{A},\ \texttt{val}(a) = v$), with $v \in \mathcal{E}$, and $\mathcal{E} = \mathbb{R}$ or $\mathcal{E} =$ the set of strings.

A `CardExConstraint` constraint ρ is written

$$\textit{Condition} \rightarrow \textit{Constraint}$$

with

- *Condition* $\in \{f_{from}, c_{from}\ f_{from}, f_{from}.a_{from}\ comp\ v_{from}\}$;
- *Constraint* $\in \{f_{to}, c_{to}\ f_{to}, f_{to}.a_{to}\ comp\ v_{to}, n_{to}\ f_{to}\}\}$;
- $f_{from}, f_{to} \in \mathcal{F}$, $f_{from} \neq f_{to}$, $n_{to} \in \mathbb{N}$;
- $a_{from} \in \texttt{attr}(f_{from})$, $a_{to} \in \texttt{attr}(f_{to})$;
- c_{from}, c_{to} are ranges. They define an interval [i, j] with i, j $\in \mathbb{N}$ and i $\leq$ j. c_{from} and c_{to} are the ranges over the set of required feature instances for f_{from} and f_{to} respectively. v_{from} and v_{to} are values for the $f_{from}.a_{from}$ and $f_{to}.a_{to}$ attributes respectively;
- $comp \in \{=, <, \leq, >, \geq\}$;

Considering σ the boolean interpretation of each form of *Condition* and each form of *Constraint*,

$$\sigma(Condition) = \begin{cases} \texttt{card}(f_{from}) \geq 1, & \textit{if } Condition = f_{from} \\ \omega(f_{from}) \in c_{from}, & \textit{if } Condition = c_{from}\ f_{from} \\ \texttt{val}(a_{from})\ comp\ v_{from}, & \textit{if } Condition = f_{from}.a_{from}\ comp\ v_{from} \end{cases}$$

$$\sigma(Constraint) = \begin{cases} \texttt{card}(f_{from}) \geq 1, & \textit{if } Constraint = f_{to} \\ \omega(f_{from}) \in c_{from}, & \textit{if } Constraint = c_{to}\ f_{to} \\ \texttt{val}(a_{from})\ comp\ v_{from}, & \textit{if } Constraint = f_{to}.a_{to}\ comp\ v_{to} \\ \texttt{card}(f_{to}) \geq (n\ \times\ \texttt{card}(f_{from})), & \textit{if } Constraint = n_{to}\ f_{to} \end{cases}$$

Then, ρ is satisfied if

$$\sigma(Condition) \Rightarrow \sigma(Constraint)$$

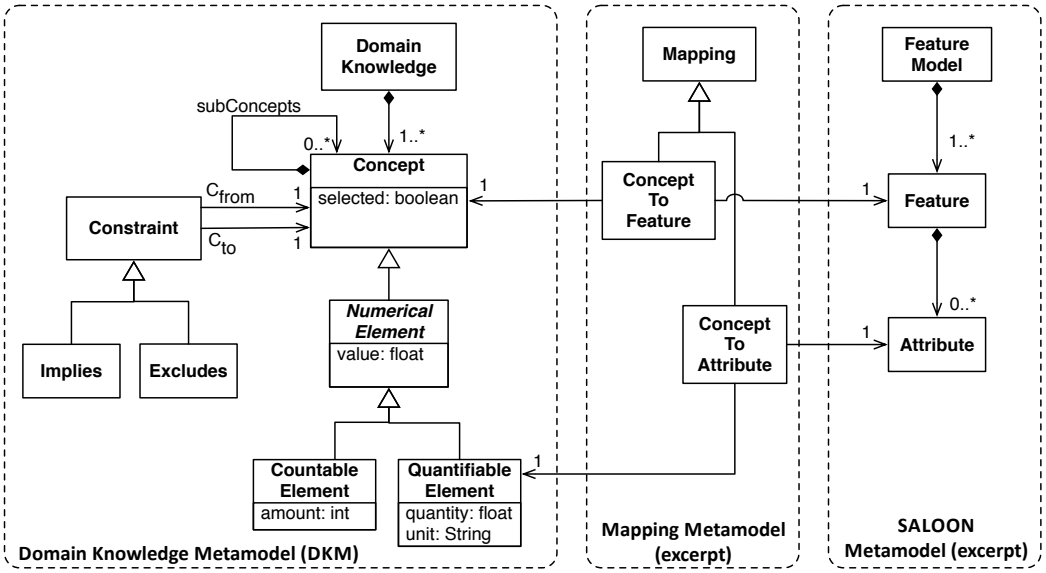
The SALOON Core thus supports the definition of FMs extended with cardinality and attributes. Even though feature models defined in the SALOON platform are used to describe cloud environments, SALOON Core is not domain specific and can be used to handle extended FMs from other domains.

4.2. Automated Feature Models Configuration

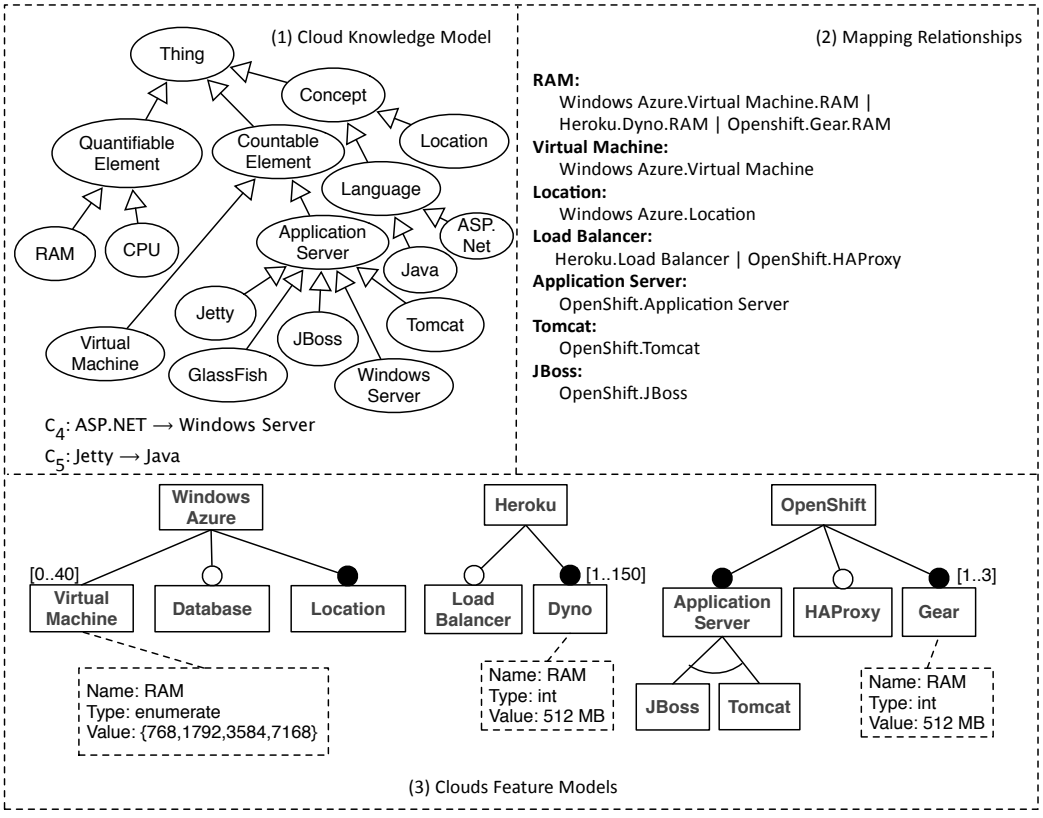
The SALOON platform relies on three metamodels, as depicted by FIG. 4 (a). The *Domain Knowledge Metamodel* is an abstract model used to define specific domain knowledge. There are three types of concepts in the Domain Knowledge Model, either `Concept`, `CountableElement` or `QuantifiableElement`. These concepts are mapped to features or attributes in the FMs, as depicted by the Mapping metamodel, using `ConceptToFeature` or `ConceptToAttribute` relationships. In SALOON, cloud environments are defined as FMs to check the validity of their configuration in an automated way. In a typical SPL engineering process, the selection of the required features is done by hand. Applied to our approach, this means selecting features in each FM, one FM after the other, which is a tedious and error-prone task since there are currently tens of cloud environments available. To cope with this issue, our approach relies on a cloud model, the Cloud Knowledge Model, describing the domain knowledge (here the one of cloud environments), together with a mapping between this model and the FMs to automate the feature selection process for these FMs. We describe below these two elements.

4.2.1. The Cloud Knowledge Model. The Cloud Knowledge Model describes the domain the SPL has been built for, here cloud environments. It defines formally all the concepts relevant to the domain, and thus gathers all the features from different cloud FMs, reifying them as concepts as depicted by FIG. 4 (b). The Cloud Knowledge Model is defined as an instance of the Domain Knowledge Metamodel. It is used by the developer to specify the application requirements, by selecting concepts and defining, when required, the related values. Basic concepts are defined as `Concept`s, *e.g.*, *Language*, that defines the language the application to deploy has been developed with and thus, the support the cloud environment must provide. `CountableElement` is used to define concepts whose required amount can be specified, *e.g.*, 4 application servers. On the other hand, `QuantifiableElement` is used to define concepts whose provided quantity and its related unit can be specified, *e.g.*, 500 MB of RAM. Some constraints are also defined over these concepts, *e.g.*, C_4: ASP.NET $\rightarrow$ Windows Server, meaning that if the application to deploy is written in ASP.NET, then the cloud environment must provide the Windows Server support to host it. Constraints defined in the Cloud Knowledge Model are constraints that are not cloud-specific, *e.g.*, C_4. Thus, these constraints are shared among every cloud environment, *e.g.*, if ASP.NET is required, any cloud environment that does not provide a Windows Server support is not well-suited and it is unnecessary to configure the related cloud FM. On the other hand, constraints defined in the FMs are cloud-specific, and thus can not be defined in the Cloud Knowledge Model.

4.2.2. The Mapping. The mapping relationships link concepts in the Cloud Knowledge Model with features or attributes in the FMs, using mappings from `Concept` to `Feature` or from



(a) The SALOON Metamodels.
The FM metamodel on the right-hand side is an excerpt of the one depicted in FIG. 3



(b) Excerpts of the Cloud Knowledge Model, the feature models and the related mappings

Figure 4. SALOON Metamodels and Related Instances

`QuantifiableElement` to `Attribute` as depicted by FIG. 4 (a). The former is used for functional requirements, like “Tomcat" or “4 Databases" (using the `CountableElement` type).

The latter defines relationships for non-functional requirements, *e.g.*, "200 MHz CPU". FIG. 4 (b) depicts how the mapping works with excerpts of the different models and relationships, Heroku and OpenShift being two other cloud environments in addition to Windows Azure. Let us now consider as an example the set of requirements REQ_1: {Tomcat, 1 GB RAM}. For instance, regarding the models and mapping relationships depicted in FIG. 4 (b), countable element `Tomcat` is mapped to the `Tomcat` feature in the Openshift FM while quantifiable element `RAM` is mapped to attributes `RAM` in all of them. For the former, the related feature is selected and a value may be given to the feature cardinality if several instances are required, while for the latter *(i)* the attribute parent features are selected and *(ii)* a value alignment algorithm taking units into account is processed that may affect feature cardinality.For example, for REQ_1 to be satisfied, the cardinality of the `Dyno` feature must be set to at least 2 (other features or constraints may require some more `Dynos`) in the `Heroku` FM, since its `RAM` attribute has 512 as value and MB as unit, and 2*512 MB $\geq$ 1 GB.

4.2.3. Advantages of this approach. Using such a mapping between the Cloud Knowledge Model and the FMs has mainly three benefits. First, it automates the feature selection process. The developer thus does not have to select features by hand in every FM, which is considerably error-prone, but simply defines the application requirements once in the Cloud Knowledge Model. Second, it bridges the semantic gap between cloud environments by mapping one Cloud Knowledge Model concept to features in different FMs with the same semantics. For example, features `Load Balancer` and `HAProxy` are mapped to the same Cloud Knowledge Model concept `Load Balancer`, since they are semantically (and functionally) equivalent even if their names differ. Finally, it reduces the range of FMs to be configured by acting like a filter. Indeed, it avoids checking the validity of certain FMs whose configuration can not cope with the requirements set. For example, if `Tomcat` is part of these requirements, then this concept can not be mapped to FMs which do not provide this application server support, *e.g.*, `Heroku` regarding FIG. 4. Thus, these FMs are not considered for the rest of the configuration process, since the related cloud environment is unsuitable. Constraints defined in the Cloud Knowledge Model, *e.g.* C_4, are also used to avoid configuring unsuitable FMs.

Even if the selection of features in the different FMs is automated regarding the defined mapping relationships, the developer still has to select the final cloud environment. Indeed, several cloud FM configurations may be valid regarding the given requirements. In such a case, the developer selects the one that best fits the requirements defined in the Cloud Knowledge Model or relying on additional criteria such as the price. This choice may be driven by the way the tool support is configured, *e.g.*, a solver, since weights can be given to the most important requirements and an optimal configuration can be found regarding those requirements. In SALOON, such an optimization is handled relying on objective functions at solver level, *e.g.*, to find a configuration that *maximizes* the memory resources while *minimizing* the configuration cost.

Since the SALOON platform relies on metamodels, it can be used for any domain where several FMs are required to describe the domain variability, with the related Domain Knowledge Model instance to specify the requirements and automate the feature selection and configuration analysis processes. Thus, what make SALOON tailored for cloud environments selection and configuration are *(i)* the Domain Knowledge Model and feature model instances and *(ii)* the artifacts used to yield the final software product, described further. Moreover, SALOON Core can be used independently from the rest of the SALOON platform, as a support to manage and configure FMs extended with cardinalities and attributes.

4.3. Cost Estimation

As described in the previous section, the configuration analysis phase allows developers to identify the cloud environments that satisfy their application requirements. However, at the end of this process, several environments can be suitable. In order to refine the selection, additional dimensions can be taken into consideration such as the cost, which represents one of the main concerns when choosing a cloud environment [19]. Therefore, once a valid configuration is found, SALOON provides a mean to compute the cost for such a configuration through a cost estimation

engine integrated to the platform. However, the real price paid by the developer once the cloud environment is configured and the application is running cannot be precisely computed, since it depends on how such an application is used, *e.g.*, the load it will support. SALOON thus *estimates* the minimum cost of a given configuration, *i.e.*, the cost to run an application on this cloud configuration. The SALOON cost estimation engine relies on a plugin-based architecture, where each cloud environment supported by the platform owns its specific plugin dedicated to estimating the cost of the related feature model's configurations. The cost estimation engine can thus be easily extended in order to support new cost plugins. LISTING 1 defines the interface implemented by the cost plugins. Depending on the cloud environment, the estimated cost is computed by using different pricing models, *i.e.*, either per hour, per month or per year. For instance, regarding Windows Azure, it is possible to select among the three of them. At this point, the reader may wonder why we did not decide to model the cost in the feature models, *e.g.*, using feature attributes. The reason is about the complexity of calculating a configuration cost, which is not as simple as the multiplication of the feature cost by the number of feature instances. Indeed, for a feature f whose cost would be $0,02$ \$/h, a configuration with 10 instances of f would not necessarily costs $0,2$ \$/h ($10 \times 0,02$ \$/h). For instance, a discount can be given starting from 5 instances, with a decreasing price for instances 5 to 10. Therefore, capturing the cost information in the feature models would require more attributes and complex constraints, thus hindering their readability and use.

```
public interface IProviderCostEstimator extends IProviderCostEstimatorFactory{

    public double estimateCost(FeatureModel fm);

    public double estimateCostPerHour(FeatureModel fm);

    public double estimateCostPerMonth(FeatureModel fm);

    public double estimateCostPerYear(FeatureModel fm);

}
```

Listing 1: Cloud environment Cost Estimator Interface

4.4. Configuration Files Derivation

In a SPL, features hold as assets reusable software artifacts that are put together to yield the final product. Thus, reasoning on feature combinations to find a valid product configuration means searching for a proper way to derive concrete software artifacts (*e.g.*, code snippets, aspects or model fragments) to yield the software product. In SALOON, feature assets are *(i)* cloud configuration files and *(ii)* configuration commands that can be executed in a command line interface or a dedicated environment. Therefore, each cloud environment FM holds its own dedicated assets. Feature can hold none, one or several assets, while an asset can be shared among several features. We illustrate the use of such assets through the Heroku PaaS example, depicted by FIG. 5. For instance, the 1.7 feature holds as asset the *system.properties* file. By default, OpenJDK 1.6 is installed on Heroku when configuring the environment to host a Java-based application. However, the developer can choose to use a newer JDK by specifying *e.g.*, *java.runtime.version=1.7* in a *system.properties* file that must be located in the root directory of the application to be deployed. Another asset example is the *Procfile*. The *Procfile* is a mandatory text file that must be located in the root directory of the application as well, that explicitly defines, among others, which process must be run once the environment is configured, *e.g.*, the main class for a Java application. While being configured, Heroku searches for such a file. If not found, the configuration process is stopped. It is thus held by the `Heroku` feature since it is required for each configuration, whatever the selected features.

Feature may also hold as assets configuration commands. Regarding the Heroku example, those commands are commands provided by the Heroku SDK, accessible from the developer's command

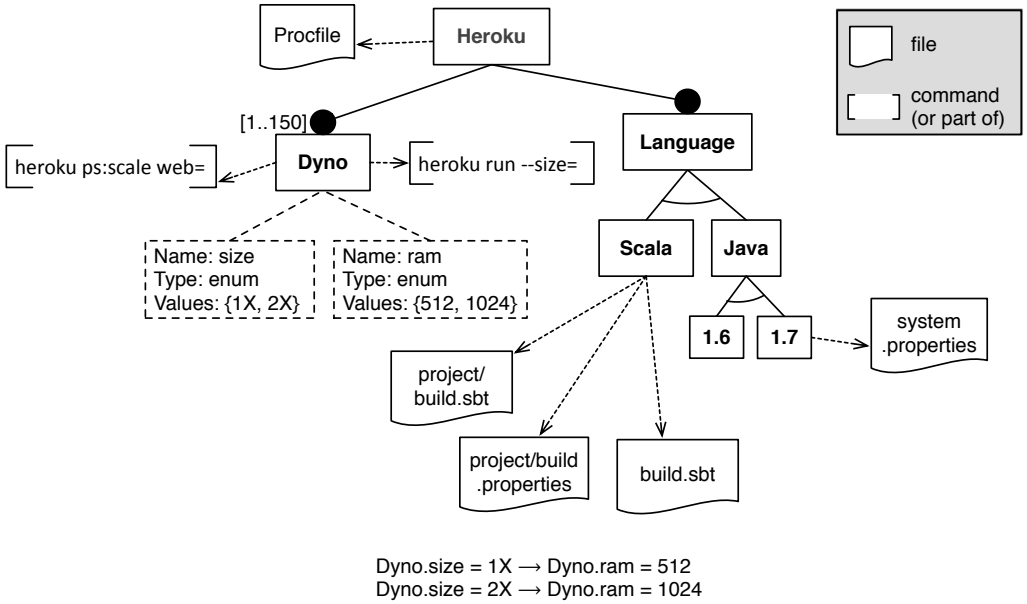
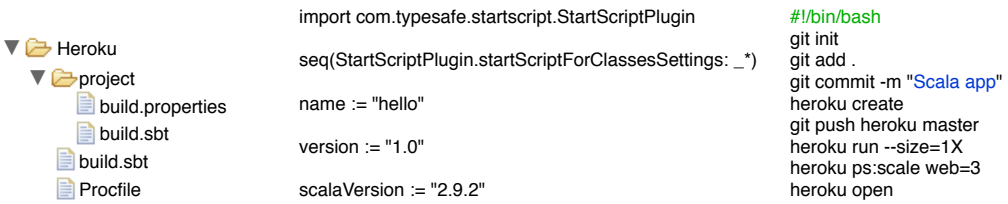


Figure 5. The Heroku FM and its Assets (excerpt)

shell. For instance, the `Dyno` feature holds as assets commands, or more precisely command parts. Thus, these commands have to be completed to be properly executed. For the `Dyno` feature, its attributes and cardinality are used to complete the commands. For instance, let us consider a valid configuration involving 8 Dynos whose size is *1X*. Taking into account this configuration, SALOON derives the two following commands, *heroku ps:scale web=8* and *heroku run --size=1X*. When several commands are required to configure the cloud environment, they are gathered in a single script shell file, which can then be executed in a command line interface.

Let us now consider as an example the set of requirements REQ_2: {Scala, 1.5 GB RAM}. Then, for Heroku to be properly configured to host a Scala application, several configuration files are required: the *Procfile*, the *build.sbt* and a *project* directory including a *build.sbt* and a *build.properties* files. All these files must be placed at the root of the Scala application, for the `Heroku` environment to recognize the Scala nature of the application and thus be properly configured. SALOON also derives the related *commands.sh* file to automate the configuration of the environment regarding the requirements. FIG. 6 depicts the different files derived by SALOON regarding REQ_2. In particular, it defines the size of the `Dyno` to `1X` and requires three of them to run, since 3*512 MB $\geq$ 1.5 GB (see FIG. 5 for `Dyno` values).



(a) Generated files (b) *build.bst* (excerpt) (c) *commands.sh*

Figure 6. Generated files to fulfill REQ_2: {Scala, 1.5 GB RAM} when deploying on Heroku

FIG. 6 (a) illustrates the derived tree view, with the 4 required configuration files. An excerpt of the *build.sbt* file is depicted by FIG. 6 (b), while FIG. 6 (c) shows the derived commands,

gathered in the *commands.sh* file. This example illustrates how defining such files by hand can be error-prone. First, the tree view must be well defined, and the files must be correctly located. Then, the files must be properly written, *e.g.*, the `build.bst` FIG. 6 (b). For example, the correct Scala version must be defined, and even the blank lines are required, otherwise Heroku does not recognize the file and the configuration fails. Finally, the commands must be well-written, and in the correct order, to be properly executed. Since the Heroku feature model holds these artifacts as assets, they can be automatically generated by SALOON, and the *commands.sh* file can thus be executed in a reliable way. As each cloud environment relies on its own commands, SALOON presupposes that the correct set of libraries are present when executing the commands, *e.g.*, Git for FIG. 6 (c).

When dealing with IaaS environments, SALOON relies on the same derivation principles but derived files are slightly different. The derivation process is twofold. First, scripts and commands for virtual machine creation are derived. Then, SALOON derives the scripts required to configure the whole environment to host the application, *e.g.*, install the Tomcat application server. Let us take as example the Windows Azure environment. To configure a virtual machine, one must *(i)* create a cloud service that will host the virtual machine, *(ii)* create a storage service to be used as hard drive of the virtual machine, *(iii)* retrieve an operating system image and *(iv)* create the virtual machine. This configuration steps can be automated, *e.g.*, by relying on the Windows Azure REST API. For instance, the request defined in LISTING 2 creates the cloud service that hosts the virtual machine.

```
$ -X POST --key arg1 --cert arg2 -H arg3 -d @arg4
    https://management.core.windows.net/arg5/services/hostedservices
```

Listing 2: REST request to create a Windows Azure cloud service

In this REST request, **`arg1`**, ..., **`arg5`** are parameters given to the derivation engine of SALOON. **`arg1`** and **`arg2`** are private key and certificate used to authenticate the developer configuring the environment, **`arg3`** is the header for the request body and **`arg4`** is the body of the request, as defined in LISTING 3, specifying in particular the location used to deploy the virtual machine.

```
<CreateHostedService xmlns="http://schemas.microsoft.com/windowsazure">
  <ServiceName>SALOON</ServiceName>
  <Label>Label4SALOON</Label>
  <Location>North Europe</Location>
</CreateHostedService>
```

Listing 3: *Body.xml* for the cloud service creation request

Finally, **`arg5`** is the identifier used to define the cloud service, *e.g.*, *SALOON_ID*. All these arguments are automatically filled when SALOON derives the related request.

Once created, the virtual machine is said *empty*, *i.e.*, there is no service running on top of the operating system to host the application. Based on the developer choices done through the Cloud Knowledge Model, SALOON can, according to the operating system, derive the commands used to install the required software. For instance, if the Tomcat 7 application server is part of the requirements, the SALOON derivation engine yields the commands depicted in LISTING 4.

```
$ sudo apt-get update
$ sudo apt-get install tomcat7
```

Listing 4: Installing software components on a Linux-based virtual machine

This example is based on the *apt-get* command-line tool, used in particular to handle the installation of software components on Linux-based distributions. Line 1 is used to update the list of available packages, *i.e.*, software components, to be sure to get the last available packages from

the *apt* repository. Executing line 2 installs the Tomcat 7 application server.

4.5. Contribution Summary

The approach we described in this section tackles the challenges identified in SEC. 2. To help the developer *selecting a suitable cloud environment* (challenge $\mathcal{C}_1$), we provide the Cloud Knowledge Model together with mapping relationships from this model to features and attributes of all cloud feature models, as described in SEC. 4.2. Thus, once requirements are defined in the Cloud Knowledge Model by the developer, SALOON searches for a cloud environment that fulfill these requirements. This search is automated by relying on feature models with attributes and cardinalities, extended with `CardExConstraints`, as explained in SEC. 4.1. Once a configuration is defined for each feature model thanks to the mapping relationships, SALOON provides a reasoning engine to *(i)* check whether the configuration is valid or not and *(ii)* estimate the cost of such a configuration, if valid (SEC. 4.3), thus tackling challenge $\mathcal{C}_2$ about *defining a proper configuration*. Finally, from a valid configuration, SALOON provides an automated support to generate configuration files and executable scripts as explained in SEC. 4.4. Parts of configuration files and executable commands artifacts are held as assets by features in the feature models, and used in the SALOON's derivation engine together with the related feature model configuration to yield the correct set of configuration files and an executable configuration script. The developer then executes this script to properly configure the cloud environment, thus ensuring the application to be *deployed in a reliable way* (challenge $\mathcal{C}_3$).

5. IMPLEMENTATION AND EVALUATION

This section describes the implementation of the SALOON platform and reports on some experiments we conducted to evaluate the platform. These experiments show that the extra capabilities in terms of configuration introduced in the SALOON platform do not penalize its performance. We also evaluate its usefulness for selecting and configuring a well suited cloud environment.

5.1. Implementation Details

As described in the previous section, the SALOON platform relies on three metamodels. These metamodels are defined using the Eclipse Modeling Framework [20], which is one of the most widely accepted metamodeling technologies. Each metamodel is thus described as an *ecore* file while dynamic instances, *i.e.*, cloud FMs, are defined as XMI models. The XMI format is used to support model persistence, in particular for SALOON to store the FM models in a dedicated repository. Once FMs are configured, SALOON loads each XMI model and parses it to generate the corresponding set of constraints, thus representing the FM as a *Constraint Satisfaction Problem* (CSP). CSP solvers are well-suited to reason about FMs extended with attributes and cardinalities, and the translation of these FMs to CSP is well-known [21, 17]. In addition, SALOON provides support for translating `CardExConstraints` described in SEC. 4.1 into CSP. Once FMs are translated into CSP, SALOON relies on the off-the-shelf Choco CSP solver [22], version 3.1.1 to reason on the configurations of these FMs [23]. We selected Choco because of its maturity and spread usage in research, education and industry. However, the architecture of the SALOON platform is flexible enough to facilitate the support for any Java CSP solver.

The configuration of these FMs is done in an automated way using the Cloud Knowledge Model as entry point of the SALOON platform. We thus expose this Cloud Knowledge Model as a RESTful service that allows its user to select concepts regarding the application requirements. We make the use of SALOON easier by defining a HTML client that invokes the RESTful service by using jQuery, a fast and feature-rich JavaScript library [24]. As depicted by FIG. 7, the client proposes the different concepts present in the Cloud Knowledge Model. They can be selected and, in some cases, values need to be specified. For example, when “PostgreSQL X.X" and “Tomcat 6.0" are selected, it is

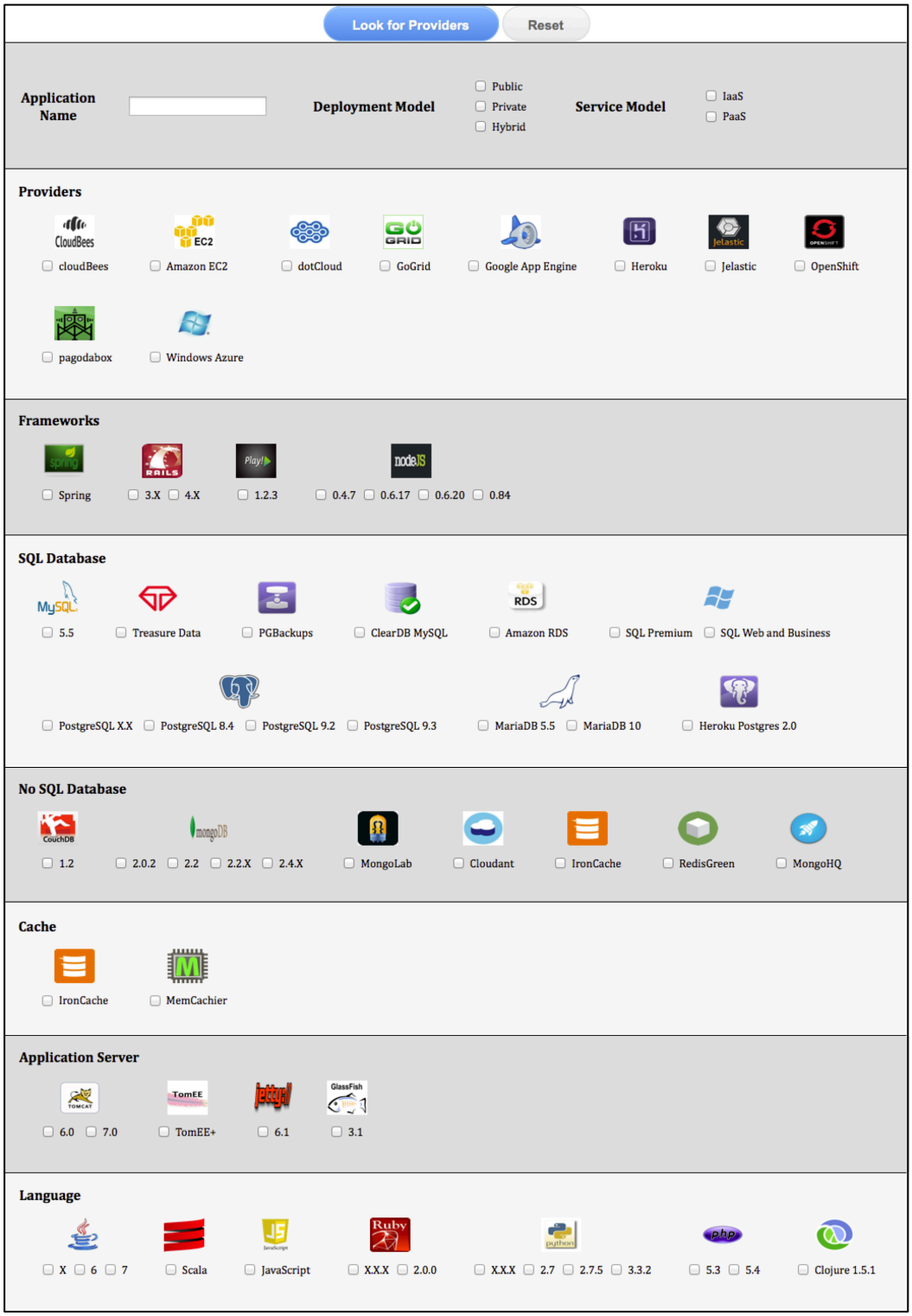


Figure 7. The Cloud Knowledge Model exposed as a RESTful service

necessary to specify the size in MB and the number of server instances respectively. The HTML

client also enables the definition of the application name, which is required for the script generation of some cloud providers.

5.2. SALOON Evaluation

In this section, we describe the experiments we conducted to evaluate the SALOON platform. This evaluation aims at investigating the following research questions:

- **R1:** ***Soundness***. Is the platform, and SALOON CORE in particular, well-suited to support cloud environment modeling and configuration?
- **R2:** ***Scalability***. Is SALOON still performing when handling FMs with a substantial amount of features and constraints and when selecting among tens of cloud environments?
- **R3:** ***Practicality***. Does SALOON usage in the configuration of cloud environments and deploying of applications improve reliability and efficiency?

Soundness. The aim of this evaluation is to empirically assess the soundness of our approach, by using the SALOON_{MM} as support to define a substantial number of cloud environments. This evaluation is based on 10 cloud environments, each one then being modeled as a FM which conforms to SALOON_{MM}. We define this set of 10 cloud FMs in the following as the $Cloud_{corpus}$. This selection is based on the following criteria:

- *Representativeness.* Both IaaS and PaaS clouds environments are represented in the $Cloud_{corpus}$. Thus, we cover a broader range of cloud providers and show that our approach is well-suited whatever the cloud layer involved is. Moreover, we select both well-known and less-known cloud providers, *e.g.*, Windows Azure and Jelastic respectively.
- *Information access.* We select clouds whose features are easily accessible either through a web configurator or in the technical documentation. Indeed, a major issue when modeling cloud environments is to find the functionalities they provide, an important information often hidden in the huge amount of available documentation.

TABLE I shows the set of cloud environments we used in our empirical evaluation. For each one, the table describes the cloud environment name (**`Cloud`**), its type (**`Type`**), the number of **`Features`** defined in the related FM, the number of **`Attributes`**, and the number of **`Constraints`**. For features and constraints, it gives details on the amount of features and constraints with cardinality and attributes, F_{card}, F_{attr} and C_{card}, C_{attr} respectively.

		Features				**Constraints**		
Cloud	**Type**	Total	F_{card}	F_{attr}	**Attributes**	Total	C_{card}	C_{attr}
`Amazon EC2`	IaaS	23	2	2	5	28	9	18
`Cloudbees`	PaaS	23	2	1	4	12	3	9
`Dotcloud`	PaaS	34	4	3	6	21	6	17
`GoGrid`	IaaS	14	3	4	10	21	7	21
`Google AE`	PaaS	23	1	5	13	10	0	10
`Heroku`	PaaS	42	1	11	20	7	0	3
`Jelastic`	PaaS	31	3	1	2	12	10	0
`OpenShift`	PaaS	29	1	2	7	18	2	15
`Pagoda Box`	IaaS/PaaS	28	5	5	9	8	4	8
`Windows Azure`	IaaS/PaaS	31	6	12	29	46	0	46

Table I. Modeled cloud environments

To assess the soundness of our approach, we determine how often do cardinalities and attributes occur in cloud environments FMs, both for features and constraints. The number of features and constraints with cardinalities and attributes varies from a cloud environment to another according to the provided services and the way we modeled it, regarding the *information access* criteria described above.

On the whole, regarding the $Cloud_{corpus}$, there are 28 features with cardinality and 46 features with attributes, while 188 constraints are based on our `CardExConstraint` expressions, which gives an average FM with about 3 features with cardinality, 5 with attributes and about 19 `CardExConstraints`. here exist some cloud feature models without constraint involving cardinalities or attributes. The main reason is the way we modeled cloud environments. Feature models used in this paper have been manually described for illustration purpose, based on our experience in cloud services configuration and deployment. We thus had to limit our feature modeling to features which are explicitly released by cloud providers, since constraints finding and modeling for implicit features are far more complex. Therefore, there might be additional constraints involving cardinalities or attributes we could not reify.

To summarize, while we can not yet conclude that our approach can be generalized to every domain with variability, results raising from this evaluation show that it remains well-suited for cloud environment modeling, while state-of-the-art approaches do not provide such a support.

Scalability. The aim of these experiments is to evaluate the performances of the SALOON platform. This evaluation is threefold. First, we measure the overhead that results from the addition of the cardinality and attribute-based constraints in the verification time of the underlying CSP solver. This evaluation aims at showing that the time to solve the models does not grow significantly with FMs modeled with the extension we provide. Second, we carried out further experiments to measure the translation time from XMI format to constraints handled by the CSP solver. This translation process, neither part of the feature modeling nor the configuration one, may be a threat to scalability of SALOON if taking too much computation time. Finally, we compute the time taken by SALOON to select features and analyze the related configuration regarding a given set of requirements. The aim of this evaluation is to show that SALOON supports the configuration of tens of cloud environments in a reasonable time.

For these experiments, we developed an algorithm that, given *nbFeat*, *nbCons* and *cardMax*, generates a random FM with *nbCons* constraints and *nbFeatures* features, whose cardinality is in the range [0..*cardMax*]. This algorithm works as follows. It creates *nbFeat* features, then randomly builds the tree hierarchy. More precisely, while there exists remaining features, it randomly selects a given amount of these features, assigns them a tree level value and increments this value, which gives the tree depth. For instance, given *nbFeat* = 10, a random tree hierarchy with 4 levels is $\{\{f1,f2,f3\},\{f4\},\{f5,f6\},\{f7,f8,f9,f10\}\}$. Then, for each feature of a given level, it randomly assigns a given amount of child features, if possible. In the previous example, if $f4$ has already been assigned as a child of $f1$, then $f2$ and $f3$ have no child feature. For features having more than one child, the algorithm determines if the relationship is a basic parent-child relationship, an alternative or an exclusive group (33% probability each). Then, 10% of features are randomly assigned an attribute, which can be an enumeration or a fixed value, either integer or real (50% probability each). The algorithm also generates *nbCons* constraints. Two features are selected randomly. If at least one of them holds an attribute, then the generated constraint is either a boolean constraint or a CardEx constraint (50% probability each). Whatever the operation generated for the CardEx constraint, each value is generated to fit within the feature cardinality or attributes value, *i.e.*, no inconsistency is introduced. In our experiments, we only consider non-void random FMs, that is, FMs with at least one valid configuration. Indeed, our algorithm sometimes generates void FMs by unfortunate generation of constraints. We performed our experiments on a MacBook Pro with a 2,6 GHz Intel Core i7 processor and 8 GB of DDR3 RAM.

For the first experiment, we generate random FMs with 10, 50, 100, 500, 1000, 5000 and 10000 features and we perform 50 random generations for each problem size. We then measure the overhead that may result from the additional verifications due to our CardEx constraints, as well as

feature cardinality themselves. We thus perform the random generation process twice. First, setting *cardMax* to 10 and generating attributes and related constraints. Second, setting *cardMax* to 1 and disabling attribute generation, thus getting a boolean FM with feature cardinality set to [0..1] or [1..1]. FIG. 8 depicts the time taken by SALOON to find a valid configuration, computed as the average time for each feature amount.

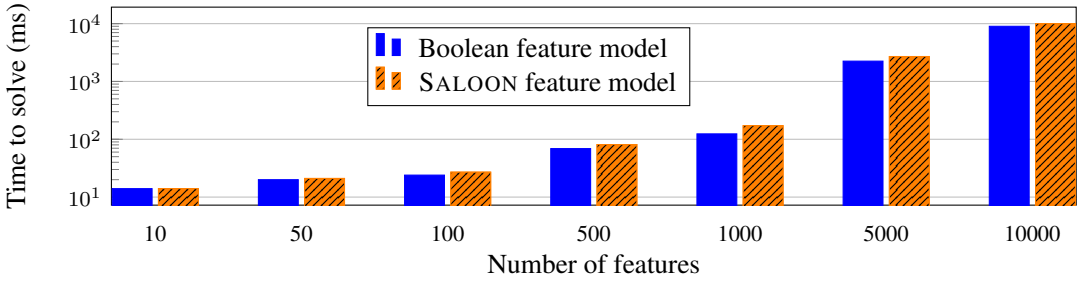


Figure 8. Time to find a valid configuration

For the second experiment, we measure the time taken by SALOON to translate XMI FMs into CSP constraints. We thus check if this translation is not a threat to SALOON scalability, in particular regarding large FMs (*i.e.*, $nbFeat > 500$). We also performed 50 random generations for each feature amount. We then measure the average computation time among these 50 runs, regarding the different model sizes. The results are depicted in FIG. 9.

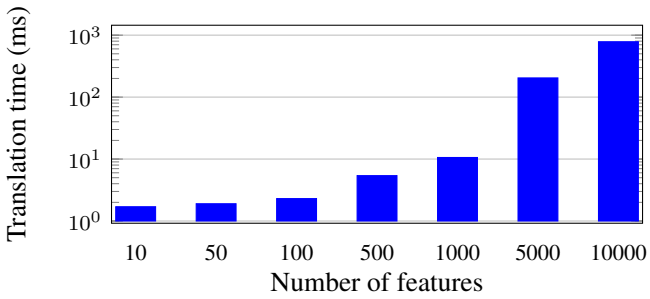


Figure 9. Time to translate from XMI to CSP

For the third experiment, we measure the time taken by SALOON for selecting features in the different FMs and check if the related configuration is valid or not. We thus pick randomly between 2 and 10 requirements in the Cloud Knowledge Model and compute the average time regarding the FM set size, as described by TABLE II. In this experiment, generated FMs size is lower than 100 features, to be as close as possible to cloud environment FMs described above.

Nb models	10	50	100	200
Time (s)	1,3	2,8	3,3	4,4

Table II. Feature selection and configuration analysis time

Analysis. Regarding the first experiment, the aim was to compute the verification time overhead for the same randomly generated FM, either with cardinality, attributes and constraints over them or considerer as a boolean FM. As illustrated by FIG. 8, the support for our cardinality-based and attribute-based expressions generates a small increase in the required time to find a solution. In average, this overhead is about +8%. Although we did not define a threshold for this experiment,

we can fairly argue that the overhead that results from using our approach is not a major threat to scalability, as solving the FMs is done within a few milliseconds. Second, as shown by FIG. 9, the translation time from a feature model described as an XMI model to CSP constraint is from 16 to 519 ms for 10 to 10000 features respectively. This time is slightly increasing with the size of the model, but we believe that it is not a major threat to scalability for the two following reasons. First, the bigger feature model from the cloud corpus contains "only" 42 features (Heroku). Moreover, most of existing feature models contain less than 500 features, *e.g.*, those from the S.P.L.O.T. repository [25]. Then, one of the biggest existing FM, which is the Linux feature model, has over 5000 features [26]. This translation time overhead remains therefore fairly low and insignificant and does not hinder the usability of the SALOON framework. Finally, SALOON is able to map the requirements by selecting the related features and check whether the configuration is valid or not within a few seconds, even for 200 FMs, as illustrated by TABLE II. This time is negligible and is not a threat to the scalability of SALOON, compared to the time taken to configure those FMs manually. We believe that 2 to 10 requirements is a representative amount. Basically, developers specifies their requirements among the language support, the application server, the database, the number of virtual machines or the amount of resources. Moreover, specifying more requirements would be usually inefficient, as it increases the risk not to find any valid configuration for this requirements set. Overall, as our empirical evaluation shows, we observe that SALOON is well-suited to *(i)* handle an important number of cloud environments and *(ii)* deal with realistic cloud FMs, with a substantial number of features and constraints, either over features, attributes or cardinalities.

Practicality. The aim of this last experiment is to evaluate the reliability and efficiency of the SALOON platform, compared to a manual configuration process. This experiment is divided into two stages. For the first stage, an experiment is conducted with a group of 10 participants, either Ph.D. students or engineers. Each of these participants is given the same task: *Configure an Heroku environment, upload a web application, then add a PostgreSQL support.* The prerequisite is that Git and Eclipse must be installed on every participant computer, while we provide the web application (a basic *HelloWorld* application as a .war file). They are then free to select the way they proceed, either using Git (G), the Eclipse plugin (P) or the web interface (W). Participants are asked to time their experiment and define their experience in cloud configuration and deployment in a range from beginner (1) to expert (5). TABLE III describes the results of this first experiment.

Participant	1	2	3	4	5	6	7	8	9	10
Time (min)	26	19	32	26	48	60	17	-	23	28
Method	G	G	G	P	P	G	G	G	W	G
Experience	2	4	2	3	3	1	4	1	3	2
App running	✓	✓	-	✓	✓	-	-	-	✓	-

Table III. Configuring Heroku and deploying the application

For the second stage, we conducted another experiment with a group of 8 participants, 5 from the first group and 3 persons which were not involved in the first stage, one engineer and two researchers. They were assigned quite the same task: *Use* SALOON *to configure an Heroku environment, upload a web application and add a PostgreSQL support.* TABLE IV describes the results of this second experiment.

Analysis. The task assigned to the first group of participants is rather simple (adding a PostgreSQL support is straight forward, it is not asked to connect the web application with the database) but takes at least 19 minutes to be manually completed by an experienced participant. One of them (#8) even gave up after several failed attempts. Moreover, the results show that whatever the way

Participant	1	2	3	4	5	6	7	8
Time (min)	4	6	7	5	8	11	12	7
Method	SALOON					SALOON		
Experience	4	1	4	1	2	2	2	3
App running	✓	✓	✓	✓	✓	✓	✓	✓

Table IV. Configuring Heroku and deploying the application using SALOON

used to deploy, it can be very long to achieve the task, *e.g.*, participant #5 with a high level of experience and a dedicated plug-in. The last row of the table indicates whether the application is running or not at the end of the deployment. Indeed, an environment can be created and (incorrectly) configured, leading to the application not to run properly. Overall, 50% of participants failed to get the application running and the average time among those who succeeded was more than 28 minutes to configure the cloud environment and get the application running properly. For the second stage, every participant succeeded. Using SALOON, there is no need to configure anything manually. Participants *(i)* select the correct set of requirements in the Cloud Knowledge Model (*Java* and *PostgreSQL*) then run SALOON, *(ii)* copy/paste the generated configuration files in the application root directory and *(iii)* run the commands generated in the executable file to automate the configuration. One can observe a difference in the time required by the five first participants and the three last ones (6 and 10 minutes in average respectively). This is mainly due to the fact that the first five participants already knew the task to achieve and experiment to be performed, while the third last discovered at the same time the tool and the experiment. In average, the time required to achieve this experiment was 8 minutes using SALOON and 30 minutes with a manual process (for the latter, we consider the average value for the five successful deployments), leading to a time reduction of 73%.

Over the five participants from the first stage who did not get their application running, four of them had a *Procfile* that was not properly written, leading to a wrong configuration. This experiment thus highlights the need for an automated support. Indeed, participants are asked to configure a cloud environment to deploy a basic application, and 50% of them fail. We argue that this number can be higher, especially if the configuration is more complex and requires more knowledge. Moreover, we only consider in this experiment one given cloud provider while there are tens of them to be taken into account when considering deploying an application.

Summary. Overall, as our empirical evaluation shows, we observe that SALOON is well-suited to handle the configuration of cloud environments. Relying on FMs in SALOON CORE to describe these environments is not a threat to practicality nor scalability as complex constraint expressions can be defined, and finding a related configuration is achieved with a negligible overhead. When handling a significant amount of modeled cloud environments, SALOON is still performing and the computation time required to find a valid configuration from a requirements set remains fairly low, thus not hindering its usability. Finally, automating the FM configuration files generation improves the reliability of the deployment process compared to manual and error-prone process.

6. RELATED WORK

In these section, we describe the work that is close-related to our approach. First, we introduce works that are specific to the cloud domain, focusing on cloud environment selection, configuration and comparison. Then, we discuss existing approaches related to extended feature models in software product lines.

6.1. Cloud Environments Configuration

Several cloud environment variability modeling and configuration approaches have been proposed in recent works.

In 2010, Van der Aalst [27] showed that handling variability is one of the main challenges to support configurable cloud services, and proposed configurable models to support cross-organizational processes mining. Calheiros *et al.* [28] developed the CloudSim framework for modeling and simulating cloud infrastructures. Clouds are described as abstract classes or interfaces at code level, which can then be implemented. This approach is well suited to simulate IaaS clouds but misses an abstraction level to handle properly both cloud selection and configuration. Ruiz-Alvarez *et al.* [29] use an XML schema format to describe cloud storage services and find which one is the best suited for a given dataset, relying on a specifically developed application. Our approach also supports that, and provides additionally a means of configuring automatically these services and expressing constraints between them using FMs. Some authors [30, 31] proposed a survey on existing approaches to model variability in cloud environment. Moreover, FMs have been used in recent work to describe cloud services. Wittern *et al.* [32] present a cloud service selection process based on variability modeling. They rely on FMs to describe cloud services, but they handle neither cardinalities nor constraints over cardinalities and attributes. Galán *et al.* [33] propose to use an SPL-based approach to configure the Amazon IaaS. They describe Amazon EC2, EBS, S3 and RDS services as FMs and rely on off-the-shelf solvers to find a suitable configuration. The approach we propose in this paper goes in the same direction, but we go further in the SPL process. Our FM analysis is not limited to boolean FMs and thus handles properly the whole configuration. We also provide a tool to build the related software artifacts. Schmid *et al.* [34] combine SPL engineering with service-oriented computing to deal with the variability of service platforms, *e.g.*, cloud platforms. Their paper explains how SPLs could help in such a case, but remains at a theoretical level, since no concrete example or validation is provided. Dougherty *et al.* [35] explain how virtual machine (VM) configurations can be captured by feature models. They also use attributes to define the energy consumption of a feature, in order to find a configuration that meets the requirements with the least energy consumption. Although this approach is closely related to ours, it does not provide means to reason about attributes and cardinalities, and does not automatically derive the VM configuration. Di Cosmo *et al.* [36] describe an approach that proposes a deployment configuration according to the requirements of the user or of a higher level application. This approach is based on a component model, where components describe resources which provide and require different functionalities. These components represent software packages, *e.g.*, packages in Debian, where components requires other ones, *e.g.*, with *Depends* requirements.

Compared to these approaches, SALOON provides the whole automated support, from requirements specification through the Cloud Knowledge Model to the automated configuration using generated configuration files by leveraging the software product lines principles.

6.2. Cloud Environments Comparison

Several tools have been developed which enable the comparison of cloud environments (PaaS and IaaS) through a friendly user interface [37, 38, 39, 40, 41, 42]. Most of them focus on criteria such as location, required resources and price. For instance, Cloudorado [38] compares the cost of 23 IaaS environments regarding the region, the operating system and the provides resources such as RAM, storage space and CPU power. Other criteria may be taken into consideration when comparing cloud environments per se. FindTheBest [37], allows the user to compare 176 environments in terms of service model, deployment model, subscription options (*e.g.*, price rates, free plan, and trials), server locations, services (*e.g.*, autoscaling, storage and firewalls) and average user rates. Cloud Finder [39], provided by Intel, considers 70 environments with two different variants: *(i)* quick search and *(ii)* detailed search. The former enables the cloud search by using criteria such as interface model (*e.g.*, GUI, proprietary and standard APIs), development support (*e.g.*, custom image and open virtualization format), subscription options (*e.g.*, self service, monthly subscription and pay as you go) and locations. The latter allows the definition of more specialized searches by using criteria such as security, usability, quality, availability and technology. Cloud Screener [40]

proposes a web tool enabling the comparison of cloud environments using similar criteria that the previously presented tools. However, it allows the definition of priorities (*i.e.*, medium, important, critical) in terms of price, performance and security. Cloud Surfing [41] is a charged solution for searching for IaaS and PaaS based on different needs such as automation, data centre, optimization, storage, security, hosting and virtualization. In this solution, it is possible to select in a catalog different cloud environments to be compared. Finally, CloudSleuth provides real time monitoring regarding availability and response time [42]. Although this is not a real comparison tool, the monitored information can be used as a base to select cloud environments.

Compared to these approaches, the main objective of the SALOON platform is not to compare cloud environments but assisting developers in selecting and configuring a suitable cloud environment according to application requirements. However, we could be inspired by those existing tools in order to improve SALOON's support by including additional dimensions in the cloud analysis, *e.g.*, environment rankings, response time and availability as well as the relevance given to these dimensions.

6.3. Extended Feature Modeling

FMs were first introduced in 1990 [8] and various extensions have been proposed since then [43]. Cardinality-based feature models support in addition feature cardinalities [10, 11], first introduced as UML-like multiplicities [9], while attributes were already part of the FODA seminal report. Then, several authors proposed the inclusion of attributes in FMs. For example, Czarnecki *et al.* [44] describe the inclusion of attributes, reference attributes and cardinalities for dealing with variability in embedded systems. Authors identify the shortcomings with boolean FMs in the context of this kind of systems but they do not mention the issues related to the management of constraints on attributes and cardinalities. Furthermore, they described some modeling tools such as MetaCASE, Ami Eddi and ConfigEditor, which do not provide a full support for the reasoning on extended FMs. Other approaches such as Clafer [45], VELVET [46] or TVL [47] only partially support constraints on attributes values. For instance, assigning a value to an attribute is not fully supported by these variability modeling approaches [48], while SALOON does provide such a support. Authors also proposed approaches regarding constraints with cardinalities. For instance, Zhang *et al.* [49] presented a binary decision diagram based approach to verify constraints with cardinalities, based on their own semantics of cloning. They described their different constraint patterns and the way they can be verified but did not provide any abstract syntax or tool support to define such constraints in FMs. In [50], the authors present their own metamodel and the way they rely on model-driven engineering to configure their FMs. However, they only introduce a new kind of constraint denoted as *Use*, where a feature A can use a given amount of features B.

Compared to these approaches, SALOON provides an automated support to reason both on constraints with cardinalities and attributes. Moreover, they can be combined in the same constraint since the value of an attribute may imply a value for a cardinality, and conversely. In addition, SALOON relies on an abstract syntax to describe such extended feature models, thus being implementation-independent and allowing any approach to use this abstract syntax.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented SALOON, a platform for selecting a suitable cloud environment, defining a valid configuration for this environment and generating the required configuration scripts and files to automate the configuration of this cloud environment. The SALOON platform relies on software product lines principles, enabling the developer to automatically *(i)* select the cloud environment that fits a set of requirements and *(ii)* get the description files and executable scripts to configure the related cloud environment. In particular, SALOON is based on feature models extended with cardinalities and attributes with constraints over them, together with a Cloud Knowledge Model used as entry point of the platform. Such extended feature models are independent of cloud environments and can be applied to different application domains, *e.g.*, real time systems [44].

Mapping rules are then established between the Cloud Knowledge Model and the FMs to enable an automated configuration of the different feature models. The Cloud Knowledge Model, the mapping rules and the FMs are all instances of their related metamodel. Such a model-based structure results in an highly modular and extensible platform. The experiments we conducted show that SALOON provides a reliable support when selecting and configuring a cloud environment. The extensions we propose to express constraints over both cardinality and attributes when feature modeling cloud environments are not a threat to practicality nor scalability as finding a valid configuration from a requirements set is done in a negligible amount of time.

Regarding future work, we plan to extend the SALOON platform towards three directions: software product lines evolution, dynamic software product lines and multi-cloud configuration. Concerning evolution, the feature models within a software product line evolve over time [51]. In particular with SALOON, since the cloud market evolves constantly, the underlying models have to evolve consequently. To deal with such changes, the evolution of feature models extended with attributes and cardinalities must be taken into consideration. Evolving such FMs is error-prone, as inconsistencies may arise, *e.g.*, between feature cardinalities and values defined in constraints. Evolving SALOON also includes improving the platform. For instance, we can include new dimensions or metrics, such as response time and availability, in order to help the developer for the cloud selection. Such a support can be included using external services, *e.g.*, Global Provider View from CloudSleuth providing real time values for those metrics [42]. The second domain of extension of SALOON, dynamic SPL, is about adding support for product adaptation at runtime [52]. Indeed, changes can occur that require the application environment to be reconfigured, *e.g.*, non-functional requirements such as response-time, availability or pricing are violated or new cloud providers, which better meet these non-functional requirements, are now available. In such cases, the SPL has to be well-suited to handle this adaptation, *e.g.*, defining a feedback control loop [53] that monitors the metrics, detects adaptation situations and execute the required actions in order to modify the deployment of applications. Finally, we believe that our approach could also be relevant to target a multi-cloud configuration [54]. In the current version of the SALOON platform, a cloud environment is not considered anymore if it does not provide the whole set of services matching the developer requirements. However, a multi-cloud configuration may be better suited regarding these requirements if one could configure several environments.

REFERENCES

1. Armbrust M, Fox A, Griffith R, Joseph AD, Katz RH, Konwinski A, Lee G, Patterson DA, Rabkin A, Stoica I, *et al.*. Above the Clouds: A Berkeley View of Cloud Computing. *Technical Report UCB/EECS-2009-28*, EECS Department, University of California, Berkeley Feb 2009. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`.
2. Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Gener. Comput. Syst.* June 2009; **25**:599–616, doi:10.1016/j.future.2008.12.001. URL `http://dl.acm.org/citation.cfm?id=1528937.1529211`.
3. Mell P, Grance T. The NIST Definition of Cloud Computing. *Technical Report*, National Institute of Standards and Technology 2009.
4. Quinton C, Romero D, Duchien L. Automated Selection and Configuration of Cloud Environments Using Software Product Lines Principles. *Proceedings of the Seventh International Conference on Cloud Computing (CLOUD)*, IEEE CLOUD'14, 2014.
5. Glinz M. On Non-Functional Requirements. *15th IEEE International Requirements Engineering Conference, 2007. RE'07.*, 2007; 21–26, doi:10.1109/RE.2007.45.
6. Clements P, Northrop LM. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2001.
7. Pohl K, Böckle G, Linden FJvd. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc.: Secaucus, NJ, USA, 2005.
8. Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS. Feature-Oriented Domain Analysis (FODA) - Feasibility Study. *Technical Report*, The Software Engineering Institute 1990. URL `http://www.sei.cmu.edu/reports/90tr021.pdf`.
9. Riebisch M, Böllert K, Streitferdt D, Philippow I. Extending Feature Diagrams with UML Multiplicities. *6th World Conference on Integrated Design & Process Technology (IDPT2002)*, 2002.
10. Czarnecki K, Helsen S, Eisenecker UW. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process: Improvement and Practice* 2005; **10**(1):7–29.
11. Czarnecki K, Kim CHP. Cardinality-Based Feature Modeling and Constraints: A Progress Report. *International Workshop on Software Factories at OOPSLA'05*. ACM: San Diego, California, USA, 2005.

12. Benavides D, Trinidad P, Ruiz-Cortés A. Automated Reasoning on Feature Models. *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, CAiSE'05, Springer-Verlag: Berlin, Heidelberg, 2005; 491–503, doi:10.1007/11431855_34. URL http://dx.doi.org/10.1007/11431855_34.
13. Batory D, Benavides D, Ruiz-Cortes A. Automated Analysis of Feature Models: Challenges Ahead. *Commun. ACM* Dec 2006; **49**(12):45–47, doi:10.1145/1183236.1183264. URL http://doi.acm.org/10.1145/1183236.1183264.
14. Windows Azure. http://azure.microsoft.com. Accessed 07.07.2014.
15. PaaSage: Model-based Cloud Platform Upperware. http://www.paasage.eu 2013. Accessed 07.07.2014.
16. Quinton C, Romero D, Duchien L. Cardinality-based feature models with constraints: a pragmatic approach. *Proceedings of the 17th International Software Product Line Conference*, SPLC'13, ACM: New York, NY, USA, 2013; 162–166, doi:10.1145/2491627.2491638. URL http://doi.acm.org/10.1145/2491627.2491638.
17. Benavides D, Segura S, Trinidad P, Ruiz-Cortés A. Using Java CSP Solvers in the Automated Analyses of Feature Models. *Proceedings of the 2005 International Conference on Generative and Transformational Techniques in Software Engineering*, GTTSE'05, Springer-Verlag: Berlin, Heidelberg, 2006; 399–408, doi:10.1007/11877028_16. URL http://dx.doi.org/10.1007/11877028_16.
18. Czarnecki K, Helsen S, Eisenecker UW. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice* 2005; **10**(2):143–169. URL http://dblp.uni-trier.de/db/journals/sopr/sopr10.html#CzarneckiHE05a.
19. Den Haan J, Thiele M, Chase N, Butcher M. The 2014 Cloud Platform Research Report. *Technical Report*, DZone 2014. URL http://www.dzone.com/page/cloud-research-report.
20. Steinberg D, Budinsky F, Paternostro M, Merks E. *EMF: Eclipse Modeling Framework 2.0*. 2nd edn., Addison-Wesley Professional, 2009.
21. Mazo R, Salinesi C, Diaz D, Lora-Michiels A. Transforming Attribute and Clone-enabled Feature Models into Constraint Programs over Finite Domains. *ENASE*, Maciaszek LA, Zhang K (eds.), SciTePress, 2011; 188–199.
22. Jussien N, Rochart G, Lorca X. Choco: an Open Source Java Constraint Programming Library. *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, Paris, France, France, 2008; 1–10. URL http://hal.archives-ouvertes.fr/hal-00483090.
23. Choco 3. http://www.emn.fr/z-info/choco-solver/index.php?page=choco-3. Accessed 07.07.2014.
24. The jQuery library. http://jquery.com/. Accessed 07.07.2014.
25. S.P.L.O.T. http://www.splot-research.org/. Accessed 07.07.2014.
26. She S, Lotufo R, Berger T, Wasowski A, Czarnecki K. Reverse engineering feature models. *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, ACM: New York, NY, USA, 2011; 461–470, doi:10.1145/1985793.1985856. URL http://doi.acm.org/10.1145/1985793.1985856.
27. Aalst W. Configurable Services in the Cloud: Supporting Variability While Enabling Cross-Organizational Process Mining. *On the Move to Meaningful Internet Systems: OTM 2010*, *Lecture Notes in Computer Science*, vol. 6426, Meersman R, Dillon T, Herrero P (eds.). Springer Berlin Heidelberg, 2010; 8–25, doi:10.1007/978-3-642-16934-2_5. URL http://dx.doi.org/10.1007/978-3-642-16934-2_5.
28. Calheiros RN, Ranjan R, Beloglazov A, De Rose CAF, Buyya R. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Softw. Pract. Exper.* Jan 2011; **41**(1):23–50, doi:10.1002/spe.995. URL http://dx.doi.org/10.1002/spe.995.
29. Ruiz-Alvarez A, Humphrey M. An Automated Approach to Cloud Storage Service Selection. *Proceedings of the 2Nd International Workshop on Scientific Cloud Computing*, ScienceCloud '11, ACM: New York, NY, USA, 2011; 39–48, doi:10.1145/1996109.1996117. URL http://doi.acm.org/10.1145/1996109.1996117.
30. Eichelberger H, Kröher C, Schmid K. Variability in Service-Oriented Systems: an Analysis of Existing Approaches. *Proceedings of the 10th international conference on Service-Oriented Computing*, ICSOC'12, 2012; 516–524, doi:10.1007/978-3-642-34321-6_35.
31. Benlachgar A, Belouadha FZ. Review of Software Product Line Models Used to Model Cloud Applications. *Computer Systems and Applications (AICCSA), 2013 ACS International Conference on*, 2013; 1–4, doi:10.1109/AICCSA.2013.6616430.
32. Wittern E, Kuhlenkamp J, Menzel M. Cloud service selection based on variability modeling. *Proceedings of the 10th international conference on Service-Oriented Computing*, ICSOC'12, 2012; 127–141, doi:10.1007/978-3-642-34321-6_9.
33. García-Galán J, Rana OF, Trinidad P, Ruiz-Cortés A. Migrating to the Cloud: a Software Product Line based analysis. *3rd International Conference on Cloud Computing and Services Science (CLOSER)*, 2013.
34. Schmid K, Eichelberger H, Kröher C. Domain-oriented customization of service platforms: Combining product line engineering and service-oriented computing. *Journal of Universal Computer Science* jan 2013; **19**(2):233–253.
35. Dougherty B, White J, Schmidt DC. Model-driven Auto-scaling of Green Cloud Computing Infrastructure. *Future Gener. Comput. Syst.* 2012; **28**(2):371–378, doi:10.1016/j.future.2011.05.009.
36. Di Cosmo R, Zacchiroli S, Zavattaro G. Towards a formal component model for the cloud. *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, Springer-Verlag: Berlin, Heidelberg, 2012; 156–171, doi:10.1007/978-3-642-33826-7_11. URL http://dx.doi.org/10.1007/978-3-642-33826-7_11.
37. Find The Best. http://cloud-computing.findthebest.com/#main. Accessed 07.07.2014.
38. Cloudorado. http://www.cloudorado.com/. Accessed 07.07.2014.
39. Intel Cloud Finder. http://www.intelcloudfinder.com/. Accessed 07.07.2014.
40. Cloud Screener. http://www.cloudscreener.com/en. Accessed 07.07.2014.
41. Cloud Surfing. http://www.cloudsurfing.com/browse/categories/576-Infrastructure/. Accessed 07.07.2014.

42. Global provider View Service. https://cloudsleuth.net/global-provider-view. Accessed 07.07.2014.
43. Schobbens PY, Heymans P, Trigaux JC. Feature Diagrams: A Survey and a Formal Semantics. *14th IEEE International Requirements Engineering Conference, RE'06.*, 2006; 136–145, doi:10.1109/RE.2006.23.
44. Czarnecki K, Bednasch T, Unger P, Eisenecker UW. Generative programming for embedded software: An industrial experience report. *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, GPCE '02, Springer-Verlag: London, UK, UK, 2002; 156–172. URL http://dl.acm.org/citation.cfm?id=645435.757601.
45. Bak K, Czarnecki K, Wasowski A. Feature and meta-models in clafer: Mixed, specialized, and coupled. *Proceedings of the Third International Conference on Software Language Engineering*, SLE'10, Springer-Verlag: Berlin, Heidelberg, 2011; 102–122. URL http://dl.acm.org/citation.cfm?id=1964571.1964581.
46. Rosenmüller M, Siegmund N, Thüm T, Saake G. Multi-dimensional Variability Modeling. *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, ACM: New York, NY, USA, 2011; 11–20, doi:10.1145/1944892.1944894. URL http://doi.acm.org/10.1145/1944892.1944894.
47. Classen A, Boucher Q, Heymans P. A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL. *Sci. Comput. Program.* Dec 2011; **76**(12):1130–1143, doi:10.1016/j.scico.2010.10.005. URL http://dx.doi.org/10.1016/j.scico.2010.10.005.
48. Eichelberger H, Schmid K. A Systematic Analysis of Textual Variability Modeling Languages. *Proceedings of the 17th International Software Product Line Conference*, SPLC'13, ACM: New York, NY, USA, 2013; 12–21, doi:10.1145/2491627.2491652. URL http://doi.acm.org/10.1145/2491627.2491652.
49. Zhang W, Yan H, Zhao H, Jin Z. A BDD-Based Approach to Verifying Clone-Enabled Feature Models' Constraints and Customization. *High Confidence Software Reuse in Large Systems*, *Lecture Notes in Computer Science*, vol. 5030, Mei H (ed.). Springer Berlin Heidelberg, 2008; 186–199, doi:10.1007/978-3-540-68073-4_18. URL http://dx.doi.org/10.1007/978-3-540-68073-4_18.
50. Gómez A, Ramos I. Cardinality-Based Feature Modeling and Model-Driven Engineering: Fitting them Together. *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*, *ICB-Research Report*, vol. 37, Benavides D, Batory DS, Grünbacher P (eds.), Universität Duisburg-Essen, 2010; 61–68, doi:http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf.
51. Pleuss A, Botterweck G, Dhungana D, Polzer A, Kowalewski S. Model-driven Support for Product Line Evolution on Feature Level. *J. Syst. Softw.* Oct 2012; **85**(10):2261–2274, doi:10.1016/j.jss.2011.08.008. URL http://dx.doi.org/10.1016/j.jss.2011.08.008.
52. Hallsteinsen S, Hinchey M, Park S, Schmid K. Dynamic Software Product Lines. *Computer* Apr 2008; **41**(4):93–95, doi:10.1109/MC.2008.123. URL http://dx.doi.org/10.1109/MC.2008.123.
53. Kephart JO, Chess DM. The vision of autonomic computing. *Computer* Jan 2003; **36**(1):41–50, doi:10.1109/MC.2003.1160055. URL http://dx.doi.org/10.1109/MC.2003.1160055.
54. Paraiso F, Haderer N, Merle P, Rouvoy R, Seinturier L. A Federated Multi-cloud PaaS Infrastructure. *IEEE 5th International Conference on Cloud Computing (CLOUD)*, 2012; 392–399, doi:10.1109/CLOUD.2012.79.