**Département de formation doctorale en informatique** **École doctorale SPI Lille**
**UFR IEEA**

# Cloud Environment Selection and Configuration: A Software Product Lines-Based Approach

## THÈSE

présentée et soutenue publiquement le 22 Octobre 2014

pour l'obtention du

**Doctorat de l'Université Lille I**

**(spécialité informatique)**

par

Clément Quinton

**Composition du jury**

| | |
|---|---|
| *Rapporteurs :* | Roberto Di Cosmo, *Université Paris Diderot* - France |
| | Christian Perez, *Inria* - France |
| *Examinateurs :* | Patrick Heymans, *Université de Namur* - Belgique |
| | Daniel Le Berre, *Université d'Artois* - France |
| | Daniel Romero, *Université Lille I* - France |
| *Directrice :* | Laurence Duchien, *Université Lille I - France* |

**Laboratoire d'Informatique Fondamentale de Lille – UMR USTL/CNRS 8022 – Inria Lille - Nord Europe**

Mis en page avec la classe thloria.

“We know it well that none of us acting alone can achieve success."

Nelson Mandela, *Inaugural Address, May 10th 1994.*

*To Louise*

# Acknowledgement

First of all, I would like to really thank my supervisor, Laurence Duchien. For five years now, we have learned to know each other, and we have built a relationship made of mutual respect, professionalism and sense of humor. Thank you for taking my request into consideration three years ago when, at the end of my temporary contract in your team, you gave me your trust and accepted me to start this PhD. Thank you for still trusting me when, at the end of my first year, I did not take charge of my thesis subject yet. Thank you again for your constant support, feedback and advices given parsimoniously to let me learn on my own. I was not really sure of the effectiveness of this kind of guidance at first, but I can now guarantee it was definitively the right way to supervise me. You gave me the taste for research, and I hope we will still have a lot of fun together. Thank you!

Next, I would like to thank the members of my thesis committee. Roberto Di Cosmo and Christian Perez, thank you for having accepted to review my manuscript. You gave me valuable comments, and I really appreciated your feedback. A special thanks to Roberto for the discussions we have had during my PhD, and for the invitation to give a talk in your team nearly two years ago. I was very honored, but at the same time tremendously stressed! Thank you. Then, I would like to thank Patrick Heymans and Daniel Le Berre, to have accepted to be part of my committee but most of all, because you generously shared your expertise with me during your presence in my office as visiting professors. In addition to be leading researchers in your own areas, you both are great people to discuss and share jokes with! Finally, a special thanks to Daniel Romero. I would like to thank you for all the great work you did during my PhD, always giving me feedback and reviewing most of the ideas described in this dissertation. Thanks for your support, thanks for the running sessions, and thanks for the love of life you bring every day in the office. Keep smiling, and I wish you the best for your future. You deserve it.

I would also like to thank Goetz Botterweck and Andreas Pleuss for their hospitality during the three months I visited them at Lero, in Limerick. Goetz, even though we did not have lot of time to discuss together, I really appreciated your research skills and most of all, the research vision you described me on my arrival day: "*have fun*"! Andreas, thanks a lot for everything you did for me. Not only the amazing research work, but all your everyday friendship, giving me the opportunity to play football together, go out and drink beers, and even meet your 3 days old daughter. Take care of her and your wife, who I thank as well for receiving me in your house, in particular for an unforgettable sweating Indian meal.

During the last three years, I had the chance to be part of the SPIRALS (formerly ADAM) team. I would thus like to thank all the permanent members of this team, Lionel Seinturier, Philippe Merle, Romain Rouvoy and Martin Monperrus. Thanks for all the feedback and comments you gave me, as many ideas emerged from these inspiring discussions. More generally, I would like to thank all the members of the team I met since 2009. It would be hard for me not to forget someone, regarding the existing turn-over in the team. So I would just thank the PhD students who assisted me in the last three years, as we suffered together

:-). I wish you the best for your future. A special thanks however to a former postdoc, Sébastien Mosser. I would like to thank you for the endless discussions we had 3 years ago, about starting or not a PhD. Thanks a lot for convincing me (well, it was not really hard, was it?), I am far from regretting today. I would like to thank you also for supporting me in the tedious writing process of my first ever research paper. Thanks to you (and Laurence), I guess I now get the correct strictness to do it on my own.

Finally, I would like to thank my family and friends for their immeasurable support. Thanks to my parents, Véronique and Philippe, to my sister Cécile, my brother Cyril and of course Laurent and Sophie. Thank you for supporting me, even in the distance. Knowing you believe in me has always been a great support in the hard times. Thank you for everything! Thank you, Aurélien, Fanny, Rémi and Sarah, for the great time we had together. I hope to see you again soon. Thank you Aldo and Thérèse for preparing me succulent dishes every week. Last but not least, I would like to thank my everyday life sugar, Louise. I know I haven't been as available as you wished, and I know how deeply you supported me for 3 years now. I will eternally be grateful to you for that. Thanks for your love, and thanks for believing in me. You are my daily motivation and inspiration and for sure, I would not be who I am today without you.

## Abstract

In the recent years, cloud computing has become a major trend in distributed computing environments enabling software virtualization on configurable runtime environments. These environments provide numerous highly configurable software resources at different levels of functionality, that may lead to configuration errors when done manually. Therefore, cloud environments selection and configuration tools and approaches have been developed, ranging from ad-hoc implementation software, to automated strategies based on model transformation. However, these approaches suffer from a lack of abstraction, or do not provide an automated an scalable configuration reasoning support. Moreover, they are often limited to a certain type of cloud environment, thus limiting their efficiency.

To address these shortcomings, and since we noticed that an important number of such cloud environments share several characteristics, we present in this thesis an approach based on software product line principles, with dedicated variability models to handle cloud environments commonalities and variabilities. Software product lines were defined to take advantage of commonalities through the definition of reusable artifacts, in order to automate the derivation of software products. In this dissertation, we provide in particular three major contributions. First, we propose an abstract model for feature modeling with attributes, cardinalities, and constraints over both of them. This kind of feature models are required to describe the variability of cloud environments. By providing an abstract model, we are thus implementation-independent and allow existing feature modeling approaches to rely on this model to extend their support. As a second contribution, we provide an automated support for maintaining the consistency of cardinality-based feature models. When evolving them, inconsistencies may arise due to the defined cardinalities and constraints over them, and detecting them can be tedious and complex whenever the size of the feature model grows. Finally, we provide as third contribution SALOON, a platform to select and configure cloud environments based on software product line principles. In particular, SALOON relies on our abstract model to describe cloud environments as feature models, and provide an automated support to derive configuration files and executable scripts, enabling the configuration of cloud environment in a reliable way.

The experiments we conducted to validate our proposal show that by using software product lines and feature models, we are able to provide an automated, scalable, practical and reliable approach to select and configure cloud environments with respect to a set of requirements, even when numerous different kind of these environments are involved.

## Résumé

Ces dernières années, l'informatique dans les nuages est devenu une tendance majeure dans les environnements distribués, permettant la virtualisation des logiciels sur des environnements d'exécution configurables. Ces environnements fournissent de nombreuses ressources hautement configurables à des niveaux de fonctionnalité différents, pouvant mener à des erreurs lors de la configuration si cette dernière est effectuée manuellement. Des approches permettant la sélection et la configuration de tels environnements ont donc été développées, allant de logiciels basés sur une implémentation ad-hoc à des stratégies automatisées basées sur des transformations de modèles. Cependant, ces approches souffrent d'un manque d'abstraction, ou ne fournissent pas un support automatisé et extensible pour raisonner sur de telles configurations. De plus, elles sont souvent limitées à un certain type de nuage, restreignant ainsi leur efficacité.

Pour faire face à ces limitations, et puisque nos travaux ont montré qu'un important nombre de caractéristiques est partagé parmi ces environnements de nuage, nous présentons dans cette thèse une approche basée sur les principes des lignes de produits logiciels, reposant sur des modèles de variabilité dédiés à la description des similitudes et de la variabilité de ces environnements de nuage. Les lignes de produits logiciels ont été élaborées pour tirer profit des similitudes à travers la définition d'artefacts logiciel réutilisables, afin d'automatiser la génération de produits logiciels. Dans cette dissertation, nous proposons notamment trois contributions essentielles. En premier lieu, nous fournissons un méta-modèle permettant de décrire des modèles de caractéristiques étendus avec des attribuées, des multiplicités et des contraintes s'appliquant sur ces extensions. Ce type de modèle de variabilité est nécessaire à la description de la variabilité des environnements de nuage. En fournissant un modèle abstrait, nous somme donc indépendant de toute implémentation et permettons ainsi aux approches existantes de s'appuyer sur ce modèles pour étendre leur support. Deuxièmement, nous proposons un support automatisé pour maintenir la cohérence des modèles de caractéristiques étendus avec des multiplicités. En effet, lorsque ceux-ci évoluent, des incohérences peuvent survenir dues aux différentes multiplicité définies et aux contraintes sur ces multiplicités. Les détecter peut alors être une tâche complexe, surtout lorsque la taille du modèle de caractéristiques est grande. Enfin, nous fournissons comme troisième contribution SALOON, une plateforme pour automatiser la sélection et la configuration des environnement de nuage basée sur les principes des lignes de produits logiciels. SALOON s'appuie notamment sur notre méta-modèle pour décrire les environnements de nuage en tant que modèles de caractéristiques, et fournit un support automatisé pour générer des fichiers de configurations et scripts exécutables, permettant ainsi une configuration fiable desdits environnements.

Les expérimentations que nous avons conduites pour évaluer notre approche montrent

qu'en utilisant des lignes de produits logiciels et des modèles de caractéristiques, on est capable de fournir une approche pratique, extensible, fiable et automatisée permettant de sélectionner et de configurer des environnements de nuage en finition d'un ensemble d'exigences, et ce même en présence de nombreux et différents environnements.

# Contents

# List of Figures

# List of Tables

# Part I

# Motivation and Context

Chapter 1

# Introduction

**Contents**

In the last years, computing was being transformed to a model consisting of services that are commoditized and delivered in a manner similar to traditional utilities such as water, electricity, gas, and telephony [BYV+09]. Back in 1969, Leonard Kleinrock [Kle03], member of the original *Advanced Research Projects Agency Network* (ARPANET) project, already envisioned such a model: "*As of now, computer networks are still in their infancy, but as they grow up and become sophisticated, we will probably see the spread of 'computer utilities' which, like present electric and telephone utilities, will service individual homes and offices across the country*" [Buy09]. In this model, users access services based on their requirements without regard to where the services are hosted or how they are delivered [BYV+09]. Over the years, several computing paradigms have been proposed promising to achieve this *utility computing* vision, such as cluster computing, grid computing and more recently, cloud computing.

In cloud computing, resources are accessed on demand by customers and are delivered as services by cloud providers in a *pay-per-use* approach, according to service level agreement contracts established between cloud providers and the application's developer. This service provisioning model brings flexibility to those developers that rely on cloud providers' environments to run their applications. However, porting an existing application or deploying a new one to one of these environments is a challenging and error-prone task. Indeed, there exist tens of those cloud environments, providing different supports and services as a wide range of resources at different level of functionality. It is generally up to the developer to specify and exploit these characteristics to her best knowledge, leading to complex choices among cloud environments which are usually made in an ad hoc manner. There is thus a high risk that the results do not meet the expected requirements due to this lack of knowledge. On the other hand, switching between platforms just for testing purposes is too costly [Paa13].

Nevertheless, there exist some techniques to select and configure cloud environments in an automated way. One of these techniques is software product line engineering [CN01]. In software product line engineering, commonalities and variabilities across a set of software products are identified, so that assets can be developed and used to create new different software products, where an asset is any software artifact that can be used in the development of an application [PBL05]. Software product lines are used in a wide variety of domain, like flight programs, engine-control software for gasoline systems or printer firmware [ABKS13]. We consider that the software product line engineering constitutes a suitable candidate to manage the variety of configurations that we find in cloud computing environments. By manage, we mean find a valid configuration, *i.e.*, a proper combination of assets, and rely on this combination to concretely configure the targeted cloud environment. Two essential tasks in software product line engineering can achieve such goals: variability management and product derivation [CN01]. Managing variability is defining the common and variable characteristics, *i.e.*, features, of the product line. One way to document efficiently variability is by means of feature models [KCH+90].

At the same time, there is a strong need to edit those variability models, as they have to co-evolve together with the cloud environments they describe. In the case of cloud environments, feature models can be large and complex, and evolving them manually may be considerably hard and error-prone. In particular, inconsistencies may arise during evolution which are difficult to detect and explain. As a solution to these issues, we thus explore in this dissertation a software product line-based approach that supports the developer when migrating an application to the cloud: from requirements definition, to the selected cloud environment configuration. Our proposal intends to leverage software product line principles, in particular using variability models to describe cloud environments and relying on the derivation process to automate the configuration of the selected environment.

The remainder of this introductory chapter is organized as follows. In Section 1.1 we identify the problems that motivate this research. Next, in Section 1.2 we present our research goals. Section 1.3 explains the contributions described in this dissertation. In Section 1.4, we give a brief introduction to each of the chapters of the document. Finally, we list in Section 1.5 the publications made during the development of our work.

## 1.1 Problem Statement

In this dissertation, we deal with the selection and configuration of cloud environments. More precisely, we state the main problem addressed by this dissertation as follows:

*Given a set of requirements related to the application to deploy, select a suitable cloud environment and propose a proper configuration of this environment to host the application. The requirements taken into consideration are both functional and non-functional ones. The selection of a cloud environment as well as the configuration proposition must be automatically performed.*

Existing approaches aiming at achieving the same objectives are subject to many limitations hampering the efforts for building a well-suited approach. In particular, we have identified that those approaches are confronted to the following issues.

**Lack of selection support.** Numerous cloud environments are available. The first problem faced by developers is therefore to select a cloud for hosting the application to be deployed. This cloud environment must be suitable regarding the application functional and non-functional requirements. The wide range of clouds likely to host the application makes the selection difficult, and there is a lack of visibility among them, in particular without any decision-support tools. This selection is thus limited to the developer's knowledge, which has a limited scope compared to available cloud environments.

**Lack of unified representation.** The selection of a cloud environment is made even more difficult by the diversity and heterogeneity of those environments. Provided services may be described at a low level, *e.g.*, at implementation level using a dedicated API, or conversely relying on a service model describing high level conceptual elements. Moreover, among the provided description mechanisms, there exists no or little consensus such as those cloud description are not standardized yet. This heterogeneity makes the cloud comparison difficult, and therefore the selection as well, in particular regarding non-functional properties, *e.g.*, the price model which is specific to each cloud environment.

**Heterogeneous configuration processes.** Close related to the previously discussed issue is the problem of configuring a cloud environment, as the diversity is not only present at functional level. Several configuration mechanisms are available among existing environments, *e.g.*, through a web configurator, using an API or dedicated SDK or using a general-purpose programming language, which often depend on the underlying cloud infrastructure. This heterogeneity makes in particular the interoperability between cloud environments difficult and therefore limits the migration between them, *e.g.*, when a less expensive configuration is available.

**Lack of evolution support.** Cloud environments evolve over time, *e.g.*, a new database support is provided. The changing nature of these environments and the heterogeneity in terms of provided services make the understanding, the selection and the configuration hardly manageable. This means that the identification and gathering of required configuration elements can be a tedious and error-prone task. Therefore, a global approach for modeling and supporting the evolution of models describing such cloud environments is required.

**Limited automated support.** Once selected, a cloud environment must be properly configured to suit a given set of requirements. Although some of these environments provide support for being - partially or not - configured, the developer sill has to make the configuration choices manually. Moreover, since each environment relies on its dedicated configuration support, it presupposes that the developer masters this technology. Once again, the configuration is error-prone, especially to the lack of knowledge of the developer regarding those heterogeneous environments. Thus, an automated support providing reliable configuration tools is required.

## 1.2 Research Goals

As explained in the previous section, the selection and configuration of cloud environments implies to consider several issues related to variability, representation, evolution and automation. The main objective of this dissertation is thus to provide a solution facing this issues, and we thus propose an approach based on software product line principles. With such an approach, we introduce in particular high-level abstraction of cloud environments, as well as configuration mechanisms using feature models. To achieve this, we decompose this objective in the following goals.

**Describe cloud environments.** First of all, our approach has to provide means to describe cloud environments of different levels of granularity, but using the same formalism. Thus, a unified representation would be used among all cloud environments, helping the developer comparing and selecting one of them.

**Manage clouds variability.** Our approach for modeling cloud environments must also be able to take into consideration commonalities and variabilities across them. This helps identifying and building reusable assets that can be used to yield cloud configurations files, helping once again the developer in its task. Leveraging feature models seems to be a suitable solution, as part of the software product line principles.

**Guarantee environment independence.** As cloud environments are heterogeneous regarding both the functionalities they provide and the mechanisms they rely on to be configured, our approach must provide a way to define the application's requirement and configure the selected cloud without technical or implementation-related considerations.

**Maintain consistency.** Cloud environments evolve over time, as new functionalities are provided or old one are deprecated. Thus, our approach must deal with this evolution and maintain the consistency of the described cloud environments representation, which in turn as to co-evolve with the environment it represents.

**Consider existing approaches.** We aim at constructing our solution for variability management and cloud description relying on existing ones. In particular, our feature modeling approach must leverage abstract models dedicated to feature modeling proposed in the literature. This choice is motivated by our desire to propose an approach that can be applied in different contexts and with different domains, by different feature modeling implementations.

**Provide a simple and flexible solution.** We aim at providing an approach relying on two main concerns: simplicity and flexibility. It must be simple for the developer to handle and apply. Regarding flexibility, the approach must provide means to be extended and maintained over time without difficulties. The application of software product line principles helps us in those purposes.

**Deliver an automated support.** Close related to the simplicity, the proposed approach must be as automated as possible to help the developer. In particular, finding a suitable cloud environments regarding the defined requirement as well as establishing a proper configuration must not be let to the developer.

## 1.3 Contribution

In this section, we present an overview of the contributions described in this dissertation. As stated before, the goal of our research is to provide models, approaches and tools for selecting and configuring cloud environments. The main contributions of our work are summarized as follows.

**Feature Models for Cloud Environments.** Our first contribution is an abstract model for feature modeling, *i.e.*, variability modeling software products sharing commonalities and differentiated by their variabilities using a feature-based approach. This abstract model enables the definition of feature models with attributes and cardinalities. Moreover, constraints over both attributes and cardinalities can be defined. Such feature models are well-suited to describe cloud environments, *e.g.*, number of instances to run or amount of resources required. By providing an abstract model, we are thus implementation-independent and allow existing approaches to rely on this model for both modeling feature models and reasoning on their configurations. Indeed, we provide in addition rules to translate this abstract model into constraints satisfaction problem, thus enabling existing tool support to automatically reason on the defined feature model.

**An Evolution and Consistency-Checking Support.** As a second contribution, we provide an automated support for maintaining the consistency of cardinality-based feature models. Indeed, even though extending feature models with cardinalities is common, little is known about their evolution. We thus show how, when evolving them, inconsistencies may arise due to the defined cardinalities and constraints over them. Moreover, detecting them can be a tedious and complex task, in particular when the size of the feature model grows. Therefore, relying on a formal description of cardinality inconsistencies, we provide an automated support for detecting them end explaining how and why they arose. This support provides a better understanding of the evolution scenario, and especially of the particular edit that led to the inconsistency.

**The SALOON Platform.** Finally, we provide as third contribution SALOON, a platform to select and configure cloud environments based on software product line principles. In particular, SALOON relies on our abstract model to describe cloud environments as feature models, and integrates our consistency-checking support as well. SALOON provides an interface to define the application's requirements, and then automatically searches for a suitable configuration among the described feature models. Finally, the platform leverages the software product line derivation process to derive automatically configuration files and executable scripts regarding a valid feature model configuration. SALOON thus acts as a black-box from the user's perspective, who retrieves those configuration files according to the defined requirements. The configuration of cloud environment is thus done in a reliable way.

## 1.4 Dissertation Roadmap

The dissertation is divided in five parts. While this introductory chapter is part of the first part, the second one encloses the State of Art. The third part presents the contribution of this dissertation, and the fourth one the validation of our proposal. Finally, the last part includes the conclusions and perspectives of this dissertation. Below, we present an overview of the chapters that compose the different parts.

**Part II: State of the Art**

**Chapter 2: Background and Concepts.** In this chapter, we give a brief introduction to the Cloud Computing, Software Product Lines and Feature Models domains. Since used throughout the dissertation, the idea of the chapter is to provide a better understanding of these background and context concerns in which our work takes place, as well as the terminology and concepts presented in the later chapters.

**Chapter 3: Modeling, Selecting and Configuring Cloud Environments.** In this chapter, we list and describe some of the most relevant works related to cloud environments selection and configuration. We compare these related works using different criteria and highlight the benefits of our approach based on software product lines. Then, we study existing approaches for feature modeling cloud environments and describe what is lacking for such a purpose.

**Part III: Contribution**

**Chapter 4: Cardinality and Attribute-based Feature Models.** In this chapter, we present our approach for feature modeling with attributes, cardinalities and constraints over both of them. In particular, we describe our abstract model and we formally define the semantics of such feature models. We also show how translating these feature models into a constraints-based formalism to automate the reasoning on their configurations.

**Chapter 5: Evolution and Consistency Checking of Cardinality-based Feature Models.** In this chapter, we show that evolving a cardinality-based feature model is error-prone regarding the consistency of its defined cardinalities. We thus present common evolution scenarios that may lead to cardinality inconsistencies. Then, we describe two kinds of cardinality inconsistencies, and present a formal approach to detect and explain such inconsistencies. Finally, an automated support providing detection and explanation mechanisms is proposed.

**Chapter 6: SALOON, a Platform for Selecting and Configuring Cloud Environments.** In this chapter, we present our implementation to select and configure cloud environments, called the SALOON framework. Based on software product lines, the platform relies on the approach described in Chapter 4 for feature modeling those environments. Then, configuration files and executable scripts are generated to automate the configuration of the selected cloud environment.

**Part IV: Validation**

**Chapter 7: Validation.** In this chapter, we describe the implementation details of SALOON. We also report on some experiments we conducted to evaluate the platform. This evaluation investigates in particular the soundness, the scalability and the practicality of our approach when dealing with numerous cloud environments. Overall, as our empirical evaluation shows, we observe that SALOON is well-suited to handle the configuration of those cloud environments.

**Part V: Conclusion**

**Chapter 8: Conclusion and Perspectives.** This chapter concludes the work presented in this dissertation. We summarize the overall approach and discuss about some limitations that motivate new ideas and future directions as short-term and long-term perspectives.

## 1.5 Publications

We present below the list of research publications related to the work done while developing the approach described in this dissertation.

**International Journal**

- Clément Quinton, Daniel Romero and Laurence Duchien. *SALOON: A Platform for Selecting and Configuring Cloud Environments*. Submitted to Software: Practice and Experience journal (SPE) on June 2014, accepted with minor revisions on September 2014.

**International Conferences**

- Clément Quinton, Andreas Pleuss, Daniel Le Berre, Laurence Duchien and Goetz Botterweck. *Consistency Checking for the Evolution of Cardinality-based Feature Models*. In Proceedings of the 18th International Software Product Line Conference, SPLC'14 (Acceptance Rate: 28 %). Florence, Italy, September 2014.
- Clément Quinton, Daniel Romero and Laurence Duchien. *Automated Selection and Configuration of Cloud Environments Using Software Product Lines Principles*. In Proceedings of the 7th IEEE International Conference on Cloud Computing, CLOUD'14 (Acceptance Rate: 20 %). Anchorage, Alaska (USA), June 2014.
- Clément Quinton, Daniel Romero and Laurence Duchien. *Cardinality-Based Feature Models With Constraints: A Pragmatic Approach*. In Proceedings of the 17th International Software Product Line Conference, SPLC'13, pages 162-166 (Acceptance Rate: 33 %). Tokyo, Japan, August 2013.
- Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire and Philippe Merle. *Feature Model Differences*. In Proceedings of the 24th International Conference on Advanced Information Systems Engineering, CAiSE'12, pages 629-645 (Acceptance Rate: 14 %). Gdansk, Poland, June 2012.

**International Workshops**

- Daniel Romero, Simon Urli, Clément Quinton, Mireille Blay-Fornarino, Philippe Collet, Laurence Duchien and Sébastien Mosser. *SPLEMMA: A Generic Framework for Controlled Evolution of Software Product Lines*. MAPLE/SCALE workshop, in Proceedings of the 17th International Software Product Line Conference, Volume 2, SPLC'13, pages 59-66. Tokyo, Japan, August 2013.

- Clément Quinton, Nicolas Haderer, Romain Rouvoy and Laurence Duchien. *Towards Multi-Cloud Configurations Using Feature Models and Ontologies*. In Proceedings of the 1st International Workshop on Multi-Cloud Applications and Federated Clouds, Multi-Cloud'13, pages 21-26. Prague, Czech Republic, April 2013.

- Clément Quinton, Patrick Heymans and Laurence Duchien. *Using Feature Modelling and Automations to Select among Cloud Solutions*. In Proceedings of the 3rd International Workshop on Product Line Approaches in Software Engineering, PLEASE'12, pages 17-20. Zurich, Switzerland, June 2012.

- Clément Quinton, Romain Rouvoy and Laurence Duchien. *Leveraging Feature Models to Configure Virtual Appliances*. In Proceedings of the 2nd International Workshop on Cloud Computing Platforms, CloudCP'12, pages 1-6. Bern, Switzerland, April 2012.

# Part II

# State of the Art

# Chapter 2
# Background and Concepts

**Contents**

## 2.1 Introduction

In this chapter, we discuss different domains and concepts applied in our proposal, including Cloud Computing, Software Product Line Engineering and Feature Modeling. The objective of this chapter is not to present an in-depth description of all the existing approaches and technologies surrounding these concerns, but to give a brief introduction to these concerns, used throughout the dissertation. This introduction aims at providing a better understanding of the background and context in which our work takes place, as well as the terminology and concepts presented in the next chapters.

The chapter is structured as follows. Section 2.2 explains the basics of the cloud computing. In Section 2.3, we present the main concepts of software product line engineering. Section 2.4 describes the principles and notations of feature models. Finally, Section 2.5 summarizes the ideas presented in this chapter.

## 2.2 Cloud Computing

Cloud computing has recently emerged as a major trend in distributed computing. It is a natural evolution of the widespread adoption of multiple technical advances in distributed computing including virtualization, grid computing, autonomic computing, utility computing and software-as-a-service.

### 2.2.1 Definition

Several attempts have been done both from academic and industrial domains to define the main characteristics of the cloud computing. We give below the most commonly used definitions:

- *"Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the infrastructure provider by means of customized service-level agreements"* [VRMCL08].
- *"Cloud is a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers"* [Buy09].
- *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (*e.g.*, networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"* [MG11].

To summarize, cloud computing is based on the virtualization of computing resources, at different levels of functionalities, that can be configured according to the user needs. More precisely, the cloud computing is a model composed of five essential characteristics, three service models, and four deployment models, as depicted by Figure 2.1 [MG11]. We describe in the following sections these different concerns.

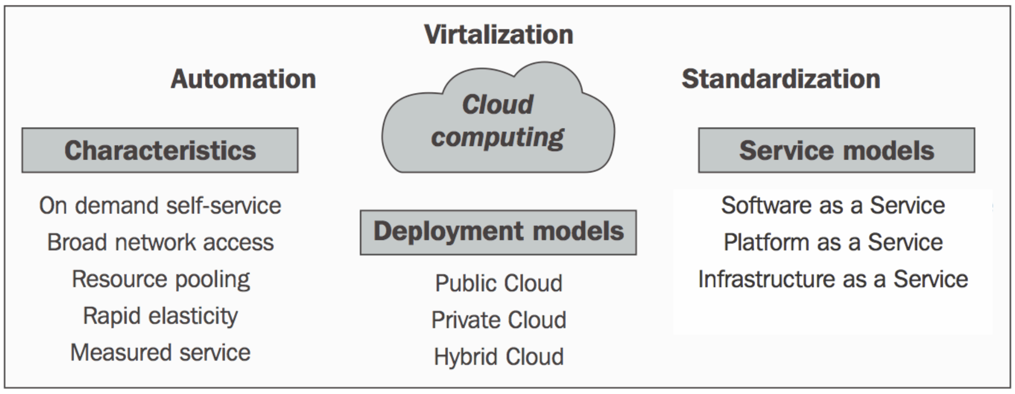


Figure 2.1: Cloud computing overview

### 2.2.2 Essential Characteristics

In practice, there are five essential characteristics that must exist for a computing infrastructure to be characterized as a cloud [MG11].

1. *On-demand self-service*: consumers are able to provision cloud computing resources, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.

2. *Broad network access*: cloud computing resources are accessible over the network through standard mechanisms that promote use by heterogeneous client platforms such as mobile phones, tablets, laptops, or workstations.

3. *Resource pooling*: the provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. Examples of resources include storage, processing, memory, or network bandwidth.

4. *Rapid elasticity*: capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly on demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

5. *Measured service*: cloud systems automatically control and optimize resource use by leveraging a metering capability. Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

### 2.2.3 Service Models

The different cloud computing service models are usually described as *Anything as a Service* (XaaS or *aaS), where *anything* is divided into layers from *Infrastructure* to *Software* including *Platform*, as depicted by Figure 2.2.

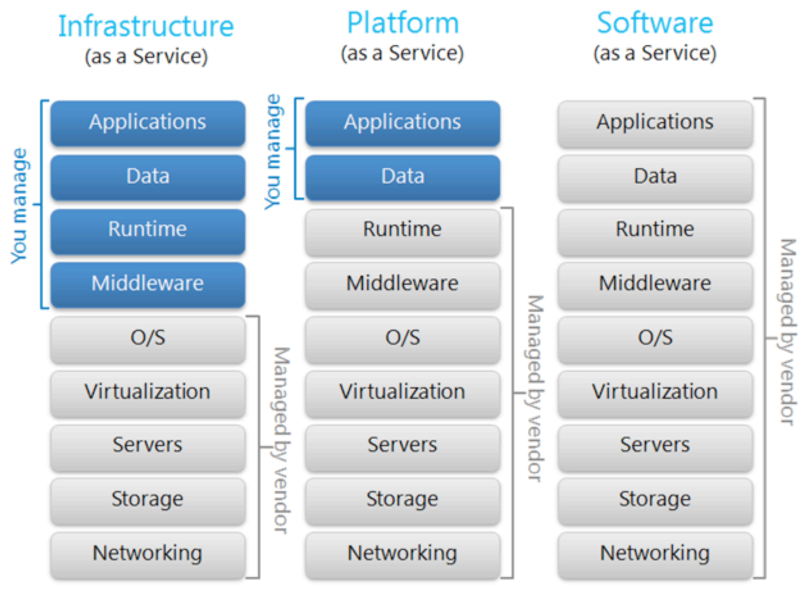


Figure 2.2: Cloud computing service models

1. *Infrastructure-as-a-Service (IaaS)*: the IaaS model provides automation in the lower layers (up to the virtualization layer). Thus, the entire software stack running inside the virtual machine must be configured as well as the infrastructure concerns: number of virtual machines, preferred operating system (selected among those provided by the IaaS), amount of resources, number of nodes, SSH access, database configuration, etc. For instance, Amazon EC2 [ama] or Windows Azure [Win] are considered as IaaS providers.

2. *Platform-as-a-Service (PaaS)*: at this layer, the whole software stack needed by the application is managed by the PaaS provider, *i.e.*, patching and updating the operating system, installing and maintaining middleware as well as the runtime the application uses. Thus, configuring a PaaS to deploy an application is restricted to software that compose this platform: which database(s), application server(s), compilation tool, libraries, etc. For instance, Google App Engine [GAE], Jelastic [jel] or Heroku [her] are PaaS providers.

3. *Software-as-a-Service (SaaS)*: the SaaS model is very similar to the off-the-shelf software model where we go and buy the CD (or download the software), install the software and start using it, but the software is virtualized and accessible through a web browser.

For instance, Google Docs, for editing documents, or Gmail, a mail server, are provided as SaaS. This model relies on the principles of multi-tenancy, where a single application is shared across several users, and leverages the computational power provided by the underlying cloud provider.

### 2.2.4 Deployment Models

There exist four different deployment models, depending on the purpose of cloud services, their physical location or their distribution. These models can be classified as follows.

1. *Private cloud*: the cloud infrastructure is provisioned for exclusive use by a single organization. It may be owned, managed, and operated by the organization, a third party, or a combination of them. This infrastructure is usually dedicated to a small group of users, *e.g.*, the employees of a company. For instance, building a private cloud can be done relying on technologies such as Eucalyptus [euc] or OpenStack [ope].

2. *Community cloud*: the cloud infrastructure is provisioned for exclusive use by a specific community. Persons who are part of this community may be from different organizations, but sharing concerns , *e.g.*, collaborative mission, security requirements or policy. It may be owned, managed, and operated by one or several organizations member of the community, a third party, or a combination of them.

3. *Public cloud*: this kind of deployment model is the most known one. The cloud infrastructure is provisioned for open use by anybody interested in, and is managed by any academic, government or business organization. For instance, Amazon EC2 [ama] or Windows Azure [Win] are public clouds.

4. *Hybrid cloud*: an hybrid cloud is the combination of two or more cloud infrastructures, either private, community or public. Such a model is useful when an organization requires an important resources amount to compute information about private data. A public and a private cloud can thus be bound together, one dedicated to computation tasks (public), the other one to store data (private).

### 2.2.5 Summary

As shown in this section, there exist several kinds of deployment and service models. Each of them provides a large pool of configurable resources that can be used to run an application and make it available as a SaaS. However, those resources are provided independently from each other and still have to be properly configured regarding the application's requirements. The configuration of such resources is complex and error-prone, as many of them may be involved at different layers of service. In this dissertation, we rely on software product line engineering to automate such a configuration. Next section thus describes the principles of such a process.

## 2.3 Software Product Lines

There exist several models of software development processes, *e.g.*, the V-model, the spiral model or the incremental model. Each of these models describes the different tasks or activities that take place during the process, required to build the final software. For instance, a traditional development process usually starts with the analysis of the customer's requirements, followed by several phases such as planning, implementation, testing and deployment. The aim of these processes is to develop one single software system at a time. By contrast, software product line engineering aims at building several similar software systems from a set of common elements [CN01, PBL05].

### 2.3.1 Principles and Benefits

A software product line is a product line (or production line) applied to a software product. Product lines rely on the concept of *mass customization*, which is a large-scale production of goods tailored to individual customer's need [Dav87]. Famous examples from the industry are Boeing, Ford or Nokia, for building aircrafts, cars and cell phones respectively relying on product lines. Boeing, for example, developed the 757 and 767 aircrafts in tandem, as they share up to 60% of common parts, thus reducing production costs at different phases such as assembling and maintenance [CN01]. Based on the same principles, and with regard to software engineering, software product line engineering aims at building software products by exploiting their commonalities. A well-known definition for software product lines is the one given by Clements *et al.* [CN01]: *"A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"*.

Software product line engineering thus focuses on the production and maintenance of multiple similar software products by reusing common software artifacts, or *assets* in the context of software product lines. With mass customization, software *reuse* is a main characteristic of software product line engineering. Back in 1976, Parnas's seminal paper on product families instilled the idea that similar programs could be treated as a family rather than as a separate and unrelated set [Par76]. Later, software reuse was defined as the process of creating software systems from existing software rather than building software systems from scratch [Kru92]. Contrarily to old reuse strategies, *e.g.*, reusing relatively small pieces of code or copy-pasting code designed for use in one system to another one, software product lines make software reuse easier since developed products are from the same family, or *domain*. Reuse indeed works best in families of related systems, and thus is domain dependent [Gla01].

Several benefits motivate the use of product line engineering to develop software systems. Thus, due to systematic reuse, software product lines are expected to improve the software quality and reduce both the time to market and development costs [CN01, PBL05].

Indeed, shared software artifacts are tested in many products. There is thus an higher chance of detecting bugs and correcting them. Then, even if the time to market is initially higher than for a traditional development process, it is then considerably shortened as many software artifacts can be reused for each new product. Finally, producing several software products significantly decreases development costs, as depicted by Figure 2.3.

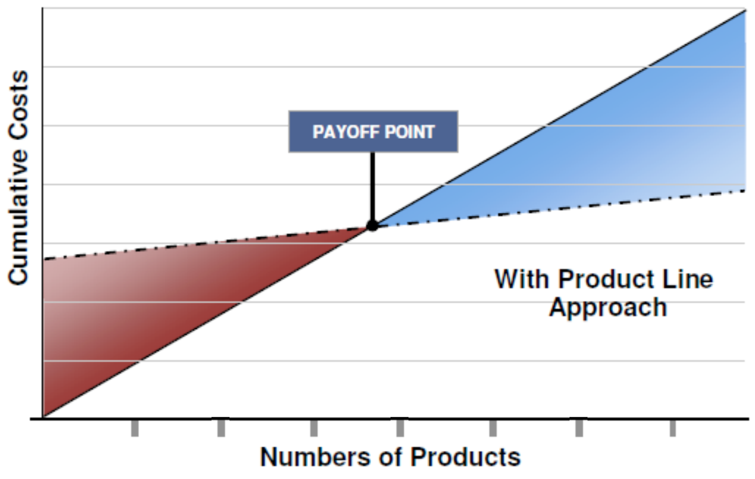


Figure 2.3: Costs for developing with software product lines

The solid line illustrates the costs of developing $n$ kinds of systems as single systems, while the dashed line sketches the costs of developing the products using product line engineering. Developing software using product lines requires a significant up-front investment, due to the creation of the assets that can be reused and planning the way in which they shall be reused. Then, both approaches are equivalent where both curves intersect. This is the payoff point, or break-even point. Empirical evaluations showed that this break-even point is around 3 or 4 products in the case of software systems [WL99, CN01]. For larger quantities of software products, the cost of product line engineering then becomes lower.

### 2.3.2 Software Product Line Processes

Software product line engineering is divided into two complementary processes: *Domain Engineering* and *Application Engineering* [WL99, PBL05]. The former usually refers to development *for reuse* while the latter is the development *with reuse*. In other words, the domain engineering process is responsible for creating reusable assets, while application engineering is the process of reusing those assets to build individual but similar software products. Figure 2.4 depicts those processes.

It is important to note that both processes are complementary, they thus do not follow any specific order. For instance, reusable software artifacts may be built from an existing set

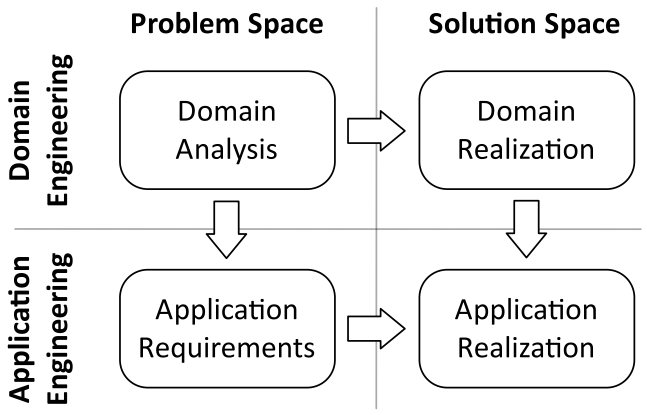


Figure 2.4: Domain and application engineering

of products. In that case, those products are analyzed to isolate software elements which are shared among those products, so that they can be used in the development of other products. Otherwise, software artifacts are built from scratch in order to be reused in several software products. In both cases, the up-front investment done by creating reusable software artifacts or isolating them from existing products (domain engineering) is outweighed by the benefits of then deriving multiple similar software products (application engineering) [DSB05].

### Domain Engineering

The domain engineering process is the process to identify what differs between products as well as reusable artifacts, to plan their development. It thus defines the *scope* of the product line. In particular, the domain analysis phase is responsible for identifying and describing the common artifacts and those that are specific for particular applications. This is the development *for reuse* process, made easier by traceability links between those artifacts [PBL05]. In the domain realization phase, each artifact is modeled, planned, implemented and tested as reusable components.

### Application Engineering

Application engineering is the development process *with reuse*. It is the process of combining common and reusable assets obtained during the domain engineering process. Applications are thus built by reusing those artifacts and exploiting the product line. During the application requirements phase, a product configuration is defined, that fits those requirements. Then, the final software product is built during a product derivation process, which is part of the application realization phase.

- *Product configuration*: this process refers to the selection or deselection of a set of reusable artifacts identified in the domain engineering process. This selection is usually done relying on a *variability model*, which describes the commonalities and differences between potential products at an higher abstraction level.

- *Product derivation*: once a configuration is defined through the variability model, the related software artifacts are given as input to the product derivation process, which in return yields the final software product. This process can be manual or automated, and differs among software product lines. For instance, it can be some source code generation or models composition.

#### Problem Space and Solution Space

Domain and application engineering processes are divided into two spaces, the problem space and the solution space [CE00]. The former includes domain-specific abstractions describing the requirements on a software system. For instance, domain analysis is part of the problem space. On the other hand, the solution space refers to the concrete artifacts of the product line. There is thus a mapping between both spaces, describing which artifact belongs to which requirement or abstraction.

### 2.3.3 Variability Management

As described in the previous section, a central activity to software product line engineering is to define common and variable artifacts of the product line, *i.e.*, manage its *variability*. Managing variability is the key, cross-cutting concern in software product line engineering [CN01, PBL05, CBK13, MP14]. It is also considered as one of the key feature that distinguishes software product line engineering from other software development approaches or traditional software reuse approaches [BFG$^+$02]. Product line variability describes the variation among the products of a software product line in terms of properties, such as features that are offered or functional and quality requirements that are fulfilled [MP14].

#### Variability

An important concern regarding variability is that there exist several notions of variability, and thus several definitions of it. When browsing the literature about variability in software product lines, it may refer to essential and technical variability [HP03], external and internal variability [PBL05], product line and software variability [MPH$^+$07]. These classifications are actually quite similar since they refer to the same concepts. Essential, external and product line variability refer to the variability from the customer's perspective. On the other hand, technical, internal and software variability refer to the software product line engineer's perspective, more interested in implementation details. Moreover, other perspectives have been proposed bout variability. Bachmann *et al.* classify variability into six categories [BB01]. The variability in function, where a particular function may exist in some products and not in others. The variability in data, when particular data structures may vary from on product to another. The variability in control flow, where a particular interaction may occur in some products and not in others The variability in technology, where the platform running the product may vary, *e.g.*, a particular product requires a given library. The variability in quality goals, which may vary according to the customer requirement, *e.g.*, quality or performance. Finally, the variability in environment refers to how a product interacts with its environment. Cross-cutting notions of variability among the previous ones are variability *in time* and variability *in space*, considered as fundamental distinct dimensions in software product line engineering [PBL05, SSA13]. The former is defined as *"the existence of different versions of an artifact that are valid at different times"*, and the latter as *"the existence of an artifact in different shapes at the same time"*. While software product line engineering mainly focuses on dealing with variability in space, variability in time is of prior interest when dealing with the evolution of the product line. According to all those criteria, variability can thus be defined as the combination of the two following definitions: variability is *"an assumption about how members of a family may differ from each other"* [WL99], and *"the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context"* [SvGB05].

As managing variability is a key factor, it must be expressed using a dedicated support. Product line variability is thus documented in so-called *variability models*.

#### Variability Models

One way to document efficiently variability is to model it. Chen *et al.* [CABA09] provide an overview of various approaches dealing with variability modeling. Decision modeling is one mean for variability modeling. A decision model is defined as *"a set of decisions that are adequate to distinguish among the members of an application engineering product family and to guide adaptation of application engineering work products"* [SRG11]. Another way of variability modeling is by mean of orthogonal variability models [PBL05]. In those models, the main concept is the one of *variation points*, which are an abstraction of software artifacts that represent variability. However, and as learned from Chen's survey, most of existing approaches in variability management can be classified (and classify themselves) as feature modeling ones [SRG11].

## 2.4 Feature Models

Using feature models for variability modeling was first introduced back in 1990 by Kang *et al.*, as part of the Feature Oriented Domain Analysis (FODA) [KCH+90]. Since then, several approaches for feature modeling have been proposed, more or less directly derived from the FODA one. Thus, feature models are nowadays considered as the *de-facto* standard for describing variability. In this section, we first explain their notation and principles, and then describe the most well-known extensions for feature models.

### 2.4.1 Notation and Principles

A feature model represents the commonality and variability of a product line in terms of features and relationships among them. Features have been defined as *"a distinguishable characteristic of a concept (e.g., system, component and so on) that is relevant to some stakeholder of the concept"* [CE00]. For instance, Figure 2.5 depicts a simplified feature model inspired from cloud environments.

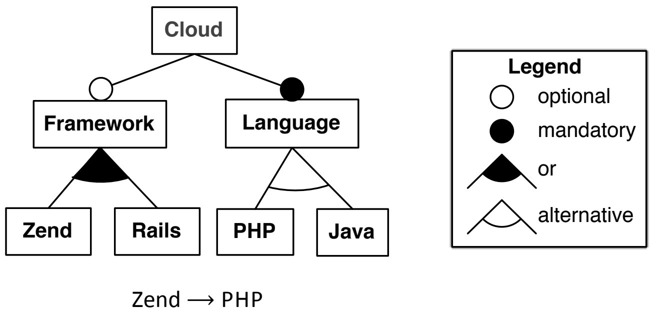


Figure 2.5: A feature model described following the FODA notation

Features, graphically represented as a rectangle, are organized in a tree-like hierarchy with multiple levels of increasing detail. The root feature describes the system under study, *e.g.*, a `Cloud`. Feature are linked together through edges, modeling the parent-child relationship between features. To express the variability of the system, feature models provide two mechanisms. First, a feature may be decomposed into sub-features, where a sub-feature may be *mandatory* or *optional*. For instance, `Language` and `Framework` are mandatory and optional features, illustrated by a black circle and an unfilled one respectively. It is important to note that a feature is mandatory with respect to its parent feature. Indeed, a feature may be modeled as mandatory but not part of the configuration if its parent feature is not part of the configuration itself. The second way to express variability is by mean of *or-* and *alternative*-relationships. In an or-relationship, one or more sub-feature can be selected, while in an alternative-group relationship, exactly one sub-feature must be selected, whenever the parent feature is selected. In addition to the main hierarchy, *cross-tree constraints* can be used to describe dependencies between arbitrary features, *e.g.*, that selecting a feature *implies* the selection of another one or that two features mutually *exclude* each other. For instance, since Zend is a framework dedicated to PHP applications, then the selection of the `Zend` feature implies the selection of the `PHP` one.

*Precision about alternatives.* This relationship is massively denoted as a *xor* relationship in the feature model literature. However, such a notation is semantically wrong. The alternative relationship is satisfied whenever exactly one of the sub-feature is selected. Regarding the XOR operator, if the number of entries set to 1 is odd, then the result is *true*. Applied to feature models, if the number of selected sub-features is odd, then the xor-relationship is satisfied, which is not what is semantically expressed by alternative-relationships. This mistake comes usually from the feature models described as use cases, in which most of alternative-relationships consider only two sub-features. In such a case, both the xor- and alternative-relationships are equivalent.

A feature model defines a set of valid feature combinations, *i.e.*, product configurations. A configuration $c$ is thus defined as a set of selected features $c = \{f_1, f_2, ..., f_n\}$. For instance, {`Language`, `Java`} is a valid product configuration, while {`Zend`} is not, because the `Language` feature is mandatory and must be part of the configuration (together with one of its chill feature, `PHP` in that case due to the related constraint). A feature model is said *void* if it represents no configuration, *valid* otherwise.

### 2.4.2 Feature Model Extensions

Two main extensions have been proposed to add more expressiveness to so-called *basic* or *Boolean* feature models: *attributes* and *cardinalities*. Those extensions are used when the FODA notation is insufficient to effectively model the variabilities of the system under study [CBUE02]. Both of them can be combined in the same feature model, that is, a feature may be extended with both a cardinality and an attribute, as depicted by Figure 2.6, inspired by the Jelastic cloud environments [jel]. However, for the sake of clarity, we describe them separately in the following sections.

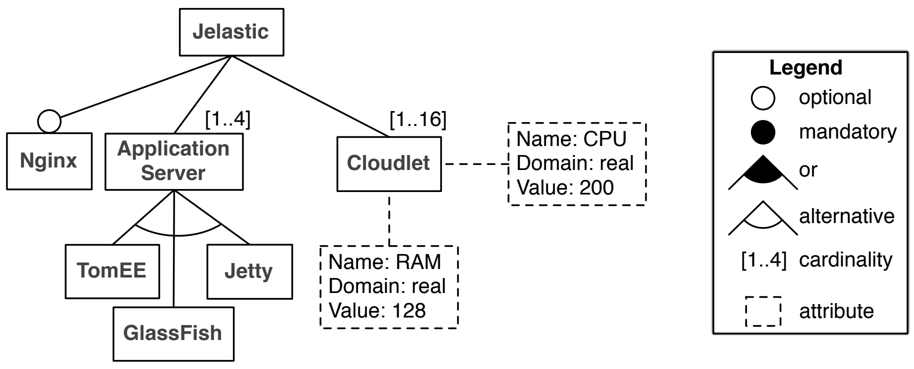


Figure 2.6: A feature model extended with cardinalities and attributes

#### Attribute-based Feature Models

To include more information about features, feature models are extended with so-called *feature attributes* [CBUE02], mainly used to describe non-functional properties of a feature. In

FODA, Kang *et al.* already discussed the addition of information using attributes [KCH+90]. This type of models is referred to as extended, advanced or attributed feature models in the literature [BSRC10]. As highlighted by Benavides *et al.*, there is no consensus on a notation to define attributes [BSRC10]. However, most proposals agree that an attribute should consist of a *name*, a *domain* and a *value* [SRP03, BTRC05, CHE05b, WDS09]. For instance, the `Cloudlet` feature holds two attributes to specify the amount of RAM and CPU provided by this computation block.

### Cardinality-based Feature Models

Cardinality-based feature models support in addition *cardinalities*, which actually refer to two different notions: *feature cardinalities* and *group cardinalities*. Feature cardinalities [CHE05a, CK05], were first introduced as UML-like multiplicities [RBSP02]. A feature cardinality $[m..n]$ can be assigned to any feature except the feature model's root feature. It specifies how many instances (also known as *clones* [MCHB11], *copies* [GR10] or *multi-features* [CSHL13]) of a feature and its subtree can be included in a product configuration with $m$ as lower bound and $n$ as upper bound, and $m \in \mathbb{N}, n \in \mathbb{N} \cup \{*\}$. Regarding the depicted feature model, there can be up to $4$ instances of `Application Server` in a configuration.

On the other hand, a group cardinality is an interval $<m..n>$ defining the minimum and maximum number of sub-features to be selected in an or-relationship [CHE05a]. Or-groups defined in the FODA notation can thus be expressed by the group cardinality $<1..n>$, where $n$ is the number of sub-features, and alternative-groups by $<1..1>$. Note that in the same way as group cardinalities generalize constraints on feature groups, feature cardinalities generalize constraints on single features as they can also be used to express that a feature is mandatory ($[1..1]$) or optional ($[0..1]$).

## 2.5 Summary

In this chapter, we have briefly introduced some principles and basic concepts we will use throughout the dissertation, and given a short explanation of each one of them: Cloud Computing, Software Product Lines, Feature Models.

We have shown that cloud environments provide lots of configurable resources at different level of functionalities, while sharing some commonalities and thus being divided into three main families. To deal with their commonalities, and therefore their variabilities, we introduced feature models and their extensions, and explained the major role they play in the domain analysis phase of a software product line engineering process.

In the next chapter, we present some approaches proposed to deal with cloud environments selection and configuration, and discuss their advantages and drawbacks. We explain how software product lines could be leveraged for automating such processes, and we describe in particular existing approaches for variability modeling.

Chapter 3

# Modeling, Selecting and Configuring Cloud Environments

**Contents**

## 3.1 Introduction

In this chapter, we review approaches, models and tools related to the modeling, the selection and the configuration of cloud environments. The goal of this chapter is to study existing cloud selection and configuration approaches, as well as existing approaches for modeling the variability of cloud environments. We then compare and discuss these approaches in order to outline their limitations.

We start by studying cloud selection and configuration approaches in Section 3.2. Then, in Section 3.3, we argue that software product lines are relevant candidates for modeling and automating the configuration of cloud environments. In Section 3.4, we study and compare variability modeling approaches, required to model cloud environments variability. Section 3.5 present existing approaches dealing with the evolution of feature models, and highlight their limitation when considering feature modeling cloud environments. Finally, we summarize our findings in Section 3.6.

## 3.2 Cloud Environments Selection and Configuration Approaches

In this section, we survey different approaches related to the migration or deployment of applications in cloud environments, thus involving the selection and/or the configuration of such environments. We select in particular five european research projects aiming at deploying application in the clouds: Aeolus [Aeo13], ConPaaS [Con14a], mOSAIC [Mos13], MODAClouds [MOD14] and Remics [REM13], as well as some close-related approaches reported in [MR12, FHS13, WSH+12]. All these approaches propose an automated way to find a cloud configuration fulfilling a set of requirements.

### 3.2.1 Description and Comparison

#### Aeolus

Aeolus is a research project aiming at automating the deployment and reconfiguration of machine pools in the clouds [Aeo13]. From a high level application design, or according to the user's requirements, it proposes a configuration of the targeted cloud environment. This configuration is provided thanks to an abstract model of the cloud, *i.e.*, the Aeolus component model. Thus, cloud environments are described as a set of components, where each component represents a software package, considered as a resource which provides and requires different functionalities, and may be created or destroyed [DCZZ12]. Required components are given as input as constraints. From this set of constraints, a valid configuration can be found relying either on a planning tool [LMZ13] or on Zephyrus, a tool allowing the user to compute a valid configuration satisfying a high level specification [DCLT+13]. Finally, the user gets the configuration as a set of software packages, *i.e.*, a configuration to be executed on top of package-based FOSS (Free and Open Source Software) distributions, *e.g.*, Debian.

#### ConPaaS

ConPaaS is an environment developed as part of the Contrail project [Con14b]. ConPaaS is an open-source Platform-as-a-Service middleware which aims at simplifying the deployment of applications in the Cloud. In ConPaaS, an application is defined as a composition of one or more services, where each service is an elastic component dedicated to the hosting of a particular type of functionality. ConPaaS thus allows the user to configure a Platform-as-a-Service regarding a required set of services. These services are defined using either the ConPaaS web client GUI or by uploading a *manifest* file. A *manifest* specifies the list of services which should be created, the location of code and data that need to be uploaded to each service, and all other configuration information necessary for the good execution of the application [PS12]. Then, the *manifest* (which is also built from the GUI services definition) is parsed using an ad-hoc implementation to provide the final configuration. However, in the current version, ConPaaS only contains nine services, of which only one database (MySQL). Moreover, it can only be deployed on two Infrastructure-as-a-Service, Open Nebula and Amazon EC2.

#### mOSAIC

mOSAIC is an open source platform aiming to simplify the programming, deployment and management of cloud applications, developed as part of the mOSAIC european research project [Mos13]. The mOSAIC platform currently defines a Java API using computing (hardware or software) resources from various cloud providers. This API is designed in order to address applications requirements, such as needs for computational or storage resources as well as various application components that must scale up or down based on current service demand [SJP12]. Thus, the user interacts with the API to define the requirements of the application to deploy. Then, mOSAIC searches for a collection of resources from several cloud providers that are matching the requirements of the application relying on an external *Cloud Broker* [PCN+11]. This cloud broker implements an agent-based approach to find matching services.

#### MODAClouds

The aim of the *MOdel-Driven Approach for design and execution of applications on multiple Clouds* (MODAClouds) project is to provide methods and a decision support system to help developers selecting a cloud provider and support them when migrating an application to the cloud [MOD14]. In particular, CloudML, a Domain-Specific Language (DSL) that facilitates the specification of provisioning and deployment, has been developed as part of this project [FRC+13]. Using CloudML, the developer models the requirements, constraints, and dependencies of the application to deploy into a generic provisioning and deployment model that is independent of the cloud provider. Then, this model is transformed semi-automatically into a specific provisioning and deployment model, dependent of the targeted cloud provider, usually at IaaS level. This requirements definition is done through the MODAClouds IDE (Integrated Development Environment), providing a user interface allowing users to edit and store a CloudML models. The work done in the MODAClouds project actually leverages results of the REMICS project and extends them by providing an additional abstraction level, CloudML [GSK13].

### REMICS

The *Reuse and Migration of legacy applications to Interoperable Cloud Services* (REMICS) project [3] aims at supporting the migration of legacy systems to the cloud based on a Service-Oriented Architecture together with a set of model-driven engineering tools and methods [REM13]. REMICS relies on the concept of Architecture-Driven Modernization (ADM) provided by the OMG [ADM14]. In this concept, modernization starts with the extraction of the architecture of the legacy application. Then, having the architectural model helps to analyze the legacy system, identify the best ways for modernization and benefit from model-driven engineering technologies for generating the new system according to the targeted cloud environment [MS11]. This approach focuses on systems based on the service-oriented architecture.

### CloudGenius

CloudGenius is a framework which automates the decision-making process based on a model and factors dedicated to web server migration to the cloud [MR12]. CloudGenius relies on a multi-criteria decision approach to select an IaaS environment from a set of requirements. These requirements are defined manually, and fit a pattern defined in the CloudGenius Model. The set of requirements is then given as input to a decision-making framework developed by the authors [MSNT11]. Moreover, a weight can be defined for a given criteria to guide the decision. Then, regarding these criteria, an IaaS environment is selected among several ones stored manually and described textually. A tool prototype, named CumulusGenius and used as a Java library, allows the user to programmatically define the requirements. Then, whenever a solution is found, virtual machines can be executed on top of Amazon EC2.

### CloudMIG

The CloudMIG approach aims at supporting developers when migrating software systems to IaaS or PaaS-based cloud environments [FHS13]. In this approach, cloud environments are modeled as instances of a cloud environment metamodel. Configurations of these environments are modeled as well, using the dedicated meta-class. Thus, for each cloud environment, all possible configurations are modeled. A configuration contains in particular a set of elements and constraints across them. CloudMIG takes as input the legacy software system, and extracts its architectural and utilization models based on the architecture-driven modernization principles. From this model, a compatible cloud environment model candidate is selected. Then, CloudMIG relies on its own constraint validators to check the conformance of the legacy software (actually the extracted models) with the candidate cloud environment model in terms of constraint violations, among Google App Engine and Amazon EC2. The authors then extended this framework to improve the search of well-suited IaaS environments using search-based genetic optimization [FFH13].

**Zeel/i**

The approach proposed by Wright *et al.* enables users to match their applications' requirements to infrastructure resources relying on a two-phases resource selection model using a constraints-based approach [WSH+12]. The proposed model provides an application-focused, rather than a provider-focused view of resources. This enables application requirements to be expressed in a domain specific way rather than using the terms used by particular providers. From a set of constraints expressed in a text file, the two-phases resource selection process works as follows. First, constraints are used to find a suitable environment in terms of *must-have* requirements, *e.g.*, a storage support. Then, to improve the filtering of cloud environments, *soft* constraints are checked among the suitable ones, *e.g.*, non-functional requirements such as latency. Once defined, constraints are given as input to a dedicated constraints optimization engine, which tries to find a suitable IaaS environment among those described as *templates* in the Zeel/i framework. Finally, a list of suitable templates is proposed to the user.

**Summary**

Table 3.1 summarizes the study. The first column contains the project or platform under study. There are then three main columns for **Requirement Definition**, **Reasoning Mechanism** and **Cloud** to indicate, respectively, how requirements are expressed, which mechanism is used to reason on those requirements and find a valid configuration, and which are the targeted cloud environments. We use a check mark ✓if the approach proposes solutions or deals with the different criteria, while a check mark between parenthesis (✓) indicates that the approach does not fully handle the criteria.

### 3.2.2 Discussion

We now discuss the advantages of these approaches, and outline their main drawbacks and limitations when selecting and configuring a cloud environment. In particular, we focus on four main criteria: *practicality*, *mechanism*, *target* and *abstraction*. The *practicality* of the approach is important as it needs to hinder the complexity of the cloud systems. Then, when the number of components to deploy grows, it is essential to be able to specify at a certain level of *abstraction* a particular configuration of the distributed software system [DCZZ12]. We also compare the different *mechanisms* used to reason on cloud configurations, while *target* refers to the cloud environments targeted by the approach, whether they are at infrastructure, platform or both levels.

| **Approach** | **Requirement Definition** | | | | | | **Reasoning Mechanism** | | | | | | **Cloud** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Ad-hoc** | **ADM** | **API** | **Constraints** | **DSL** | **Manifest** | **Ad-hoc** | **Agent-based** | **Constraints** | **DSL** | **Planning tool** | **Search-based** | **IaaS** | **PaaS** |
| Aeolus | | | | ✓ | | | | | ✓ | | ✓ | | ✓ | |
| ConPaaS | | | | | | ✓ | ✓ | | | | | | ✓ | (✓) |
| mOSAIC | | | ✓ | | | | | ✓ | | | | | ✓ | |
| MODAClouds | | | | | ✓ | | | | | ✓ | | | ✓ | |
| REMICS | | ✓ | | | | | ✓ | | | | | | ✓ | |
| CloudGenius | ✓ | | | | | | ✓ | | | | | | ✓ | |
| CloudMIG | | ✓ | | | | | ✓ | | ✓ | | | ✓ | ✓ | ✓ |
| Zeel/i | | | | ✓ | | | ✓ | | ✓ | | | | ✓ | |

Table 3.1: Summary of cloud selection and configuration approaches.

### Practicality

When providing an approach to select a cloud environment and find a suitable configuration, one main concern is about its usefulness, which can be seen from two perspectives: the user one and the developer one, *i.e.*, the person who develops and maintains the platform. From the user perspective, the platform must provide means to cope with the heterogeneity and variability of cloud environments, as resources choice can be a complex and error-prone task. Approaches providing an abstract layer to specify application's requirements offer an high degree of flexibility, as they are not loosely-coupled with the cloud environments properties. For instance, Aeolus, MODAClouds and Zeel/i enables the user to specify requirements in a non cloud-specific way, either through a component model [DCZZ12], a DSL [FRC+13] or a dedicated constraints builder [WSH+12] respectively. ADM-based approaches such as REMICS [MS11] and CloudMIG [FHS13] offer an high degree of abstraction as well, but with a limited decisional autonomy. Indeed, requirements are automatically inferred from the application architecture and the user cannot specify additional ones, *e.g.*, maximum configuration cost. Finally, approaches relying on manually-defined requirements are not very practical, especially in a long-term usage when the platform is used several times and requirements must be defined for each deployment. For instance, in mOSAIC, the user interacts with the API to define the requirements of the application to deploy [SJP12]. Within CloudGenius, requirements are defined manually [MSNT11]. In ConPaaS, this is done through a manifest file [PS12], which can however be fulfilled using a web interface.

From the developer perspective, the main objective is to provide a platform that can be easily reused - and therefore maintained - over time, otherwise there is no interest to build such a platform as a manual deployment would be faster. Thus, the architecture of the platform must be flexible and evolutive enough to easily cope with changes in the cloud environments, *e.g.*, evolving existing ones or adding new ones. In that sense, approaches with an ad-hoc reasoning engine are the less practical ones [MS11, MSNT11, PS12, FHS13]. On the other hand, approaches relying on a model-based architecture such as Aeolus [Aeo13], MODAClouds [MOD14] or Zeel/i [WSH+12] can be easily extended with less development efforts.

### Reasoning Mechanism

Close-related to the practicality from the developer perspective is the mechanism used to find a suitable configuration. We previously discussed about its flexibility. We now present the pros and cons of the different approaches regarding the reasoning support. Most of the presented approaches rely on an ad-hoc implementation to reason on cloud configurations [Con14a, REM13, MR12, FHS13]. Such approaches are not likely to scale when handling a significant number of cloud environments, as their are not well-suited to deal with hundreds of requirements and constraints. For instance, ConPaaS is only able to reason on nine cloud services [Con14a]. A cloud service is not even a cloud environment, but a functionality provided by such an environment, *e.g.*, a database support. Regarding CloudGenius, all requirements are defined programmatically using a Java library and, in particular, leveraging enumeration classes to define these requirements [MR12]. Then, an algorithm implemented in the platform searches for a set of suitable virtual machine images, defined programmatically [MR12]. Such an approach is considerably error-prone as there is no way to verify that both requirements and cloud environments are well defined. In mOSAIC, cloud providers matching the requirements are searched relying on an external cloud broker [PCN+11]. Although this separation of concern brings flexibility to the platform, it also implies the requirements to be expressed in a given way and, moreover, that the performance of the platform is tightly-coupled to the one of the related cloud broker. For instance, this can lead to obvious problems in case this broker is not working anymore. In the MODAClouds project, the reasoning engine relies on the internal CloudML DSL [FRC+13]. More precisely, CloudML transforms semi-automatically a generic provisioning and deployment model defining the requirements into a cloud specific model. Although the model-driven approach brings reliability in the reasoning operation, this process requires however to be partially completed manually. Moreover, transformation rules must be well-formed to ensure a correct cloud-specific model regarding the defined requirements. In CloudMIG, the initial reasoning engine was an ad-hoc implementation as well. Then, the search for cloud environment has been improved using a search-based genetic optimization engine, providing better results in terms of feasibility and scalability [FFH13]. Finally, two approaches rely on a constraints-based reasoning engine, Zeel/i [WSH+12] and Aeolus [Aeo13]. The main benefits of such approaches are to be independent from the reasoning engine (or constraints solver) and to hide the complexity of finding a configuration. Indeed, once constraints are defined, the reasoning process is let to the underlying tool support. On the other hand, if such a support is not properly implemented, this can lead in particular to scalability issues. For instance, Zeel/i relies on a reasoning engine develop in an ad-hoc manner, and scalability is likely to be a problem when a large number of infrastructure providers are considered [WSH+12].

#### Target

When deploying an application on the cloud, either a PaaS or a IaaS environment can be configured. For the former, only cloud services provided by the PaaS environment can be configured, *e.g.*, Tomcat for the application server support. For the latter, a virtual machine must be launched, and any software can be configured on top of it, from the operating system to the required libraries. All described approaches have been developed to deal with IaaS environments. The main reason is that IaaS environments have shown that this type of environment offers more flexibility than PaaS ones. The problem with the PaaS solutions is that there is a significant dependence between the services these platforms offer, as explained in the REMICS project objectives [MS11]. The only one approach targeting both PaaS and IaaS environments in its experimentations is the CloudMIG one (Google App Engine and Amazon EC2 respectively) [FHS13]. Both environment configurations have been modeled and can thus be used in the platform. We thus think that other approaches such as Zeel/i or CloudGenius could target PaaS environments as well, since they could be described in the platforms using the dedicated approach, either as a template or a text file respectively [WSH+12, MR12]. Regarding ConPaaS, the proposed platform does not really target PaaS environments, but configures such an environment on top of a IaaS one. This PaaS environment is suitable regarding the application requirements. However, the approach is limited, as only nine services are used to configure those PaaS environments, and only two IaaS are considered for the deployment, Open Nebula and Amazon EC2. The Aeolus project shares the same idea: build a software stack hosting the application on top of a virtual machine [Aeo13]. The approach is slightly different, as any existing software can be used to configure the environment, on condition that this software can be managed by package-based FOSS distributions.

#### Abstraction

To cope with the complexity of cloud environments, it is essential to specify a certain level of abstraction. Some approaches, such as Zeel/i [WSH+12] and CloudGenius [MR12], does not provide such an abstraction, as cloud environment configurations are defined manually in a textual description. In the mOSAIC project, an API is provided to address such an abstraction layer [Mos13]. The advantage of this low-level programming interface is to bring flexibility to the approach, as it is thus not tied to a specific implementation language. Other approaches rely on a model-based abstraction to describe cloud environments and their configuration, either with architecture-driven modernization [FHS13, REM13], a DSL [FRC+13] (actually a DSML, *Domain-Specific Modeling Language*, ) or a component-based model [Aeo13]. Although model-based approaches are well suited to provide an abstraction layer, the main issue that may arise is the granularity level considered in those models. Indeed, a model could be either too coarse-grained or fine-grained, thus not enabling the modeling of low level functionalities or, on the contrary, being to specific. However, the proposed approaches cope with such issues as models can be refined in the model-driven engineering process of ADM approaches, the DSL is implemented to fit a certain level of abstraction, and component-based approaches rely on the principles of *component/composite* (a composite can contain several components) to define several levels of granularity.

### 3.2.3 Summary

We now summarize our study of strengths and weaknesses of existing approaches related to the selection and configuration of cloud environments. Table 3.2 synthesizes the results of this study, divided into four criteria: *flexibility*, *maintainability*, *abstraction* and *heterogeneity*. By *flexibility*, we refer to the way requirements are defined, as well as to the independence of the reasoning process from the rest of the platform. As cloud environments evolve over time, the approaches have to evolve as well to remain well-suited. We thus indicate whether these approaches offer such *maintainability* support or not. We then illustrate if the considered approach provide *abstraction* mechanism to cope with the complexity of cloud environments. Finally, we indicate whether the approach deals with the *heterogeneity* of cloud environments, *i.e.*, if it target different level of the cloud service model, *e.g.*, PaaS and IaaS. We use a check mark ✓if the approach proposes solutions or deals with the different criteria, while a check mark between parenthesis (✓) indicates that the approach does not fully handle the criteria.

As shown by Table 3.2, there exists currently no approach which, at the same time, provides a correct abstraction level to define cloud environments, is flexible enough to handle requirements and reason on them while being easily evolved, and targets both Iaas and PaaS environments. In the next section, we propose one possible way to deal with these challenges.

## 3.3 Software Product Lines for Cloud Environments Configuration

Based on our related works study, we argue that an approach based on software product line principles is well-suited to select and configure cloud environments. Leveraging software product line engineering has three major benefits.

| Approach | Flexibility | Maintainability | Abstraction | Heterogeneity |
|---|---|---|---|---|
| Aeolus | ✓ | (✓) | ✓ | |
| ConPaaS | (✓) | | | (✓) |
| mOSAIC | (✓) | (✓) | | |
| MODAClouds | (✓) | (✓) | ✓ | |
| REMICS | | ✓ | ✓ | |
| CloudGenius | (✓) | | | (✓) |
| CloudMIG | | (✓) | ✓ | ✓ |
| Zeel/i | (✓) | | | (✓) |

Table 3.2: Synthesis of approaches for cloud selection and configuration.

First, as explained in Chapter 2 Section 2.3, a central activity to software product line engineering is variability modeling. Thus, variability models, and feature models in particular, can be used to model cloud environments. There are two benefits from relying on feature models to model cloud environments. First, it provides a mean to model a system at any level of granularity. Indeed, features are organized hierarchically in the tree, enabling the description of a potentially large number of concepts (*i.e.*, features) into multiple levels of increasing detail [Ach11]. Second, complex variable functionalities and dependencies between them can be handled. Addressing such a concern is very important, as Van der Aalst showed that handling variability is one of the main challenges to support configurable cloud services [VDA10].

Second, to reason on the configurations of the software system under study and thanks to their abstraction level, feature models are independent from any reasoning engine. Usually, feature models and their configurations are translated into a specific representation or paradigm such as propositional logic, constraint programming, description logic or ad-hoc data structures [BSRC10]. Then, off-the-shelf solvers or specific algorithms are used as reasoning support to automatically analyze the feature model given as input in terms of *analysis operations*. Regarding the operation to be performed, *e.g.*, give the number of products or is the product valid, different reasoning supports can be used to focus on a given criteria *e.g.*, improve the scalability of the platform to perform a significant number of operations.

Finally, once a configuration is found by the reasoning support, a software product line based approach could benefit from the derivation process. Indeed, such an approach could not only find a correct configuration, but derive such a configuration. For instance, the derivation process could leverage existing cloud configuration tools and mechanisms provided by cloud platforms to configure the cloud environment regarding the configuration found in the previous stage. From an implementation perspective, this derivation process could be the building and generation of configuration files, or the execution of configuration commands provided by the targeted cloud environment, *e.g.*, through an API or a SDK. Such a process brings reliability to the approach, as the configuration only relies on tools used by cloud environment themselves, thus avoiding an ad-hoc and error-prone configuration tool.

In the next section, we study the state-of-the art approaches for variability modeling. We discuss in particular the main characteristics of these approaches, and report the limitation of the existing work regarding modeling the variability of cloud environments.

## 3.4 Variability Modeling Approaches

Over more than two decades, numerous variability modeling techniques have been introduced in academia and industry. In this section, we extend the systematic analysis of textual variability modeling languages proposed by Eichelberger *et al.* [ES13] and survey both textual and graphical variability modeling approaches. In particular, we describe their characteristics, and compare them in terms of both configuration and constraint expression support.

### 3.4.1 Description and Comparison

#### AHEAD

AHEAD (*Algebraic Hierarchical Equations for Application Design*) is an architectural model for feature oriented programming [BSR03]. A variability model in AHEAD is defined using the GUIDSL tool [Don08], where a GUIDSL model is an annotated grammar [Bat05]. The GUIDSL grammar is used as input as a file format by the AHEAD tool suite, which relies on a Java implementation to reason on feature models configurations.

#### Clafer

Clafer (<u>cla</u>ss, <u>fe</u>ature, <u>r</u>eference), is a general-purpose, lightweight textual meta-modeling language, with first-class support for feature modeling [BCW11]. The language, designed to naturally express metamodels, feature models, mixtures of meta- and feature models, aims at providing a common infrastructure for analyses of such feature models and metamodels [Cla14]. Clafer integrates feature modeling into class modeling and thereby can also be used to specify feature models. In particular, Clafer extends the FODA notation so that feature models with attributes can be defined.

### FaMa

FaMa is a Java-based extensible framework for the automated analyses of feature models [TBRC+08, FAM14]. FaMa allows the integration of different logic representations and solvers in order to optimize the analysis process. It can be configured to select automatically the most efficient of the available solvers according to the operation requested by the user [BSTC07]. The implementation is based on an Eclipse plug-in and uses XML to represent feature models so it can interoperate with other tools that support it. FaMa supports attribute-based feature models, as well as the definition of group cardinalities.

### FAMILIAR

FAMILIAR (*FeAture Model scrIpt Language for manIpulation and Automatic Reasoning*) is a domain-specific language for managing feature models [ACLF11a, Mat14]. The aim of FAMILIAR it to provide a support for separating concerns in feature modeling through the provision of composition and decomposition operators *e.g.*, slice or merge, and for reasoning facilities on these feature models. To manage and reason on feature models, FAMILIAR provides an Eclipse-based development environment, allowing the user to define feature models graphically or using the text editor.

### FDL

The first textual language to describe feature models was FDL, back in 2002 [vDK02]. FDL was constructed by applying design principles of domain-specific languages. In addition to the syntax, the authors specify a feature diagram algebra. This algebra is a set of rules for operating on FDL models such as normalization, expansion and satisfaction rules. Further, the authors describe the mapping of FDL models to UML class diagrams for Java code generation.

### FeatureIDE

FeatureIDE is an open-source framework for feature-oriented software development based on Eclipse [TKB+14]. It supports all phases of feature-oriented software development for the development of software product lines: domain analysis, domain implementation, requirements analysis, and software generation [Fea14]. The edition of feature models is graphical or text-based. In addition, it supports edits on feature models, *i.e.*, categorizing edits into refactoring, generalization, specialization or none of these. As several variability modeling implementation techniques are integrated to FeatureIDE, it supports feature model grammars from other approaches described in this survey such as GUIDSL, S.P.L.O.T, Velvet or FMP.

### FMP

The emphFeature Model Plugin (FMP) has also been implemented as an Eclipse plugin, and provides tree views for creating feature models [AC04]. It supports cardinality-based and attribute-based feature modeling, specialization of feature diagrams and configuration based on feature diagrams. It relies on a feature metamodel, represented as a feature model itself and containing definitions of feature diagrams, features, groups, etc. Feature models can be edited manually as XML files or using the dedicated graphical environment. the main purpose of FMP is to support the configuration and specialization of feature models. FMP is no longer maintained by its developers, and has been since replaced by Clafer [FMP14].

### GEARS

GEARS is a commercial tool from BigLever dedicated to the development of software product lines [Big14]. Although not directly focused on variability modeling, GEARS integrates feature modeling and configuration process support. In particular, it supports the definition of feature models that express the full product line feature diversity for all assets in all stages of the system and software development lifecycle, plus one set of feature profiles to describe product instantiations in terms of selected feature options and alternatives from the feature model [KC13]. GEARS models feature declarations as parameters, where different parameter types stand for different kinds of variability, *e.g.*, Boolean for optionality or enumeration for alternatives. Feature assertions describe constraints and dependencies among the feature declarations. Feature assertions in Gears express *requires* or *excludes* relations. BigLever provides a dedicated tool ecosystem integrating GEARS.

### pure::variants

pure::variants is a commercial tool from pure::systems GmbH for variant management of product line based software development [pur14a]. Variability modeling with pure::variants can be done using a web browser, a command line interface or the dedicated IDE. pure::variants supports the definition of attribute-based feature models [pur14b]. It allows modeling global constraints between features and it offers interactive, constraint-based configuration using a Prolog-based constraint solver.

### S.P.L.O.T.

The *Software Product Lines Online Tools* (S.P.L.O.T.) is a web-based tool providing a feature model editor as a tree view, a configuration editor with decision propagation, automated analysis on feature models, and example feature models stored in a feature models repository [spl]. S.P.L.O.T. provides two major services: automated reasoning and product configuration. Reasoning focus on automating statistics computation (*e.g.*, depth of the feature tree, number of features) and critical debugging tasks such as checking the consistency of feature models and detecting the presence of dead and common features. In addition, reasoning supports measuring properties such as the number of valid configurations and the variability degree of feature models [MBC09].

### TVL

TVL (a *Text-based Variability Language*) is a text-based language for feature models with a C-like syntax. The goal of the language is to be scalable, by being concise and by offering mechanisms for modularity, and to be comprehensive so as to cover most of the feature model dialects proposed in the literature [TVL14]. Further goals for TVL are to be lightweight (in contrast to the verbosity of XML for instance) and to be scalable by offering mechanisms for structuring the feature model in various ways [CBH11]. TVL enables the definition of attribute-based feature models, together with related constructs to express constraints over them. The current available version can be run in a command line interface to perform some operations on feature models [TVL14].

### VELVET

Velvet is a language for multi-dimensional variability modeling. It integrates separate modeling and integrated analysis of variability, feature model composition, and configuration [VEL14]. The definition of a variability model with VELVET is similar to a class definition, and the syntax of VELVET uses parts of the syntax of TVL [RSTS11]. VELVET comes as a set of tools to model, compose, and configure feature models, and provides support to read other kinds of variability models such as FeatureIDE or S.P.L.O.T. The main purpose of VELVET is to support separation of multi-dimensional concerns in feature-based variability modeling. VELVET thus allows a stakeholder to decompose the variability model of an software product line into multiple models to handle its complexity.

### VSL

The *Variability Specification Language* (VSL) is part of CVM, a framework for compositional variability management [CVM14]. CVM is implemented as an Eclipse plugin, but stand-alone versions are available too. Feature models can be defined using the graphical editor or as textual VSL specifications, and conform to the CVM metamodel. The syntax of VSL was inspired by programming languages such as Java [APS$^+$10]. It supports in particular the definition of cardinality-based feature models.

### Summary

Table 3.3 summarizes the study. The first column contains the approach under study. There are then three main columns for **Description**, **Extensions** and **Constraints** to indicate, respectively, how variability models are described, which kind of extension is supported, and is there a support for constraints on these extensions. We use a check mark ✓if the approach proposes solutions or deals with the different criteria, while a check mark between parenthesis (✓) indicates that the approach does not fully handle the criteria. In particular regarding cardinalities, it means that only group cardinalities are supported, not feature ones. When an approach does not provide any support for both extensions and constraints, it implicitly means that it addresses basic feature models, *i.e.*, feature models without extensions but supporting Boolean constraints such as *implies* and *excludes*.

| **Approach** | **Description** | | **Extensions** | | **Constraints** | |
|---|---|---|---|---|---|---|
| | **Textual** | **Graphical** | **Attributes** | **Cardinalities** | **Attributes** | **Cardinalities** |
| AHEAD | ✓ | | | | | |
| Clafer | ✓ | | ✓ | | ✓ | |
| FaMa | ✓ | ✓ | ✓ | (✓) | ✓ | |
| FAMILIAR | ✓ | ✓ | | | | |
| FDL | ✓ | | | | | |
| FeatureIDE | ✓ | ✓ | | | | |
| FMP | ✓ | ✓ | ✓ | (✓) | ✓ | |
| GEARS | ✓ | ✓ | | | | |
| pure::variants | ✓ | ✓ | ✓ | | ✓ | |
| S.P.L.O.T. | ✓ | ✓ | | | | |
| TVL | ✓ | | ✓ | (✓) | ✓ | |
| VELVET | ✓ | | ✓ | | | |
| VSL | ✓ | | ✓ | ✓ | | |

Table 3.3: Summary of variability modeling approaches.

All described approaches propose a textual format to define feature models. Some of these approaches are *real* text-based approaches, *i.e.*, address feature modeling using text *e.g.*, Clafer, FAMILIAR, FDL, TVL or VELVET. According to Classen *et al.*, there are several benefits from using a text-based approach, as existing tool support for graphical feature models is lacking or inadequate, and inferior in many regards to tool support for text-based formats [CBH11]. One main reason is the graphical nature of feature models syntax. Almost all existing feature modeling languages are based on the FODA notation which uses graphs with nodes and edges in a 2D space, and becomes difficult to handle and understand when the feature model size grows. Another reason is that most of those textual-based approaches handle feature attributes. Feature attributes are intrinsically textual in nature and do not easily fit into this representation. Furthermore, constraints on the feature models are often expressed as textual annotations using Boolean operators. If they were given a graphical syntax, attributes and constraints would only clutter a feature model [CBH11].

Other approaches do not specifically rely on text-based modeling to describe feature models, but the underlying implementation provides such a support. For instance, FaMa, FMP, pure::variants or S.P.L.O.T propose an XML-based formats to describe feature models. These formats were not intended to be written or read by the engineer and are thus difficult to interpret, mainly due to the overhead caused by XML tags and technical information that is extraneous to the model. The semantics of FaMa and pure::variants is tool-based, given by the algorithms that translate an feature into the dedicated solver formalism, and is thus not readily accessible to an outsider [CBH11].

### 3.4.2 Requirements for Feature Modeling Cloud Environments

Although comparing text and graphical-based approaches is important, the main interest in this study is to discuss relevant concerns with respect to cloud environments modeling and configuration. Recently, some approaches for feature modeling cloud environments have been described [DWS12, WKM12], including ours. Dougherty *et al.* [DWS12] proposed an approach for optimizing the energy consumption of cloud configurations. They thus rely on feature attributes to specify energy amount for each feature, as well as their price. Wittern *et al.* [WKM12] also propose attribute-based feature models to describe cloud environments. For instance, attributes are used to define non-functional properties such as the cost of a feature, or its disk space size. Moreover, constraints on feature attributes values are defined, and a product configuration is thus computed regarding those constraints. We argue that such an extension is required to describe cloud environments and reason on their configurations. Indeed, when configuring a cloud to deploy an application, not only functional concerns are important, but also non-functional ones, such as the price of an element or the provided amount of CPU or RAM, which can themselves impact the total configuration price.

In addition to feature attributes, feature cardinalities are required, as well as constraints over them. Feature cardinalities are used to define the number of elements that must be part of the configuration, *e.g.*, the number of nodes or virtual machines. Moreover, constraints must be expressed in terms of feature instances, *e.g.*, to specify that instances of one feature should be included or excluded from a product, thus reasoning on feature cardinalities. Quantifier over those cardinalities must also be used, *e.g.*, to specify that a feature requires a given amount of instances of another feature. However, none of the existing approaches addresses properly both support for attributes and cardinalities and support for constraints over them. One possible solution would be to extend one of the closest approaches, *i.e.*, the ones that partially address such supports: FaMa, FMP and TVL. Nonetheless, the integration effort would be quite consequent. For instance, FMP has not been maintained for several years now and replaced by Clafer, while the current TVL tool prototype is still a work in progress. Regarding FaMa, the aim of the framework is not to support all feature modeling extensions, but to compare different solvers when performing operations on feature models. Extending the platform would thus not be interesting regarding this objective as it does not bring new operations nor solver support.

One of our goals in this thesis is, therefore, to provide a feature modeling approach handling feature cardinalities and attributes, as well as constraints over them to properly describe cloud environments variability. Nonetheless, our aim is not to develop another feature modeling approach. We will thus focus on extending existing feature modeling abstract syntaxes, *e.g.*, the one proposed by the developers of FaMa [BSTRC06]. An approach with such a level of abstraction will thus be "approach-independent", *i.e.*, implementable by any feature modeling approach. Once defined, cloud feature models need to evolve to be up-to-date with the cloud environment they describe. In the next section, we thus survey existing work dealing with the evolution of feature models.

## 3.5 Approaches for Evolving Feature Models

As software product lines are often a long-term investment, they need to evolve to meet new requirements over many years [BP14]. This is reflected in the need to evolve feature models. The main issue when evolving a model, and feature models in particular, is to maintain its consistency, as problems may arise while editing the model. In this section, we thus survey and discuss different concerns regarding existing approaches for evolving feature models.

### 3.5.1 Description and Comparison

Alves *et al.* investigate issues that need to be addressed when refactoring software product lines, and feature models in particular [AGM+06]. They describe a feature model refactoring as a "*transformation that improves the quality of a feature model by improving (maintaining or increasing) its configurability*". Their main contribution is then to provide a catalog of edits for refactoring feature models, *e.g.*, *add new alternative* or *replace mandatory feature*. Finally, they propose an approach for verifying the correctness of the refactorings. Refactorings are first done at solution space level, by refactoring the original source code of the software product. Then, at problem space level, feature models are generated. If a sequence of refactorings can be applied from the original feature model, *i.e.*, the one enabling the derivation of the original product, to the resulting feature model, then the whole refactoring is considered as correct. Their approach only handles Boolean feature models.

Guo *et al.* also propose an approach to check the consistency of evolving feature models [GWTB12]. Their approach focuses on the changes since the last version of the feature model rather than checking the overall consistency of the resulting feature model. To achieve their goal, the authors formalize feature models from an ontological perspective and analyze the semantics of their changes using ontologies. More precisely, feature models are described as a set of syntactical and semantic consistency constraints, that must hold after changes. Once a change is performed on a feature model, the semantics of the resulting feature model is analyzed to improve the reliability of next changes. Thus, they propose a set of predefined strategies to keep the consistency in error-prone operations. An evolution strategy defines the way a change will be applied, as a list of atomic operations, *e.g.*, *add feature*, *rename feature*.

Lotufo *et al.* study the evolution of a real world variability model, the Linux kernel one [LSB+10]. However, their approach is suitable for any Boolean feature model, as the authors provide rules for translating the Linux kernel variability model, to a feature model. They describe different kinds of operation performed to evolve this variability model, extracted from an analysis of the evolution of the Linux kernel feature model over almost 5 years. In particular, half of the edits are motivated by inclusion of a new functionality. The main objective of this work is to provide an empirical evidence on how a large, real world variability model evolves.

Neves *et al.* propose an approach for dealing with safe evolution of product lines [NTS+11]. By safe, they mean an evolution that preserves the original behavior of the product lines, *i.e.*, products that could be generated before evolution can still be after. Their approach is based on the notion of product line refinement, derived from the program refinement one. As most of product line refinement scenarios described involve many changes both at problem space and solution space level, *e.g.*, evolve code assets and feature model, the refactored product lines generates more products. They thus present several safe evolution templates, described as valid modifications that can be applied to a product line while preserving existing products. These templates can be applied to both feature model and artifacts level, and ensure existing products to be part of the new set of generated products.

Passos *et al.* also focus their work on the evolution of the variability model together with the source code [PCW12, PGT+13]. They extend Lotufo's work, using as example the Linux kernel variability model as well, and describe a catalog of evolution patterns for the co-evolution of both the variability model and the related artifacts. Those pattern were extracted by inspecting over 500 Linux kernel commits spanning almost four years of development. The main contribution of this work is to provide evidence that variability evolution can follow systematic patterns.

Pleuss *et al.* rely on a model-driven support to handle feature model evolutions [PBD+12]. In their approach, feature models are considered as a composition of a set of clustered model fragments. Fragments are a set of feature model elements, *i.e.*, a feature model subtree, that are added or removed during the same evolution step. Both fragments and feature models are instances of their related metamodel. In addition, the authors describe a special kind of feature model, called EvoFM, that specifies dependencies and relationships between fragments. Thus, if a fragment is evolved and has a certain relationship with another fragment, then this other fragment has to evolve as well. The authors also present a set of evolution operators to be performed on the EvoFM, *e.g.*, move a feature below another one or convert an optional feature into a mandatory one.

Seidl *et al.* propose an approach for co-evolving feature models and feature mappings, *i.e.*, mappings from elements in the problem space to elements in the solution space [SHA12]. They thus present evolutions scenarios for both spaces, *e.g.*, duplicate, split, insert and remove a feature or replace and extract method. Evolving a software product line in two different spaces may lead to inconsistencies regarding the mappings. The authors thus present a catalog of remapping operators that preserves the consistency of mappings between spaces, by modifying existing mappings in accordance with the performed model evolution.

Thüm *et al.* [TBK09] describe an approach to analyze edits performed on a feature model and classifies them into *refactoring*, not changing the set of valid products, *generalization*, only adding products, *specialization*, reducing the set of products, or *arbitrary* changes otherwise. Their approach takes as input two feature models (one before and one after edits), where the sets of features in both models is not necessarily the same, and automatically computes the classification. To reason on those classifications, feature models are translated into propositional formula, which in turns are converted to conjunctive normal form to be handled by off-the-shelf SAT solvers.

### 3.5.2 Summary and Discussion

Table 3.4 summarizes the study. The first column contains the approach under study. There are then two main columns for **Evolution Patterns** and **Evolution Analysis** to indicate, respectively, if the proposed evolution patterns (if any) apply to feature models or any other element, and if the authors propose an approach to analyze feature models after evolution, or any other elements. We use a check mark ✓if the approach proposes solutions or deals with the different criteria, while a check mark between parenthesis (✓) indicates that the approach does not fully handle the criteria. Note that all of these approaches only address Boolean feature models.

| **Approach** | **Evolution Patterns** | | **Evolution Analysis** | |
|---|---|---|---|---|
| | **Feature Model** | **Other** | **Feature Model** | **Other** |
| [AGM+06] | ✓ | | | Refactorings |
| [GWTB12] | | | ✓ | |
| [LSB+10] | ✓ | | | |
| [NTS+11] | ✓ | | | Products |
| [PGT+13] | ✓ | | | |
| [PBD+12] | ✓ | | (✓) | |
| [SHA12] | ✓ | Mappings | | Mappings |
| [TBK09] | | | | Products |

Table 3.4: Summary of approaches for evolving feature models.

This survey shows that several works exist on evolving feature models, spanning over different research areas, such as describing evolution patterns [AGM+06, LSB+10, NTS+11, PBD+12, SHA12, PGT+13], reasoning on products evolution [NTS+11, TBK09] or checking the consistency of the feature model [GWTB12, PBD+12]. Even though studying evolution patterns is important to understand how and why feature models evolve, analyzing the behavior of the product line after evolution is fundamental as well. Alves *et al.* verify the correctness of the applied refactorings [AGM+06], while Seidl *et al.* study the mapping operators between problem space and solution space [SHA12]. On the other hand, some authors propose an approach to reason on the set of products for the product line, *i.e.*, is there less, more, or an equivalent set of products that can be generated after evolution [NTS+11, TBK09].

Another major concern when evolving feature models is to check that such evolution do not lead to inconsistencies regarding the feature model. In the case of Boolean feature models, evolutions may lead for instance to dead or false optional features, as well as redundancies. To deal with these issues, surveyed works propose different approaches. Guo *et al.* [GWTB12] rely on semantic constraints to check feature models consistency while Pleuss *et al.* [PBD+12], even though it is not the main objective of their work, ensure feature models to conform to their metamodel before and after evolution.

However, as described in the previous section, feature modeling cloud environments requires both attributes and cardinalities. Although extending feature models with attributes introduces complexity, it does not bring additional risk for the model to be inconsistent. A feature has one or several attributes, or has not. Contrarily to attributes, cardinalities may introduce inconsistencies when evolving the feature model. However, little is known about consistency checking cardinality-based feature models in the current state-of-the-art. To the best of our knowledge, only Trinidad *et al.* [TR09] investigated the consistency of feature models regarding group cardinalities. They thus consider a wrong cardinality "*when a cardinal is never used in any product*". They rely on abductive reasoning to search for inconsistencies and explain their origin.

Thus, since cloud environment feature models are extended with cardinalities, and will evolve over time together with the environment they describe, one of the main goals in this dissertation is to provide a mean to check their consistency. Not only group cardinalities, but feature cardinalities as well, to find whether there exists a value in the cardinality range that is never used in any product. In addition, as cardinality inconsistencies may be complex to detect and understand, our approach must support the developer by automatically finding where and why such an inconsistency arose.

## 3.6 Summary

In this chapter, we reviewed the state-of-the-art approaches for cloud environments selection and configuration. An high degree of abstraction is required to cope with clouds complexity and variability. However, only some of existing approaches offer such an abstraction layer, sometimes not even offering a flexible enough support. In addition, the reasoning engine often relies on an ad-hoc implementation that may cause scalability issues, while the targeted cloud environments are most likely to be IaaS ones.

We thus described how software product line principles could be leveraged to select and configure cloud environments. In particular, there are three main benefits of using such an approach. Variability models, and feature models in particular, can be used to describe clouds variability, while off-the-shelf reasoning engines such as solvers can be used to reason on their configuration, making the approach scalable and flexible. Finally, software product lines bring reliability by automating the cloud configuration through the derivation process.

Since the derivation process is implemented with respect to the assets described for the software product line, and since the reasoning support is externally provided, then only the variability modeling support must be defined. We thus propose a survey on existing variability modeling approaches, and discuss their different modeling and configuration supports. We argue that none of those approaches is well-suited for variability modeling cloud environments, as there exists no one supporting feature cardinalities and attributes, together with constraints over them.

Finally, we showed different approaches for supporting the evolution of feature models. We need such a support as cloud feature models are likely to evolve to stay up-to-date with the cloud environment they describe. However, as explained in the previous section, there exists no approach supporting the evolution of cardinality-based feature models, which is required as cardinalities increase the complexity level of the feature model, leading to inconsistencies during evolution.

Considering the approaches and concepts discussed in this chapter, we describe in the next part of the dissertation our contribution for selecting and configuring cloud environments. We propose in particular an approach leveraging software product line principles that addresses the research goals we discussed in Chapter 1, and highlighted below.

First, in Chapter 4, we propose to **describe cloud environments** using feature models extended with cardinalities, attributes and constraints over them. Using feature models, one can describe cloud environments with different levels of granularity and easily **manage their variability**. As our goal is not to provide another feature modeling approach, we thus **consider existing approaches** and propose to work on the abstract syntax level of feature models, so that those existing approaches could benefit from this work.

Then, Chapter 5 describes our approach for evolving cardinality-based feature models. Indeed, feature models evolve over time, but cardinalities introduce complexity when handling such evolution. We therefore present some evolution scenarios with respect to cardinalities, and describe how and why they may lead to model inconsistency. We thus propose an automated approach to help the developer **maintain consistency** when evolving such feature models.

Finally, we **deliver an automated support** based on software product line principles to select and configure cloud environments. Relying on these principles, we **provide a simple and flexible solution** that can be easily extended, as described in Chapter 6. In addition, our approach **guarantees environment independence**, as it does not require any environment-specific knowledge for the configuration.

# Part III

# Contribution

# Chapter 4

# Cardinality and Attribute-based Feature Models

**Contents**

## 4.1 Introduction

Feature models were first introduced as part of FODA back in 1990 [KCH+90] (see Chapter 2, Section 2.4). Since then, the FODA notation has been extended to improve the expressiveness of feature models. *Cardinality-based* feature models support in addition feature *cardinalities* [CHE05a, CK05], first introduced as UML–like multiplicities [RBSP02]. A feature cardinality [*m*..*n*] can be assigned to any feature except to the feature model's root feature. It specifies how many instances of a feature and its subtree can be included in a product configuration with *m* as lower bound and *n* as upper bound ($m \in \mathbb{N}, n \in \mathbb{N} \cup \{*\}$). Feature instances are also known as *clones* in the literature [CHE05a, MCHB11]. Another well-known extension is by means of features *attributes*, used to describe the non-functional properties of a given feature, *e.g.*, its size, its provided quantity or its energy consumption [CBUE02, BTRC05]. There is currently no consensus on a notation to define attributes, but they have been widely used by several authors to define additional information in feature models. For the sake of simplicity and clarity, feature models extended with *both* cardinalities and attributes will be referred to as CardEx feature models in the rest of the dissertation.

However, even though using feature attributes and cardinalities is well-known when feature modeling, existing approaches do not offer a support for expressing constraint over these extensions and cannot reason in an automated way on the related feature model configurations, as stated in Chapter 3. To deal with this issue, we extend the existing feature model notation and propose an abstract syntax to define CardEx feature models. We formally define the semantics of such feature models and describe how this abstract syntax can be translated and used as input to automate the configuration of CardEx feature models. This chapter thus covers the complete phase of feature modeling, constraint expressing and configuration reasoning.

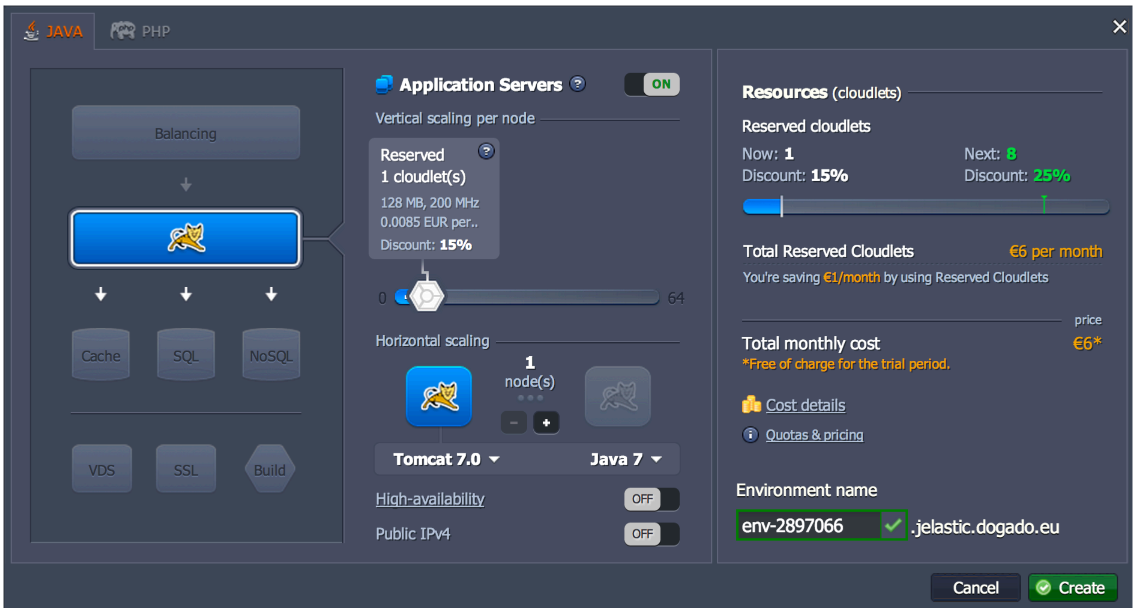
The chapter is structured as follows: Section 4.2 describes a motivating example highlighting the need for expressing constraints over features cardinality and attributes and discusses the related challenges. Section 4.3 presents the CardEx feature model metamodel used as abstract syntax, describes the constraints semantics and explain the translation rules to automatically reason on these feature models. Finally, Section 4.5 discusses several concerns regarding our approach while Section 4.6 concludes the chapter.

## 4.2 Motivation and Challenges

This section introduces a motivating example for CardEx feature models. It illustrates the need for new constraint expressions when using feature models with cardinality and attributes. Based on this example, we then discuss challenges related to modeling CardEx feature models and reasoning on their configurations.

### 4.2.1 Motivating Example

To introduce our approach, let us consider as an example the Jelastic cloud environment [jel]. Jelastic is a PaaS environment, providing support to deploy Java or PHP applications. In this example, the environment is configured to host a Java-based application, as depicted by Figure 4.1, top left corner. When configuring the Jelastic environment, at least one application server instance must run, *e.g.*, the Tomcat 7.0 server, and a language support must be selected. In such a case, Java 7 is required. Then several options are available, such as configuring up to two database supports, either SQL or NoSQL, a Cache service, a *Virtual Dedicated Server* (VDS), a secured access to the deployed application (SSL) or a Maven Build node, as depicted in the bottom left corner. To run the application, Jelastic provides several cloudlet instances. A cloudlet is the computation unit of Jelastic. As described in the center of the figure, it provides 128MB of RAM and 200MHz of CPU power. When describing the Jelastic variability in a feature model, these *non-functional* properties would be defined as feature attributes. There is one cloudlet in the configuration, which is actually due to the implicit constraint that to run an application server, a minimum amount of computing resources is required, that is, at least one cloudlet.



Reserved
1 cloudlet(s)
128 MB, 200 MHz
0.0085 EUR per..
Discount: **15%**

1
node(s)

Tomcat 7.0
Java 7
High-availability OFF
Public IPv4 OFF

**Resources** (cloudlets)
Reserved cloudlets
Now: **1** Next: **8**
Discount: **15%** Discount: **25%**

Total Reserved Cloudlets €6 per month
You're saving €1/month by using Reserved Cloudlets

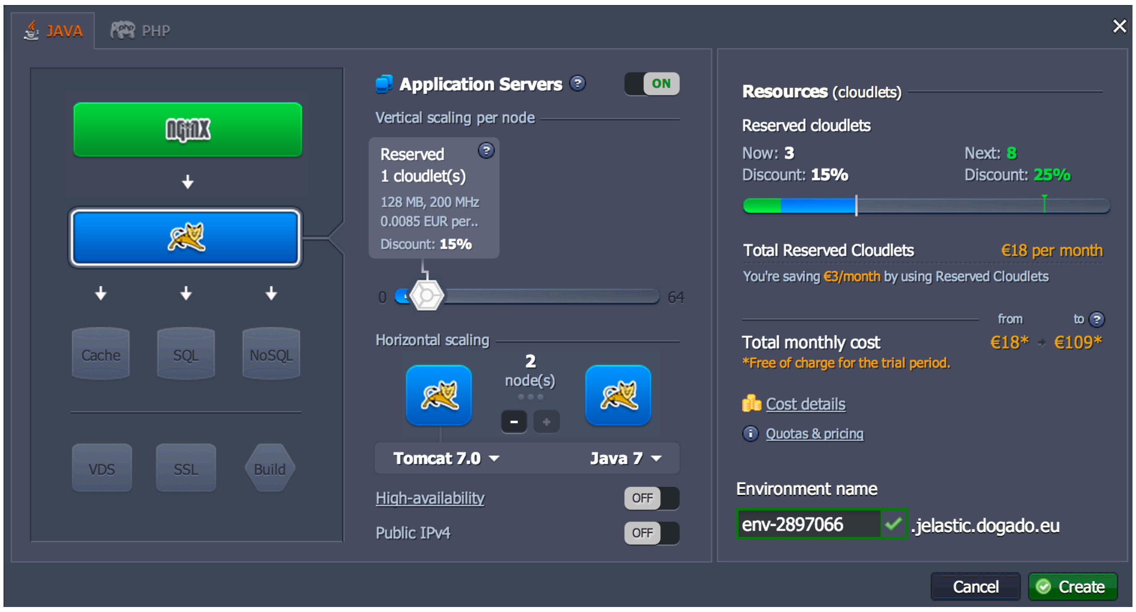
price
Total monthly cost €6*
*Free of charge for the trial period.

Cost details
Quotas & pricing

Environment name
env-2897066 .jelastic.dogado.eu

Figure 4.1: The Jelastic interface

Intuitively, such a constraint would be expressed as a classical Boolean constraint one may use when feature modeling, *e.g.*, $\mathcal{C}$: `ApplicationServer` $\rightarrow$ `Cloudlet`. However, this constraint would be semantically wrong. Let us explain this concern relying on the Jelastic configuration depicted by Figure 4.2. This configuration differs from the one depicted by Figure 4.1 by the number of application server nodes which are configured. Here, a second one has been added. There are now three cloudlets in the configuration, as depicted on the right-hand side of the figure. Two of them are dedicated to the application server nodes (blue color), while the third one is used to run the Nginx load balancer (green color). Thus, the `Cloudlet` feature is not part of the configuration because the `Application Server` is, which is the meaning of $\mathcal{C}$, but *each configured application server instance requires one cloudlet* to run properly. In the same way, the Nginx load balancer requires a cloudlet as well. This leads us to a second example. When configuring at least two application server instances, the Nginx load balancer is automatically configured to balance the load between nodes, if required. If only one application server instance is running, the load balancer cannot be configured. When modeling the variability of the Jelastic environment, a constraint is thus required to express that *if at least two instances of the* `Application Server` *feature are configured, then the* `Nginx` *feature must be selected.*



Application Servers ON
Vertical scaling per node
Reserved
1 cloudlet(s)
128 MB, 200 MHz
0.0085 EUR per..
Discount: 15%
0 64
Horizontal scaling
2
node(s)
Tomcat 7.0 Java 7
High-availability OFF
Public IPv4 OFF

Resources (cloudlets)
Reserved cloudlets
Now: 3 Next: 8
Discount: 15% Discount: 25%
Total Reserved Cloudlets €18 per month
You're saving €3/month by using Reserved Cloudlets
from to
Total monthly cost €18* → €109*
*Free of charge for the trial period.
Cost details
Quotas & pricing
Environment name
env-2897066 .jelastic.dogado.eu

Figure 4.2: Configuring two application server instances in Jelastic

These examples show that when relying on CardEx feature models to describe the variability of software systems, constraints over cardinalities and attributes must be defined to properly handle their configuration. It is thus necessary to provide a mean for expressing such constraints and the related support to reason on the CardEx feature models configurations.

### 4.2.2 Challenges

The examples previously discussed use constraints over feature instances to define a proper configuration. Our goal is to provide a support for expressing these constraints as well as constraints over features attributes, and reason about the related configurations. To achieve these objectives, we identify several challenges that have to be faced:

1. **Extend the concept of constraint.** *The selection of a feature implies or excludes the selection of another feature.* In certain circumstances, such Boolean constraints are not enough to express constraints in CardEx feature models. The first challenge is thus to extend the existing concept of constraint, and provides means to express new constraint constructs.

2. **Use a language independent syntax.** Our goal is not to provide a new feature modeling language with support for expressing such constraints, but rely on existing notations. The second challenge is thus to provide an abstract syntax to describe CardEx feature models that can be used by any variability modeling language or implemented tool support.

3. **Reason about CardEx feature model configurations.** Once the CardEx feature model is defined, one must be able to reason about its configurations, *e.g.*, the number of valid products. The third challenge is thus to be able to translate the proposed CardEx feature model abstract syntax into a format that can be managed by an automated reasoning engine.

## 4.3 Variability Modeling with CardEx Feature Models

This section describes in details the metamodel of our approach when variability modeling CardEx feature models. We then illustrate the use of this metamodel with several examples of the new constraint expressions taken from feature modeling variability of cloud environments and give the related semantics. Finally, we show how CardEx feature models can be translated to constraint programming to support automated reasoning operations.

### 4.3.1 CardEx Metamodel

Several works on feature modeling have proposed a metamodel to describe the use of feature models with an abstract syntax [CHE05a, BSTRC06, PBL05, Par11]. Therefore, our purpose with the CardEx metamodel is not to provide an entirely new metamodel, but an extension of the existing ones. We thus gather the literature existing concepts and reuse them to yield what we define as the $Init_{MM}$, as depicted by Figure 4.3. For example, we refer as `Feature` what is also called Feature in [CHE05a], but defined as Node by Parra [Par11], or `Alternative` what is described as AlternativeChoice by Pohl *et al.* [PBL05] but Grouped in [BSTRC06]. The extensions we provide are depicted in green in the metamodel and, together with the $Init_{MM}$, yield the CardEx metamodel.

The root of the CardEx metamodel is the `FeatureModel` meta-class, which contains *(i)* the `Root` feature and *(ii)* a set of `Constraints`. Each `Feature` may have `subFeatures` (if not, this is a leaf) or `Attributes`. Attributes are of type *int*, *boolean*, *real* or *enumeration*, which is a fixed set of values. A feature can be an `Alternative` feature, which in turn can be an `Exclusive` feature (for or-group and alternative-group respectively). An alternative has at least two `variants`, and is given a `GroupCardinality`, in opposition to features which are given a `FeatureCardinality`. In addition, we introduce in this $Init_{MM}$ the `DuplicatedExclusive` feature. A `DuplicatedExclusive` feature is a type of `Exclusive` where, when several instances can be configured, each feature instance and its whole subtree is identically configured. Note that duplicated subtrees can be obtained using an `Exclusive` feature and configuring all its instances in the same way, even if duplication is not explicitly specified. However, this duplication is forced when using a `DuplicatedExclusive` feature. Regarding constraints, `BooleanConstraints` are well-known and provide means to express that the selection of a feature `Implies` or `Excludes` another one. Relying on this $Init_{MM}$, one can model the Jelastic cloud as it was described in Section 4.2.1. The related feature model is depicted in Figure 4.4.

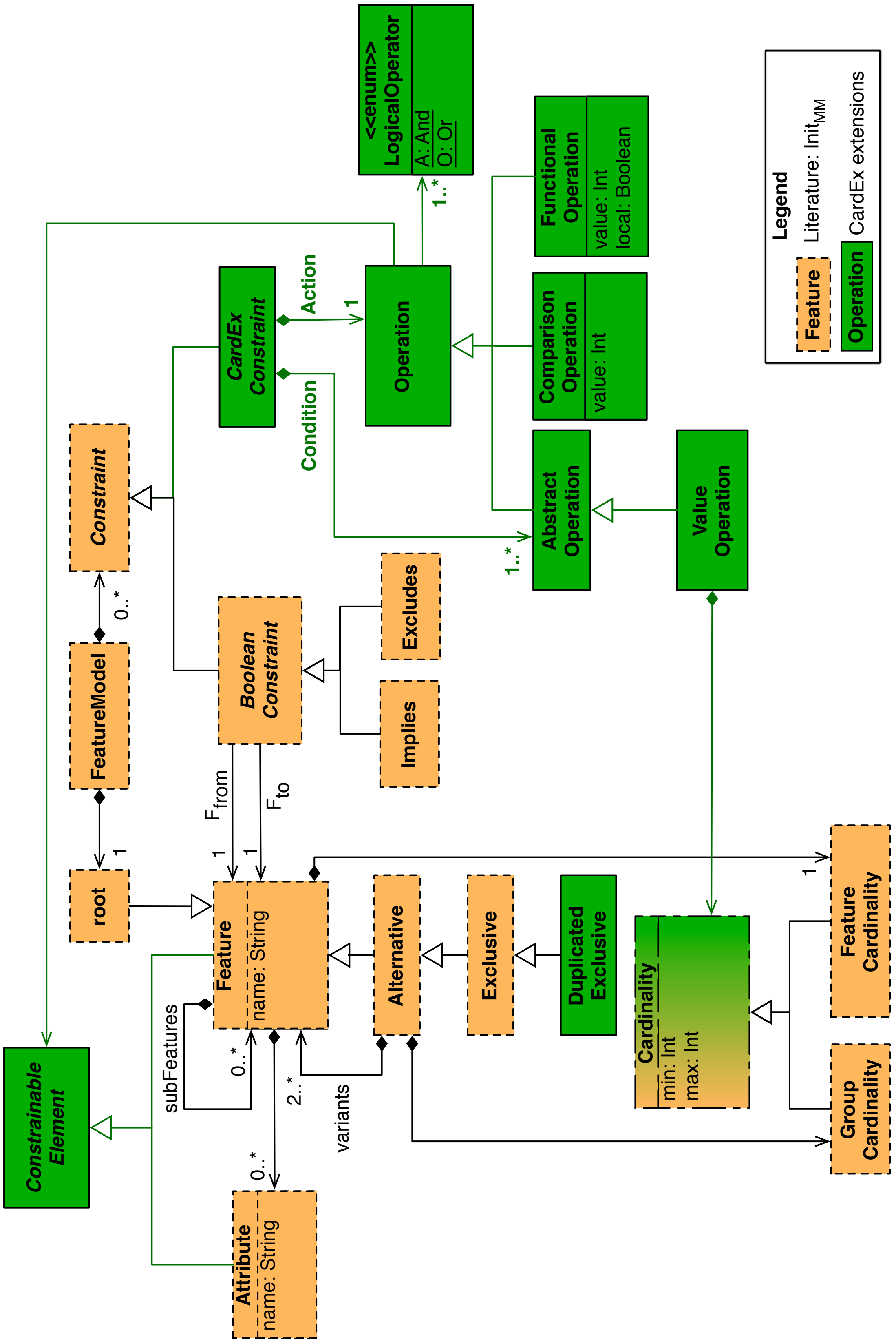


Figure 4.3: CardEx metamodel

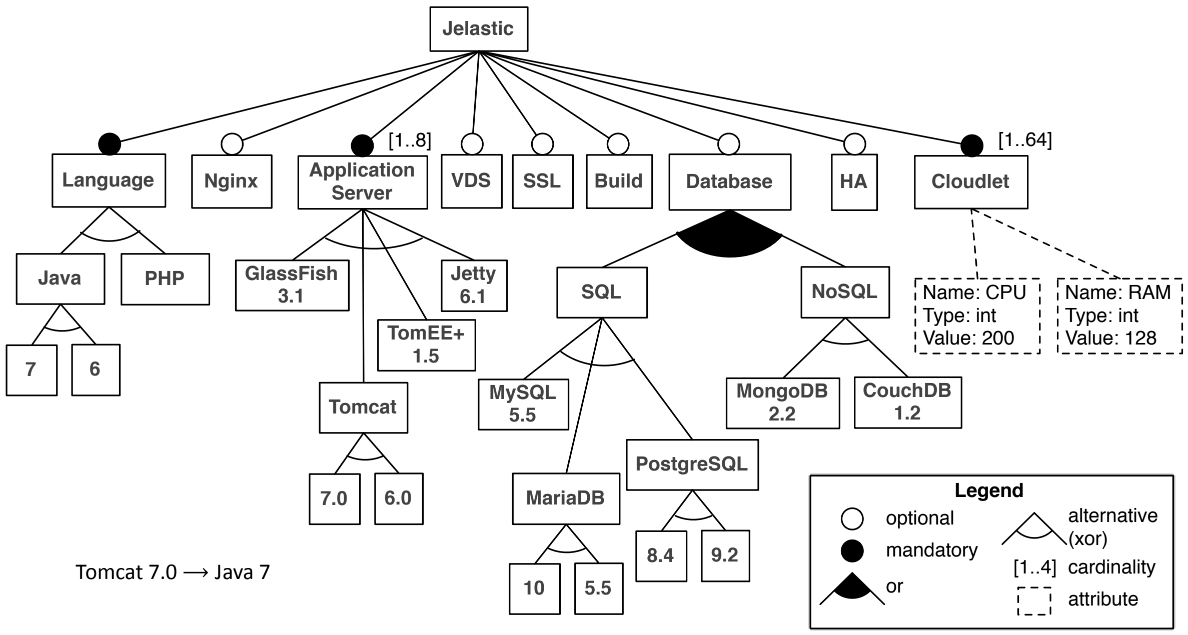


Figure 4.4: The Jelastic feature model

For instance, the *Nginx* feature is optional while the *Language* one is both a mandatory and an alternative feature, and one of the two proposed languages must be selected, *i.e.*, *Java* or *PHP*. The *ApplicationServer* illustrates a `DuplicatedExclusive` feature. Indeed, in Jelastic, one can configure up to 8 instances of the same application server. Then, if several instances are configured, the same application server must be selected for each instance, *e.g.*, *Jetty 6.1*. When configuring *Database*s, both a *SQL* one and a *NoSQL* one can be selected, illustrating an or-relationship. Finally, feature *Cloudlet* illustrates a cardinality and attributes, *e.g.*, the *RAM* one. We now discuss in details the constraints we introduced in the CardEx metamodel.

### 4.3.2 CardEx Constraints: Examples and Semantics

In addition to Boolean constraints, we introduce in our metamodel `CardEx Constraint`s. CardEx constraints are built on the following pattern: a list of at least one `condition` (connected by a `Logical Operator` if more than one) which, if satisfied, implies what is defined within the `action`. Both conditions and the action are `Operation`s, which are applied on `Constrainable Element`s, that is, a feature or an attribute. Thus, a CardEx constraint is written

$$condition_1\ \delta\ ...\ \delta\ condition_n \rightarrow action$$

with $\delta \in \{\wedge, \vee\}$. Note that the `Abstract Operation` is only for modeling purpose, as a condition is either a `Value Operation` or a simple `Operation`. For the latter, it is used to define a constrainable element with no particular related CardEx construct. Although our purpose is to provide a mean to express CardEx constraints (not to define a new syntax), we introduce in the following a concrete syntax for these constraints to illustrate their use with concrete examples. Considering that:

- $\mathcal{M} = (\mathcal{F}, \varphi)$ is a CardEx feature model, with $\mathcal{F}$ its non empty set of features and $\varphi$ its set of constraints;
- $\omega : \mathcal{F} \to \mathbb{N} \times \mathbb{N}$ indicates the cardinality of each feature, *i.e.*, $\forall f \in \mathcal{F},\ \omega(f) = [n, m]$;
- card: $\mathcal{F} \to \mathbb{N}$ indicates the number of instances for a feature, *i.e.*, $\forall f \in \mathcal{F},\ \texttt{card}(f) = n$ with $n \in \mathbb{N}$;
- $\texttt{attr} : \mathcal{F} \to \mathcal{A}$ 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 $\mathcal{M}$;
- $\texttt{val} : \mathcal{A} \to \mathcal{E}$ 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.

We define the available operations as follows.

**Value Operation.**

A value operation is used to reason on the value of a constrainable element, and can be used as the condition or the action of the CardEx constraint. We propose two ways of writing such an operation:

$$[i, j]\ constrainableElement \qquad (4.1)$$

$$constrainableElement\ \theta\ n \qquad (4.2)$$

with $i, j, n \in \mathbb{N}$, $i \leq j$ and $\theta \in \{=, <, \leq, >, \geq\}$.

To illustrate this notation, let us consider the two following examples. In the dotCloud PaaS cloud [dot], when configuring this environment to host an application, one is asked for a wished amount of RAM, which is provided from 32MB to 4GB. Regarding the specified RAM amount, dotCloud automatically allocates some disk space to store the application service. For instance, if there is at least 1GB and at most 2GB of RAM, then the disk size is set to 10GB. Such a constraint can be written

$$\mathcal{C}_1 :\ [1, 2]Memory.size \to Disk.size = 10$$

Thus, the condition relies on the notation (4.1) while the constraint action relies on the notation (4.2). For both operations, the constrainable element is a feature attribute. For the second example, please consider again the constraint we explained in Section 4.2.1, *if there* *are at least two instances of the Application Server feature, then the Nginx feature must be selected*. Then, relying on the notation (4.2) such a constraint can be written

$$\mathcal{C}_2 : \ ApplicationServer \geq 2 \rightarrow Nginx$$

In our approach, we rely on the `Cardinality` meta-class to define this operation. Although we could have chosen to use a dedicated meta-class, we think it would be redundant in the CardEx metamodel, as we just need an upper bound and a lower bound to define the range required for this operation. For example, an instance of this meta-class with $min = 1$ and $max = 2$ is used to express the condition of $\mathcal{C}_1$, while $min = 10$ and $max = 10$ are used for its action operation. Regarding $\mathcal{C}_2$, the use is slightly different, since there is no upper bound required. To handle this situation, we use the $-1$ value to define that the given bound is not a fixed value and will not be taken into consideration, that is, the condition of $\mathcal{C}_2$ is expressed with $min = 2$ and $max = -1$. In this case, the defined range is equivalent to a set expressed with the "*" multiplicity, *e.g.*, $[2, *]$ [RBSP02].

To summarize, there is one abstract syntax (the `Cardinality` meta-class) used to write constraints with different notations and semantics. Thus, considering a constraint $\mathcal{C}ons$ expressed

$$c_{from} \left\{ \begin{array}{l} f_{from} \\ a_{from} \end{array} \right. \rightarrow c_{to} \left\{ \begin{array}{l} f_{to} \\ a_{to} \end{array} \right.$$

with

- $c_{from}$, $c_{to}$ ranges defined as explained above,
- $f_{from}$, $f_{to} \in \mathcal{F}$,
- $a_{from}$, $a_{to} \in \mathcal{A}$,

then $\mathcal{C}ons$ is satisfied iff:

$$\left\{ \begin{array}{l} \texttt{card}(f_{from}) \\ \texttt{val}(a_{from}) \end{array} \right. \in c_{from} \Rightarrow \left\{ \begin{array}{l} \texttt{card}(f_{to}) \\ \texttt{val}(a_{to}) \end{array} \right. \in c_{to}$$

**Comparison Operation.**

Constraints based on a comparison operation are used to maintain a relationship between the values of two constrainable elements. This operation is only used as the action of the CardEx constraint, where the condition is usually a constrainable element. We propose to write such an operation:

$$n \ constrainableElement \tag{4.3}$$

with $n \in \mathbb{N}$.

To be satisfied, such constraints must hold whatever the value of the condition, where the constrainable element must be $n$ times greater than the condition value. For instance, this operation is extensively used in the PaaSage project [Paa13], where constraints require this operation, *e.g.*, the cloud environment must provide at least as much memory as the application to be deployed requires, or there must be twice more CPU power than memory space in the cloud configuration. Regarding the second example and considering the notation (4.3), the related constraint would be written

$$\mathcal{C}_3 : \; Cloud.RAM \rightarrow 2\, Cloud.CPU$$

In a more general way, considering a constraint $\mathcal{C}ons$ expressed

$$\left\{ \begin{array}{l} f_{from} \\ a_{from} \end{array} \right. \rightarrow n \left\{ \begin{array}{l} f_{to} \\ a_{to} \end{array} \right.$$

with

- $f_{from}, f_{to} \in \mathcal{F}$,
- $a_{from}, a_{to} \in \mathcal{A}$,

then $\mathcal{C}ons$ is satisfied iff:

$$\left\{ \begin{array}{l} \texttt{card}(f_{to}) \\ \texttt{val}(a_{to}) \end{array} \right. \geq n \times \left\{ \begin{array}{l} \texttt{card}(f_{from}) \\ \texttt{val}(a_{from}) \end{array} \right.$$

**Functional Operation.**

Constraints based on a functional operation are used to compute a given amount of required constrainable elements. As well as a comparison operation, this operation is only used as the action of the CardEx constraint, where the condition is a constrainable element. We propose to write such an operation:

$$+n \; constrainableElement \tag{4.4}$$

with $n \in \mathbb{N}$.

To illustrate this notation, let us consider an example taken from the Jelastic cloud environment. Jelastic provides the GlassFish open source application server to host Java EE applications, as depicted in Figure 4.4. Since GlassFish is an heavy application server, it requires 5 cloudlets to work properly [Gla]. Therefore, there must be (at least) 5 more cloudlets in the configuration than the number of running instances of GlassFish. Moreover, each database instance requires 1 more cloudlet in the configuration. Intuitively, and relying on the notation (4.3), such constraints would be written *GlassFish* $\rightarrow$ *5 Cloudlet* and *e.g.*, *MySQL 5.5* $\rightarrow$ *1 Cloudlet*. However, these constraints would be semantically wrong. Let us take an example based on the constraint *GlassFish* $\rightarrow$ *5 Cloudlet*. If 5 cloudlets are part of the configuration, then this constraint is satisfied. But what if 2 instances of GlassFish are running? Then 10 cloudlets are required, which is not expressed by the constraint.

There is thus a notion of *scope* to express the constraint. Does the constraint have to hold for each feature instance or for the feature itself only? Is the scope *local* or *global* respectively? The notion of scope we consider here differs from the one described by Michel *et al.* [MCHB11], that was dealing with the semantics of feature cardinalities according to the way feature models are modeled. The scope was either *Clone* or *Feature*, and used in the presence of feature cardinality to define if the instances of features should be counted rather than the features that were instantiated. To define the scope in our CardEx constraints, we rely on the *local* attribute of the `Operation` meta-class. In the previous example involving GlassFish, this attribute would be set to *true*, meaning that for each running GlassFish instance, there must be 5 more cloudlets. To take into consideration the scope in our CardEx construct, we propose to use the apostrophe punctuation mark $'$ in the notation when the scope is local, that is, when the constraint must hold for each feature instance. The constraint discussed above would then be written

$$\mathcal{C}_4: \ GlassFish' \rightarrow +5\ Cloudlet$$

At this point, the reader may wonder what is the difference between a constraint based on a comparison operation and a functional operation. We consider functional constraints as *local* constraints, that is, constraints which are local to the targeted constrainable element, *e.g.*, Cloudlet. Local constraints must be computed before being considered as *global* constraints, which are used to reason on the feature model configurations. Let us illustrate the difference using a generic feature model as example, *i.e.*, the `FM` feature model depicted by Figure 4.5.

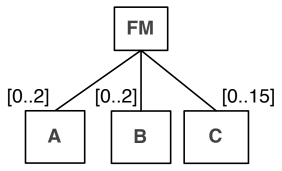


Figure 4.5: A feature model

Now, consider the following configuration of the `FM` feature model $conf$: *2 instances of A and 2 instances of B*, with two constraints $Ex_1$ and $Ex_2$ expressed either as comparison or functional constraints. The resulting amount of instances of C regarding the used constraints is given by Table 4.1.

Regarding comparison constraints, 6 instances of C are required for the configuration to be valid, since it is enough for $Ex_1$ and $Ex_2$ to be satisfied. When using functional con-

| | $Ex_1 : A \rightarrow 2C$ <br> $Ex_2 : B \rightarrow 3C$ | $Ex_1 : A' \rightarrow +2C$ <br> $Ex_2 : B' \rightarrow +3C$ |
|---|---|---|
| Amount of C *w.r.t.* A | 4 | 4 |
| Amount of C *w.r.t.* B | 6 | 6 |
| Required amount of C | $\geq 6$ | $\geq 10$ |

Table 4.1: Difference between comparison and functional constraints

straints, 10 instances are required, which is the sum of all constraints. That is why such constraints must be computed independently before being translated as global constraints. For instance, constraints $Ex_1$ and $Ex_2$ computed as a global constraint when considering $conf$ and functional constraints give $Ex_{1+2}$: $2A \wedge 2B \rightarrow C \geq 10$. These constraints are based on the assumption that the targeted constrainable element is not directly configurable. In such a situation, the configuration space can be seen as two configuration perspectives, the user perspective and the system perspective. The user can configure elements she/he has access to, while elements from the system perspective and their values are deducted from the user configuration, *e.g.*, the configuration total price or the related resource amount.

## 4.4 Reasoning on CardEx Feature Models

Manually analyzing feature models is a tedious and error-prone task, quite infeasible to do with large-scale feature models. Thus, automating the analysis of feature models has become an important and active area of research for both practitioners and researchers in the software product line community [Ach11]. The automated analysis of feature models is about extracting information from feature models using automated mechanisms and has been subject to an exhaustive literature review [BSRC10]. Among proposed analysis operations, some of them are extensively used *e.g.*, "*is a feature model void*" (*i.e.*, it represents no product), "*is a product valid*", "*list all products*" or "*give the number of products*".

To support such operations, several kinds of automated support have been proposed and can be classified regarding if they are based on:

- *Propositional Logic.* Feature models are mapped to a propositional formula, which is a set of primitive symbols or variables and a set of logical connectives constraining the values of the variables, *e.g.*, $\wedge, \vee$ or $\Rightarrow$. The formula is then given as input to an off-the-shelf SAT or *Binary Decision Diagram* (BDD) solver. The first connection between feature models and propositional formula was established in [Man02]. Then, Batory *et al.* were the first to use a SAT solver to reason on feature models [Bat05], while

Benavides *et al.* proposed a multi-solver approach selecting either a BDD or SAT solver regarding the kind of operation to be performed [BSTC07].

- *Constraint Programming.* A *Constraint Satisfaction Problem* (CSP) consists of a set of variables, a set of finite domains for those variables and a set of constraints restricting the values of the variables. Constraint programming can be defined as the set of techniques such as algorithms or heuristics that deal with CSPs. A CSP is solved by finding states (values for variables) in which all constraints are satisfied. In contrast to propositional formulas, CSP solvers can deal not only with binary values (true or false) but also with numerical values such as integers or intervals [BSRC10].

- *Description Logic.* As explained in [BCM+03], description logics are a family of knowledge representation languages enabling the reasoning within knowledge domains by using specific logic reasoners. A problem described in terms of description logic is usually composed by a set of concepts (*i.e.*, classes), a set of roles (*e.g.*, properties or relationships) and a set of individuals (*i.e.*, instances) [BSRC10]. A description logic reasoner takes as input a problem described in description logic and provides facilities for consistency and correctness checking and other reasoning operations, *e.g.*, based on OWL-DL [WLS+05].

- Other approaches which are not part of one of the previous groups, proposing ad-hoc solutions, algorithms or paradigms [BSRC10].

In our approach, we rely on CSP to reason on CardEx feature models, as we need numeric values, for both attributes and cardinalities. The translation from feature models to CSP is well-known [BTC05, BSTRC06, MSDLM11], but must be extended to support the constraints previously defined. Table 4.2 lists the rules for mapping a CardEx feature model into a CSP. The table describes the different relations between features, their feature model notation and the related CSP constraint. Thus, features depicted in the central column are defined as CSP variables in the right column. Note that since constraints based on functional operations are then translated to a global constraint, there is not an unique way of writing such constraints, as it depends on their original formula. The constraint described in the last row of the table thus relies on the example of constraint $Ex_{1+2}$: $2A \wedge 2B \rightarrow C \geq 10$.

Regarding features, each one is translated into a variable of the CSP while cardinalities define the domain of these variables. Mandatory and optional features are given the $\{0,1\}$ domain. Why is the domain of mandatory not set to $\{1,1\}$? Because in that case, the related feature would be part of every configuration, which may be wrong, *e.g.*, if its parent feature is an optional feature not selected. The constraint defined in the first row of Table 4.2 thus forces a mandatory feature to be selected whenever its parent feature is selected. In the same way, features whose cardinality is $\{m,n\}$ are given the $\{0,n\}$ domain, while the constraint described in the third row of Table 4.2 forces the value of the variable to be at least equal to $m$ whenever its selected.

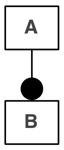
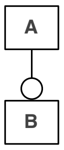
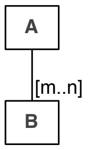
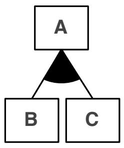
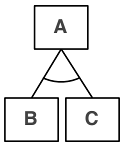
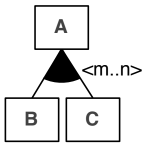
| **Relation** | **Feature model notation** | **Constraint** |
|---|---|---|
| Mandatory |  | $B = A$ |
| Optional |  | ifThen$(A = 0; B = 0;)$ |
| Feature Cardinality |  | ifThenElse$(A = 0; B = 0; B \in \{m, n\})$ |
| Or-group |  | ifThenElse$(A > 0; sum(B, C) \geq 1; sum(B, C) = 0;)$ |
| Alternative-group |  | ifThenElse$(A > 0; sum(B, C) = 1; sum(B, C) = 0;)$ |
| Group cardinality |  | ifThenElse$(A > 0; sum(B, C) \in \{m, n\}; sum(B, C) = 0;)$ |
| Implies | $A \to B$ | ifThen$(A > 0; B > 0;)$ |
| Excludes | $A \to \neg B$ | ifThen$(A > 0; B = 0;)$ |
| Value | $[i, j]A \to [m, n]B$ | ifThen$(A \in \{i, j\}; B \in \{m, n\};)$ |
| Comparison | $A \to nB$ | ifThen$(A > 0; B \geq n \times A;)$ |
| Functional | $mA \wedge nB \to C \geq k$ | ifThen$(and(A = m, B = n); C \geq k;)$ |

Table 4.2: Feature model notations and related constraints

## 4.5 Challenges Revisited and Discussion

We have presented our approach for feature models extended with attributes and cardinalities and described how they can be mapped to CSP for automated reasoning on their configurations. Let us now revisit the challenges identified in Section 4.2 and discuss how our approach face them.

1. **Extend the concept of constraint.** To face this challenge, we extend the existing concept of Boolean constraints, *i.e.*, *implies* and *excludes*, where a condition *e.g.*, the selection of a feature, implies an action, *e.g.*, the selection of another feature. The extensions we provide for expressing constraints allow us to define several constructs regarding features and/or attributes values, *e.g.*, $n$ instances of a feature implies an attribute value to be greater than $m$.

2. **Use a language independent syntax.** To face this challenge, we provide an abstract syntax to define CardEx feature models. We rely on existing concepts one can find in the literature to describe feature models, gathered in the $Init_{MM}$ metamodel. Thus, our approach is not a new formalism in the feature modeling community, but an extension of existing approaches.

3. **Reason about CardEx feature model configurations.** To face this challenge, we describe how operations on CardEx feature models can be performed in an automated way. Our approach is based on constraint programming, where CardEx feature models are translated into CSP. Once again, we extend the existing translation proposals to handle constraints based on attributes and cardinalities.

**On the expressiveness of CardEx feature models.**
Even though the need for constraints with cardinalities and attributes has already been discussed [MCHB11], there is no consensus on the constructs required to express such constraints. Thus, one question that may naturally arise is: is our approach expressive enough regarding CardEx constraints? In this chapter, we proposed means to express constraints for both feature cardinality and attributes, where their value can be constrained to be in a given range, greater, lower and/or equal to a given value, or compared with another value. Although we do not pretend our approach to be exhaustive, we can fairly argue that this is enough to support CardEx feature models definition in most cases. For instance, cloud computing environments variability can be entirely handled using our approach.

**On extending the existing $Init_{MM}$ metamodel.**
Another question that may arise while reading this chapter is: since the proposed extensions are dedicated to constraint expressions, why not using the *Object Constraint Language* (OCL) instead? There are several reasons for that. First, OCL is not dedicated to define constraints

in feature models as it is a general-purpose constraint language. Thus, writing constraints is not intuitive when feature modeling, and complex expressions are hard to write and maintain [KC12]. Second, the use of OCL is not popular in the feature modeling community. A significant amount of approaches dealing with feature models rely on an automated tool support based on approaches classified in Section 4.4. Using OCL would then be risky as our approach may not be used by the community. Moreover, using OCL introduces shortcomings mainly related to its scalability [KC12], while off-the-shelf solvers are well-known to reason on feature models with thousands on features in a few seconds [TBK09]. Finally, in a more general way, the question could be: why extending the existing constraint support instead of reducing OCL? Reducing OCL to fit our needs would be a difficult task, since equivalent to build a *Domain Specific Language* (DSL) for constraints in CardEx feature models, that is, much more effort than extending existing constraint support. Moreover, that would be implementing another OCL-based tool, among the plethora of existing ones [CODA+11].

## 4.6 Summary

This chapter presented our abstract syntax to define feature models extended with attributes and cardinalities, in particular when they are involved in constraints. We have illustrated our approach with examples from cloud environments variability and described the related semantics of those constraints. New constraint constructs extend the existing formalism, where a given condition, if satisfied, implies a specific action. Relying on these extensions, one can define constraints on feature and attribute values. We also described the related translation into CSP, thus providing means to use it to reason in an automated way on CardEx feature models.

In the next chapter of this dissertation, we present our approach for detecting inconsistencies that may arise when evolving cardinality-based feature models, as defined cardinalities may not represent the concrete set of products.

# Chapter 5

# Evolution and Consistency Checking of Cardinality-based Feature Models

**Contents**

## 5.1 Introduction

According to Kent Beck, change is unpredictable, and software evolution seems to be inevitable [BB10]. Many factors can explain why software evolve [Leh80]. For instance, software are maintained over time, *e.g.*, financial systems used by bank companies. Maintaining software systems means making them evolve, *e.g.*, to meet new user or domain requirements or to adapt to a new hardware or related software, *e.g.*, to a library the software is dependent of. As any other software systems, software product lines are subject to changes and evolution along their lifecycle [GWTB12]. Moreover, the cost of variability management in software product lines is meant to be offset by the savings in deployment of product variants over time [LSB+10], and since software product lines are a long-term investment, they need to evolve to meet new requirements over many years [BP14].

As a consequence, variability models of software product lines, and feature models in particular, are subject to evolution as well. While evolving feature models can be seen as a common task, *e.g.*, edits like adding or removing a feature, more complex evolutions may be required, *e.g.*, merging the existing feature model with another one or splitting it into several part [Ach11]. Moreover, incremental changes can be applied to feature models, making it difficult to check the correctness of the whole evolution process. Thus, the consistency of feature models may be compromised during the evolution.

As described in Chapter 3, several research works provide an automated support or describe an approach to reason about edits to feature models and their impact [TBK09, GWTB12, PDv12], or propose a catalog of evolution patterns for variability models [LSB+10, PCW12] and their related artifacts [GWTB12, SHA12, PGT+13]. However, these approaches are dedicated to Boolean feature models. Thus, support to reason on edits to cardinality-based feature models is still missing. In practice, feature model configuration can be a complex task due to their size and complexity [RGD10]. It is hence strongly desirable to avoid additional complexity or extra effort (*e.g.*, manually analyzing why a certain cardinality value cannot be selected) caused by an inconsistent feature model. This chapter addresses this need. We discuss feature models edits during evolution with respect to feature cardinalities and present a formal approach to detect and explain inconsistencies in cardinality-based feature models, that is, with cardinality for both features and constraints, as described in Chapter 4.

The chapter is structured as follows: Section 5.2 describes a motivating example highlighting the need for consistency checking cardinality-based feature models when evolving them and discusses the related challenges. Section 5.3 presents atomic edits that can be performed when evolving cardinality-based feature models, and how they may lead to inconsistencies. Section 5.4 explains the different kinds of cardinality consistencies, gives their definition and explain how they are related each other. In Section 5.5, an automated support for checking the consistency of cardinality-based feature models and giving explanations - when required - to its designer is presented. Section 5.6 describes an improved version of this support to provide more user-friendly feedback for the detections and explanations. Finally, Section 5.7 discusses several concerns regarding our approach while Section 5.8 concludes the chapter.

## 5.2 Motivation and Challenges

This section illustrates the problems that might arise when evolving cardinality-based feature models with a motivating example involving on the Jelastic feature model. Based on this example, we then discuss challenges related to evolving cardinality-based feature models and reasoning on their consistency.

### 5.2.1 Motivating Example

In August 2011, Jelastic announced the full support for the GlassFish application server as new functionality for their cloud platform. Initially, Jelastic already provided support for Jetty, the Java open source application server. Figure 5.1 depicts this situation on an extract of the Jelastic feature model.

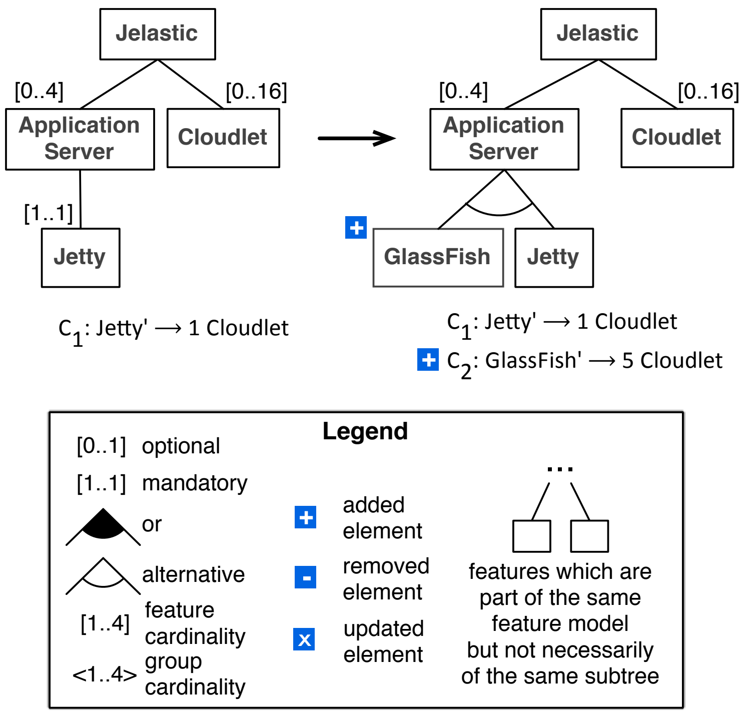


Figure 5.1: Evolution Example

The left-hand side shows an initial version that supports only Jetty as application server. The right-hand side shows the feature model after the evolution where GlassFish has been added as an alternative application server. While Jetty is a lightweight application server, GlassFish is an heavier one and, as described on its related Jelastic documentation web page, *"more functions = more resources"*. Thus, the `GlassFish` feature comes together with the constraint $C_2$, as GlassFish requires 5 Cloudlets to work properly [Gla]. Elements added during the evolution are depicted with the "+" symbol, removed ones with a "−" and updated ones with a "×". As features may not be part of the same subtree, we indicate this situation using the three dots symbol as the common parent. In the remainder of this chapter, we will rely on this notation to depict cardinality-based feature model evolution scenarios. In particular, as the focus of the chapter is on cardinalities, the FODA notation for mandatory and optional features is replaced by the [1..1] and [0..1] cardinality respectively.

However, although the depicted evolution scenario remains quite basic, this change makes the feature model range inconsistent: according to the `ApplicationServer` feature cardinality, there can be up to 4 application server instances in a given configuration. Nevertheless, configuring 4 GlassFish instances means that 20 Cloudlet instances are required, which is not allowed by the `Cloudlet` feature cardinality. This example demonstrates that manually evolving a small cardinality-based feature model is error-prone. Thus, handling feature models with hundreds or thousands of features requires specific tooling.

A feature cardinality is considered as range-inconsistent if no product exists for some values of its range. This definition is analogous to the concept of *wrong cardinality* defined for group cardinalities in the existing literature [TR09, BSRC10]. Thus, the feature model may be well-formed (*syntactic consistency*) and defining at least one valid product (*semantic consistency*), but inconsistent regarding defined cardinalities. In the remainder of this chapter, we discuss cardinality-based feature models edits and the resulting range inconsistencies of feature cardinalities and introduce a formal approach to support detection and explanation of range inconsistencies in such feature models.

### 5.2.2 Challenges

The example previously discussed highlights the need for detecting inconsistencies when evolving cardinality-based feature models. Our goal is thus to provide such a capability, and help the user finding the reason why those inconsistencies arise. To achieve these objectives, we identify the following challenges that have to be faced:

1. **Detect cardinality inconsistencies.** The cardinality-based feature model may be well-formed, but the expected number of feature instances one may be able to configure regarding the defined cardinalities, and therefore the number of products, may be different from the real one. The first challenge is therefore to detect such inconsistencies, thus improving the correctness of the feature model.

2. **Explain the inconsistencies.** Detecting inconsistencies is good, understanding why such inconsistencies arise is better, as they may be caused by different factors. The second challenge is thus to provide some feedback and explanations to the feature model designer for her to modify the feature model design and fix the related inconsistencies.

2. **Provide an automated support.** Detecting inconsistencies and getting explanations about them can be a tedious process when handling large feature models. The third challenge is thus to provide an automated support to reason on cardinality-based feature models consistency, as well as an effective explanation engine.

## 5.3 Edits to Cardinality-Based Feature Models

This section investigates edits to cardinality-based feature models with respect to consistency of the defined cardinalities. By edits, we consider atomic model edits, *i.e.*, to *add*, *remove* or *update* a model element, where updating a model element means changing one of its properties. Figure 5.2 depicts the cardinality-based feature model, which is actually the same than the CardEx metamodel depicted by Figure 4.3 in Chapter 4, without feature attributes. Note that for the sake of simplicity, constructs described about CardEx constraints described in Chapter 4 have been removed from the figure, but are still considered.

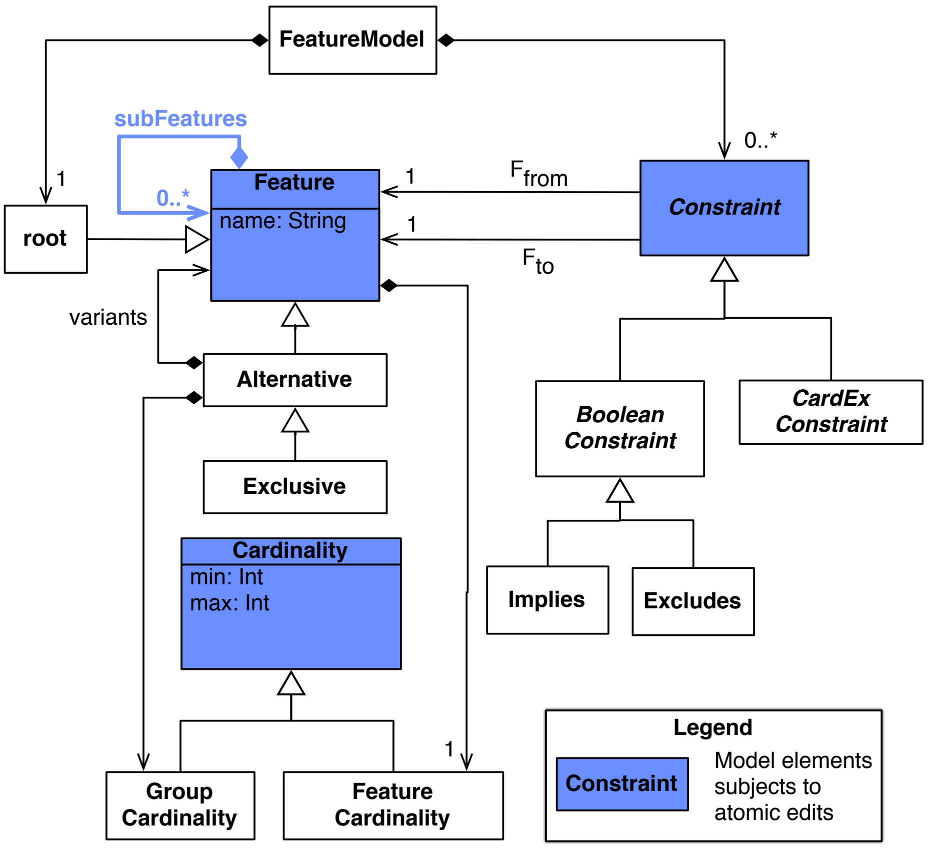


Figure 5.2: Cardinality-based feature model

Thus, relevant model elements concerned by atomic edits, highlighted in blue in Figure 5.2, are features, cardinality and cross-tree constraints, while relevant properties are feature name, cardinality *min* and *max* bounds, and location in the feature model (*i.e.*, the reference to the parent feature). By means of inheritance relationships, all types of features, cardinality and constraints are thus concerned by those edits as well. This results in nine atomic cardinality-based feature model edits discussed in this chapter and listed in Table 5.1. An exclamation mark (!) indicates if the considered edit can lead to inconsistent feature cardinalities while a dash (–) is used otherwise. In practice, knowing which edits do not lead to inconsistent feature cardinalities is useful because it provides the possibility to save effort by not checking the feature model consistency after one of those edits has been performed. The table also describes if an edit is not applicable (n.a.) for the given model element.

| | Add | Remove | Update | Move |
|---|---|---|---|---|
| Feature | – | – | – | ! |
| Feature Cardinality | n.a. | n.a. | ! | n.a. |
| Group Cardinality | n.a. | n.a. | ! | n.a. |
| Constraint | ! | – | ! | n.a. |

Table 5.1: Atomic edits

There are thus five of the nine atomic edits that can lead to inconsistent feature cardinalities. However, more complex edits can be composed from atomic edits (e.g., *inserting* a feature into the feature tree hierarchy consists of adding a feature and moving a sub-tree to become its child) and can result in inconsistencies if any of the involved atomic operators lead to inconsistencies. Note that range inconsistencies are always detected on feature cardinalities. If a change on a group cardinality leads to a range inconsistency for a feature cardinality, we consider that we have to fix the feature cardinality, not the group one. Most of the scenarios described here arose during our work regarding cloud environment variability modeling. However, we believe these scenarios may occur in any other domain involving cardinality-based feature modeling, and we therefore depict them in a general way, relying on the notation described in Section 5.2.

**Add Feature.** The atomic edit *add* means adding a new feature as a leaf node to the model, since inserting a feature as non-leaf node requires to move an existing sub-tree to become its child. Adding a feature does not lead to inconsistent feature cardinalities as it does neither influence any existing cardinalities nor any existing constraints.

**Remove Feature.** The atomic edit *remove* operation means to delete a feature and its children from the model. While this can lead to inconsistencies in general, such as dangling references in cross-tree constraints (in such a case, the involved constraint must be removed before removing the feature), it does not lead to inconsistent feature cardinalities for the same reasons as the *add* edit.

**Update Feature Name.** Updating the name of a feature does not lead to inconsistent feature cardinalities.

**Move Feature.** Moving features can occur during refactoring a model, when inserting new features (as non-leaf nodes), or removing non-leaf features. It can lead to inconsistent feature cardinalities if there are hierarchies of cardinalities.

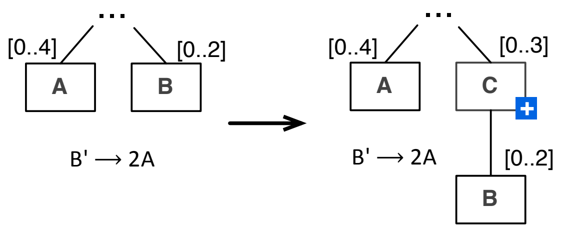


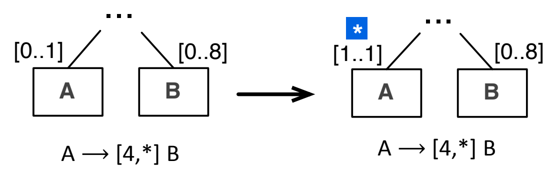
Figure 5.3: Moving a feature

Figure 5.3 depicts an example where feature B is moved in the feature model to become a child of feature C (e.g., as part of inserting a new feature C). An inconsistency can arise from the combined cardinalities of C and B. We rely on the local notion of feature instances already chosen by other authors [CHE05a, MCHB11], that is, each instance of C provides up to two instances of B. As there can be up to three instances of C, there can be altogether up to six instances of B. The cross-tree constraint specifies that each instance of B requires two instances of A, *i.e.*, up to 12 instances of A. But the feature cardinality of A allows only up to four instances which constitutes an inconsistency.

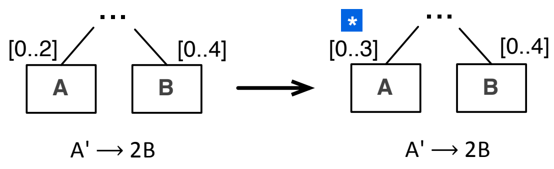
**Update Feature Cardinality.** Each feature is given a cardinality, even mandatory and optional features, which are represented as cardinality [1..1] and [0..1] respectively. Thus, a cardinality cannot be removed, added nor moved as summarized in Table 5.1, but only updated. Such an update is performed on the lower and/or upper bound of the cardinality and can result in inconsistent feature cardinalities. Updating a feature cardinality occurs, for instance, when a cloud provider changes its service offer, *e.g.*, one can now run more than two database instances.

Figure 5.4 shows two examples for this scenario where the cardinality of feature A is updated. In Figure 5.4a, A is an optional feature and is evolved to become a mandatory feature. The cross-tree constraint specifies that if A is selected there must be at least 4 instances of B. When A becomes mandatory there can never be less than four instances of B which is inconsistent with the feature cardinality of B. In Figure 5.4b, the upper bound changes from 2 to 3, but since the lower bound of the new cardinality is still 0, feature A remains an optional feature after this edit. The cross-tree constraint specifies that each instance of A requires 2 instances of B. As the upper bound of B is still 4, it is not possible to configure 3 instances of A which constitutes an inconsistency.

**Update Group Cardinality.** A group cardinality may change when the system specifications evolve. For instance, the cloud environment might now support additional frame-



(a) Optional to mandatory



(b) Arbitrary cardinality

Figure 5.4: Updating feature cardinality

works to be used in the same configuration. Updating a group cardinality can lead to inconsistent feature cardinalities.

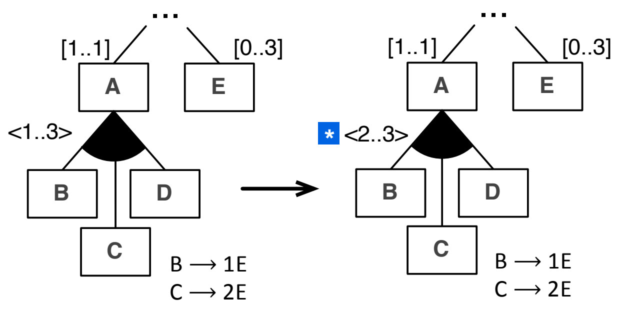


Figure 5.5: Updating a group cardinality

In the scenario depicted by Figure 5.5, one must now select at least two child features when A is configured, instead of one before the evolution. The constraints define the number of E instances to be configured when selecting B or C. After the evolution, either B or C must be selected (or both of them in the same configuration), which then requires at least one instance of E. However, this is inconsistent with the feature cardinality of E, which is an optional feature that thus does not have to be part of all configurations.

**Add Cross-Tree Constraint.** Adding a cross-tree constraint occurs, *e.g.*, when a constraint is not part of the initial specification, but is added later based on experience with the system. It may also be added together with a new feature, *e.g.*, as depicted in Figure 5.1. Both, cardinality-based constraints and purely Boolean constraints can lead to inconsistent feature cardinalities.

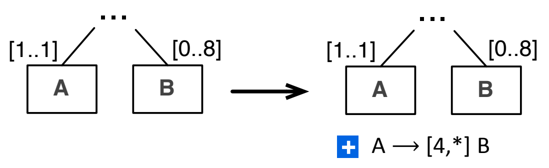


Figure 5.6: Adding a constraint

In the example depicted by Figure 5.6, a new constraint is added to specify that feature A requires at least four instances of B. As A is a mandatory feature there can never be less than four instances of B which is inconsistent with the feature cardinality of B.

**Remove Cross-Tree Constraint.** Removing a cross-tree constraint occurs when the constraint is not necessary anymore and cannot result in inconsistent feature cardinalities. Given that the model is consistent before the evolution (*i.e.*, a feature is part of at least one product for each of its cardinality value), an inconsistency can only arise if *(i)* at least one of the products is removed from the set of valid products or *(ii)* a new cardinality value is added to the set of cardinality values. However, removing a constraint can never restrict the set of products nor extend the cardinality values set.

**Update Cross-Tree Constraint.** Updating a cross-tree constraint occurs, *e.g.*, when an existing cloud service now provides less CPU power, thus requiring more instances to fit the same requirements than before. It can lead to inconsistent feature cardinalities in the same way as adding cross-tree constraints. Figure 5.7 shows an example where the cardinality

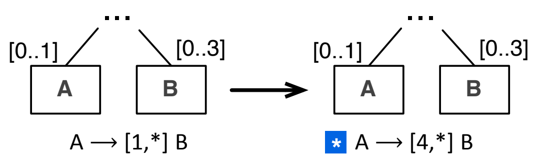


Figure 5.7: Updating a Constraint

range evolves. In this scenario, this edit leads to an inconsistency if A is selected, since feature A requires at least four instances of B, which is not possible.

This section has shown that among all atomic edits, five out of nine can result in inconsistent feature cardinalities (see Table 5.1). Since more complex edits can be composed from these atomic edits, there is a high probability to get an inconsistent cardinality-based feature model when evolving it manually. In the following section, we present a formal approach to check the consistency of cardinality-based feature models.

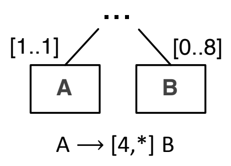
## 5.4 Cardinality-based Feature Model Consistency

In the previous section, we described how edits on cardinality-based feature models can lead to inconsistent cardinalities. Some of those inconsistencies may seem obvious, *e.g.*, the one in the *add cross-tree constraint* scenario depicted by Figure 5.6. However, the reader should keep in mind that features involved in these scenarios may be located in different subtrees of the feature model, with tens of features and constraints in the feature model, or even more, making those inconsistencies difficult to detect. In this section, we thus present a formal approach to detect cardinality inconsistencies, based on a definition of consistency for cardinality-based feature models.

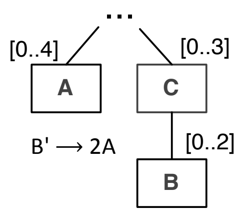
### 5.4.1 Different Forms of Consistency

To define consistency for cardinality-based feature models, we specify two closely related types of inconsistencies over feature cardinalities:

- **Local range inconsistency.** There is a local range inconsistency in the feature model when there exists no product for one or several values defined by a cardinality range.
- **Global range inconsistency.** There is a global range inconsistency in the feature model when a range inconsistency arises after taking into account the hierarchies of ranges.



(a) Local range inconsistency



(b) Global range inconsistency

Figure 5.8: Cardinality Inconsistencies

Examples for local and global inconsistencies can be found in Figure 5.4a and Figure 5.3 after evolution has occurred, depicted here in Figure 5.8 again for convenience. Figure 5.8a shows a local range inconsistency: as feature A is mandatory and due to the constraint, there must always be at least 4 instances of B while the feature cardinality of B specifies a lower bound of 0. This inconsistency occurs without having to take the cardinalities of parent feature into account. Figure 5.8b shows a global range inconsistency: it only occurs due to the hierarchy of cardinalities. As there is up to 3 instances of C with up to 2 instances of B for each instance of C, there can be altogether up to 6 instances of B. Due to the cross-tree constraint this can require, in turn, up to 12 instances of A which conflicts with the feature cardinality of A. This inconsistency occurs only when taking the hierarchies of feature cardinalities (here C and B) into account.

Thus, considering a cardinality-based feature model $\mathcal{M} = (\mathcal{F}, \varphi, \omega)$, with:

- $\mathcal{F}$ the non-empty set of features of $\mathcal{M}$;
- $\varphi$ the set of constraints of $\mathcal{M}$;
- $\omega : \mathcal{F} \rightarrow \mathbb{N} \times \mathbb{N}$ indicates the cardinality of each feature. It will be denoted as a range $\omega(f) = [m, n]$;

In our context, a product may contain several instances of the same feature. It can either be represented as a multiset of feature names or as a set of pairs *(feature name, number of instances)*. We only require to retrieve the number of instances of a given feature for a given product. We denote $\mathcal{P}$ the set of all $\mathcal{M}$ products and $|f|_p$ the number of instances of the feature $f$ in product $p$.

**Definition 1.** LOCAL FEATURE MODEL RANGE CONSISTENCY
A cardinality-based feature model $\mathcal{M}$ is locally range consistent if and only if, for each feature, there exists a product for each value defined in the feature cardinality containing exactly that number of instances of the feature. More formally, it can be defined as follows:

$$\forall f \in \mathcal{F}, \forall i \in \omega(f), \exists \{p \in \mathcal{P} \mid f \in p \wedge |f|_p = i\}$$

Thus, the feature model depicted in Figure 5.8a is not locally range consistent, since there are values in $\omega$(B) that are never used, *i.e.*, {0, 1, 2, 3}. Regarding the feature model depicted in Figure 5.8b, as explained above, one cannot configure more than two instances of B. However, this feature model is still local consistent, that is, there is a valid product for each cardinality value of B and C. Indeed, if there is no B configured, then C can take any cardinality value and if the cardinality of C is set to 1, then B can take any cardinality value as well. Relying on the local consistency property of a feature model to detect all cardinality inconsistencies is thus not enough, as it does not take into account the number of feature instances that can be configured regarding the parent feature cardinality. We thus extend the Definition 1 to deal with this issue.

**Definition 2.** GLOBAL FEATURE MODEL RANGE CONSISTENCY
A cardinality-based feature model $\mathcal{M}$ is globally range consistent if and only if, for each feature, there exists a product for each value defined in the feature cardinality containing exactly that number of instances of the feature associated to each instance of the parent feature. More formally, it can be defined as follows:

$$\begin{gathered}\forall f \in \mathcal{F} \setminus \{root\}, g = \texttt{parent}(f), \forall j \in \Omega(g),\\ \forall (x_1, ..., x_j) \in \omega(f) \times ... \times \omega(f),\\ \exists \{p \in \mathcal{P} \mid |g|_p = j \wedge \bigwedge_{i=1}^{j} |f_i|_p = x_i\}\end{gathered}$$

where `parent`(*f*) is a function that indicates the parent feature of the $f$ feature, *root* is the root feature of $\mathcal{M}$, $\Omega(g)$ denotes the range of occurrences of $g$ in the product, and $f_i$ denotes the instances of $f$ associated to the $i^{th}$ instance of $g$. Note that the $root$ feature is considered as globally range consistent since it only has one occurrence and no parent. Note also that to break symmetries, it is sufficient to consider the cases where $x_1 \geq x_2 \geq \ldots \geq x_j$.

We thus define a cardinality-based feature model as a *range complete* feature model if no local or global range inconsistency is detected.

**Definition 3.** RANGE COMPLETENESS
A cardinality-based feature model is range complete if it is both globally range consistent and locally range consistent.

Yet, it is interesting to note that the notion of range completeness in feature model (Definition 3) is very close to the notion of *Global Inverse Consistency* in the area of constraint satisfaction problems, as recently proposed by Bessiere *et al.* [BFL13]. A constraint satisfaction problem is global inverse consistent *iff* for every value in the domains of its variables, there exists a solution of the constraint satisfaction problem with such value. A value in a domain for which there is no solution is called non-global inverse consistent. Thus, detecting a local range inconsistency in a feature model is equivalent to detecting that there exists a non-global inverse consistent value in the constraint satisfaction problem representing the feature model. The benefit of relating our approach to the one on global inverse consistency is that it enables the reuse of the tools to maintain global inverse consistency in constraint satisfaction problems, to detect such inconsistencies in cardinality-based feature models.

### 5.4.2 From Global Range to Local Range Inconsistency

There exists a relationship between local range and global range consistencies. The latter can be expressed using the former on a normalized feature model $\mathcal{M}'$, in which feature cardinalities define the number of instances of that feature in the product, thus taking into account the cardinality of the parent feature. We translate a cardinality-based feature model $\mathcal{M}$ into the normalized cardinality-based feature model $\mathcal{M}'$, such as in $\mathcal{M}'$, the set of features remains the same than in $\mathcal{M}$, but feature cardinality ranges are updated to describe the number of feature instances. Such a translation $\mathcal{T}$ is written

$$\mathcal{T} : \mathcal{M} = (\mathcal{F}, \omega, \varphi) \rightarrow \mathcal{M}' = (\mathcal{F}, \omega', \varphi)$$

with:

$$\begin{aligned} \omega'(f) &= \omega(f) \textit{ iff } (\texttt{parent}(f) = \varnothing) \\ &= \omega(f) \times \omega'(\texttt{parent}(f)) \end{aligned}$$

$\omega'(f)$ computes exactly $\Omega(f)$. If a feature has no parent, then $\omega(f)$ corresponds to the range of occurrences of $f$ in the product. If a feature has a parent, then the range of occurrences of $f$ in the product depends on the number of instances of that parent (we multiply the ranges together).

For instance, regarding Figure 5.9, the idea is to replace the initial range $[0..2]$ of `B` feature cardinality in $\mathcal{M}$ with $[0..6]$ in $\mathcal{M}'$, where $0 = 0 * 0$ and $6 = 2 * 3$. Checking the local range consistency of $\mathcal{M}'$ is, hence, performed regarding the range of feature instances ($[0..6]$ is not the real cardinality of `B` but a range defining how many instances of `B` can be configured).

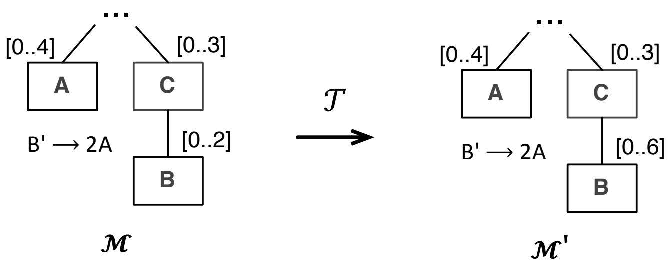


Figure 5.9: Updating a Constraint

In this example, the feature model $\mathcal{M}'$ is not locally consistent because there exists no product with more than two instances of `B`. We can show that checking global consistency in $\mathcal{M}$ is equivalent to checking local consistency in $\mathcal{M}'$. The relationship between the global range consistency and the local range one can thus be summarized as follows:

**Property 1.** CONSISTENCIES RELATIONSHIP
*Checking the global range consistency of a feature model $\mathcal{M}$ is equivalent to checking the local range consistency of $\mathcal{T}(\mathcal{M})$.*

Note that this result does not hold in general (*i.e.*, for any model), but holds in feature modeling where feature order is not considered. Let us consider the example depicted in Figure 5.10.

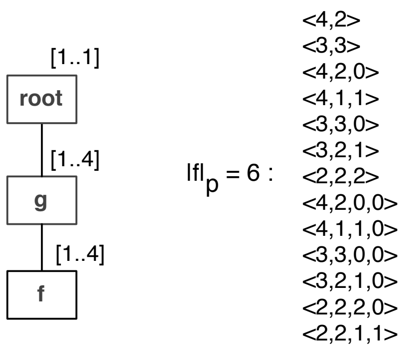


Figure 5.10

As you can notice from the figure, there are 13 different ways to get 6 instances of feature $f$. For example, 2 instances of $g$ with 4 instances of $f$ for the first instance of $g$ and 2 instances of $f$ for the second one. Global range consistency ensures that all those 13 configurations are possible while local range consistency only guarantees that at least one configuration for a given number of instances of $g$ is possible. For instance, suppose that configuration `<4,1,1>` is not possible. In that case, the feature model would not be globally range consistent but locally range consistent. Such case cannot happen in our context since the cardinality-based constraints only focus on the number of instances of each feature, not on their precise location in the product.

*Proof.* We first prove that if $\mathcal{M}$ is globally range consistent then $\mathcal{T}(\mathcal{M})$ is locally range consistent (defined for the sake of simplicity as *GRC($\mathcal{M}$) $\Rightarrow$ LRC($\mathcal{T}(\mathcal{M})$)*). Then, we prove that if $\mathcal{T}(\mathcal{M})$ is locally range consistent, then $\mathcal{M}$ is globally range consistent, *i.e.*, *LRC($\mathcal{T}(\mathcal{M})$ $\Rightarrow$ GRC($\mathcal{M}$)*.

■ *GRC($\mathcal{M}$) $\Rightarrow$ LRC($\mathcal{T}(\mathcal{M})$), ad absurdum.*
We suppose that $\mathcal{M}$ is globally range consistent but that $\mathcal{T}(\mathcal{M})$ is not locally range consistent. Thus, there exists a feature $f$ and a $k \in \omega'(f)$, such as no product contains $k$ instances of $f$ ($\nexists\, p \in \mathcal{P} \mid |f|_p = k$). This holds for any number of instances of $g$.
Let us take into consideration a specific one, $j \in \Omega(g)$. We know that $\sum_{i=1}^{j} x_i = k$. For all tuple $(x_1, ..., x_j)$ such that $\sum_{i=1}^{j} x_i = k$, there is thus no product with $|g|_p = j$ and $|f|_p = x_i$. Therefore, $\nexists\, p \in \mathcal{P} \mid |g|_p = j \wedge \bigwedge_{i=1}^{j} |f_i|_p = x_i$, which contradicts the hypothesis.

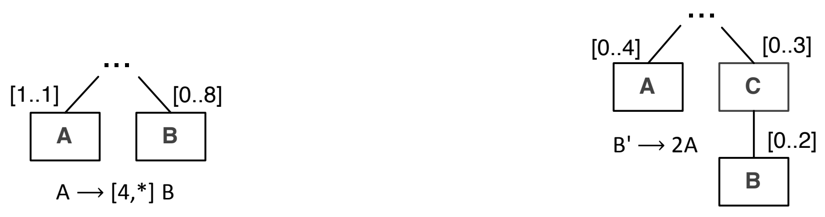
■ *LRC($\mathcal{T}(\mathcal{M})$ $\Rightarrow$ GRC($\mathcal{M}$), ad absurdum.*
We suppose that $\mathcal{T}(\mathcal{M})$ is locally range consistent but that $\mathcal{M}$ is not globally range consistent. There exists a feature $f$ which is not the root, with a parent feature $g$ ($g = \texttt{parent}(f)$), a $k \in \omega(f)$ and an $l \in \Omega(g)$, such as no product contains $k$ instances of $f$ in the $l^{th}$ instance

of $g$, *i.e.*, $\nexists\, p \in \mathcal{P} \mid |f_l|_p = k$. There exists a $j$ that corresponds to the number of instances of $g$ such as $\sum_{i=1}^{j} x_i = k'$. We compute $\omega'(f) = \omega(f) \times \Omega(g)$. For one value of $\omega'(f)$, there are several possible configurations for the same feature and its parent as explained earlier. Since our constraints do not allow us to discriminate the configurations, then $k'$ is not a valid number of occurrences for $\mathcal{M}$. Thus, we know that $\exists\, k' \in \omega'(f) \mid \nexists\, p \in \mathcal{P} \mid |f|_p = k'$, which contradicts the hypothesis. $\square$

In the next section, we describe our tool support to detect and explain such inconsistencies, in particular by leveraging the proposed property.

## 5.5 Implementing Local Range Consistency

As well as the description of CardEx feature models in Chapter 4, Section 4.4, we rely on constraint programming to express and solve the global inverse consistency of a cardinality-based feature models. Those feature models are thus translated into a constraint satisfaction problem to reason on their consistency. To illustrate our approach, we translate cardinality-based feature models into the textual format of the BR4CP (*Business Recommendation for Configurable Products*) project [BR4]. Even though the BR4CP project supports two different formats, *i.e.*, CSP (XML format) and Aralia (textual) [DR97], we consider the latter as more intuitive since its syntax is closer to feature modeling constraints than the XML one, and makes the translation easier. We take as illustrative examples the two cardinality-based feature models depicted here in Figure 5.11 again for convenience, $\mathcal{M}_{LRI}$ and $\mathcal{M}_{GRI}$, respectively describing a locally range inconsistent and a globally range inconsistent feature model.



(a) $\mathcal{M}_{LRI}$ (b) $\mathcal{M}_{GRI}$

Figure 5.11: Cardinality Inconsistencies

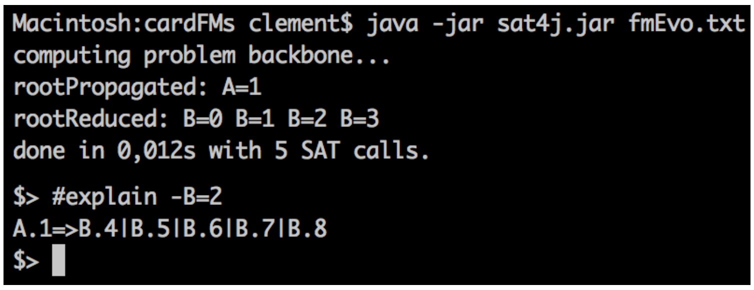
Transforming $\mathcal{M}_{LRI}$ is straightforward. Relying on the previously defined translation $\mathcal{T}$, the LRI feature model $\mathcal{M}_{LRI}$ is translated into the feature model $\mathcal{M}'_{LRI}$ as described by Listing 5.1 using the Aralia textual format.

Lines 1 and 2 define the variables of the model, `A` and `B`, representing features A and B respectively. Values `1,1` just before the left square bracket specify how many values can be

```
#(1,1,[A.1]);
#(1,1,[B.0, B.1, B.2, B.3, B.4, B.5, B.6, B.7, B.8]);
(A.1 => B.4 | B.5 | B.6 | B.7 | B.8);
```

Listing 5.1: $\mathcal{M}'_{LRI}$ described as a textual configuration problem

selected in the variable range, *i.e.*, *at least 1* and *at most 1*. Since feature A is mandatory, its unique possible value is 1, *i.e.*, `[A.1]`, contrarily to feature B which holds as cardinality the range [0..8]. Line 3 describes the constraint `A` $\rightarrow$ [4,8] `B`. Thus, if the value of variable `A` is 1, then the value of variable `B` is either 4, 5, 6, 7 or 8.



```
Macintosh:cardFMs clement$ java -jar sat4j.jar fmEvo.txt
computing problem backbone...
rootPropagated: A=1
rootReduced: B=0 B=1 B=2 B=3
done in 0,012s with 5 SAT calls.
$> #explain -B=2
A.1=>B.4|B.5|B.6|B.7|B.8
$> 
```

Figure 5.12: Inconsistency detection and explanation for $\mathcal{M}'_{LRI}$

To check the consistency of $\mathcal{M}'_{LRI}$ in an automated way, one need to rely on a tool able to detect a global inverse inconsistency. In the BR4CP project, there exist two of them, either based on SAT or CSP, *e.g.*, Sat4j [BP10] or Abscon [BFL13] respectively. However, since Sat4j provides in addition an explanation engine, our approach therefore relies on this tool support. Indeed, once the consistency of the cardinality-based feature model checked, we help the user understanding what is the problem (if any) by giving her feedback. For that, we rely on the notion of *explanation*, *e.g.*, explanation about the lower bound with value $n$ for variable $x$, well-known in the constraint programming community [JO01]. For instance, Figure 5.12 depicts the use of Sat4j to detect the inconsistency in $\mathcal{M}'_{LRI}$. Several information are given about $\mathcal{M}'_{LRI}$. As expected, the value of variable `A` is always 1 (*rootPropagated: A=1*). The next line is an inconsistency detection, meaning that the range value of variable `B` has been reduced, since values 0, 1, 2 and 3 are unreachable. To understand where does such an inconsistency come from, the explanation mechanism is called using the *#explain* command. In this example, an explanation is asked to understand why variable `B` cannot be equal to 2 (or more precisely, why "*not B equals 2*"). As a result, the constraint leading to the inconsistency is displayed: if variable `A` equals 1, then the value of `B` is greater or equal to 4.

To find an explanation about the inconsistency of a feature cardinality, the related feature model must first be translated to a propositional formula in a conjunction of constraints. Let

$F$ be a conjunction of constraints defining a cardinality-based feature model, with $V$ the set of variables for the model. Then, finding a value non-global inverse consistent in $F$ is defined as follows:

$$\begin{aligned}&\exists\, v \in V | F \vDash \neg v\\ &\Leftrightarrow F \wedge v \vDash \bot\\ &\Leftrightarrow F \wedge v \textit{ is unsatisfiable}\end{aligned}$$

An explanation $F'$ on the unsatisfiability of $F \wedge v$ is said *minimal unsatisfiable subformula*. The problem of finding a minimal unsatisfiable subformula is an active area of research [BLMS12] and is defined as follows:

- $F' \subseteq F$;
- $F' \wedge v \vDash \bot$;
- $\forall f \in F', F' \setminus \{f\} \nvDash \bot$;

In practice, the set $F$ of constraints may be divided into two sets $D$ and $C$. For instance, $D$ may represent knowledge or integrity constraints, while $C$ may represent beliefs or relaxable constraints. In such context, the definition of minimal unsatisfiable subformula becomes:

- $F = D \cup C; D \cap C = \emptyset$;
- $F' \subseteq C$;
- $D \wedge F' \wedge v \vDash \bot$;
- $\forall f \in F', D \wedge F' \setminus \{f\} \nvDash \bot$;

To illustrate how such a minimal unsatisfiable subformula is found, and therefore how an explanation is given about the non-global inverse consistency of a value, let us take as example the $\mathcal{M}_{LRI}$ feature model, depicted by Figure 5.11a and described in Listing 5.1. The translation of this feature model into a conjunction of constraints is given in Listing 5.2.

```
A1 = 1
B0 + B1 + B2 + B3 + B4 + B5 + B6 + B7 + B8 = 1
¬A1 ∨ B4 ∨ B5 ∨ B6 ∨ B7 ∨ B8
```

Listing 5.2: $\mathcal{M}_{LRI}$ defined as a CNF formula

The inconsistency detection is obvious. Since A1 must be equal to 1 (line 1), then line 3 forces B4, B5, B6, B7 or B8 to be greater or equal to 1. Thus, B0, B1, B2 or B3 cannot be equal to 1 (line 2), because of the constraint expressed line 3. Note that the three constraints form a minimal unsatisfiable subformula. However, the tool reports only the last constraint (see Figure 5.12) because lines 1 and 2 correspond to definitions belonging to $D$.

## 5.6 Improving the Encoding

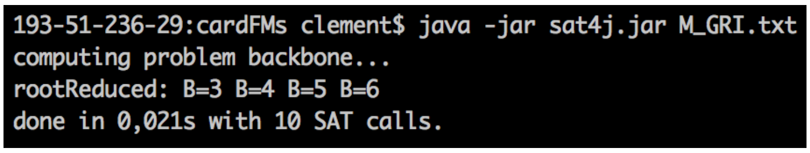
Let us now consider the $\mathcal{M}_{GRI}$ cardinality-based feature model. Relying on the translation $\mathcal{T}$, $\mathcal{M}_{GRI}$ is translated into the feature model $\mathcal{M}'_{GRI}$. Listing 5.3 describes $\mathcal{M}'_{GRI}$ in the Aralia textual format. Lines 1, 2, 3 define the variables of the model, `A`, `B` and `C`. As described in Section 5.4.2, the domains of the variables now represent the instances that can be configured, *e.g.*, 6 instances of feature B.

```
1  /* Variables */
2  #(1,1,[A.0,A.1,A.2,A.3,A.4]);
3  #(1,1,[C.0,C.1,C.2,C.3]);
4  #(1,1,[B.0,B.1,B.2,B.3,B.4,B.5,B.6]);
5  /* Child-Parent relationships */
6  (-B.0 => -C.0);
7  /* Constraint B' --> 2A */
8  (B.1 => (-A.0 & -A.1));
9  (B.2 => (-A.0 & -A.1 & -A.2 & -A.3));
10 (B.3 => (-A.0 & -A.1 & -A.2 & -A.3 & -A.4));
11 (B.4 => (-A.0 & -A.1 & -A.2 & -A.3 & -A.4));
12 (B.5 => (-A.0 & -A.1 & -A.2 & -A.3 & -A.4));
13 (B.6 => (-A.0 & -A.1 & -A.2 & -A.3 & -A.4));
```

Listing 5.3: $\mathcal{M}'_{GRI}$ described as a textual configuration problem

Line 5 defines the relationship between features B and C, describing that if the value of `B` is not equal to 0, then the value of variable `C` must no be equal to 0 (we provided the list of translation rules from feature models to CSP in Chapter 4, Section 4.4). In other words, if feature B is selected, then feature C must be selected. Lines 7 to 12 describe the constraint requiring twice more instances of feature A than instances of feature B. For instance, if variable `B` is equal to 2, then the value of variable `A` must not be lower than 4 (line 9). Figure 5.13 depicts the use of Sat4j to reason on the consistency of $\mathcal{M}'_{GRI}$.

As expected, there can never be more than 2 instances of B in the configuration, as depicted by the third line *rootReduced: B=3 B=4 B=5 B=6*. Even though this result is correct regarding the described feature model, we think that it is not enough to help the designer



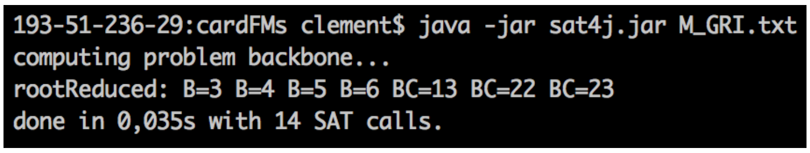
```
193-51-236-29:cardFMs clement$ java -jar sat4j.jar M_GRI.txt
computing problem backbone...
rootReduced: B=3 B=4 B=5 B=6
done in 0,021s with 10 SAT calls.
```

Figure 5.13: Inconsistency detection for $\mathcal{M}'_{GRI}$

understand the underlying inconsistency. Indeed, one can see this inconsistency in two perspectives. First, in terms of instances. As we just saw, there cannot be more than 2 instances of B. Second, in terms of configuration options. The result depicted by Figure 5.13 is not expressive enough to understand which configurations are not allowed in the related feature model. Therefore, to deal with this issue, we propose to add a fictional variable in the cardinality-based feature model $\mathcal{M}'$, representing all possible combinations of feature cardinality when a global range consistency must be checked. For instance, when dealing with $\mathcal{M}_{GRI}$, the translated $\mathcal{M}'_{GRI}$ is yield as depicted by Listing 5.4.

Lines 1 to 13 are the same than in Listing 5.3. The fictional variable, `BC`, is added line 16. It takes as values all the possible combinations of B and C. For example, `BC.21` means two instances of B and one instance of C. Thus, `BC.10` is not possible since there cannot be one instance of B if there is no instance of C. Then, lines 18 to 27 describe the constraints related to the fictional variable. They describe how many instances of feature B correspond to the related combination. For example, `(BC.21 => B.2);` line 25 means that the combination of two instances of B for one instance of C leads to two instances of B. The same amount of B can be obtained by using a different combination, as depicted line 23.

Now, when reasoning on the consistency of $\mathcal{M}'_{GRI}$, we get a more detailed and helpful result, as depicted by Figure 5.14. First, it gives the number of instances of B that cannot be configured, *i.e.*, *rootReduced: B=3 B=4 B=5 B=6*, but also the related combinations, not allowed regarding the cardinalities and constraints defined in the feature models, *i.e.*, *rootReduced: BC=13 BC=22 BC=23*. For instance, a configuration with the cardinality of B and C set to 2 and 3 respectively is not possible.



```
193-51-236-29:cardFMs clement$ java -jar sat4j.jar M_GRI.txt
computing problem backbone...
rootReduced: B=3 B=4 B=5 B=6 BC=13 BC=22 BC=23
done in 0,035s with 14 SAT calls.
```

Figure 5.14: Inconsistency detection for the "user-friendly" $\mathcal{M}'_{GRI}$

```
/* Variables */
#(1,1,[A.0,A.1,A.2,A.3,A.4]);
#(1,1,[C.0,C.1,C.2,C.3]);
#(1,1,[B.0,B.1,B.2,B.3,B.4,B.5,B.6]);
/* Child-Parent relationships */
(-B.0 => -C.0);
/* Constraint B' --> 2A */
(B.1 => (-A.0 & -A.1));
(B.2 => (-A.0 & -A.1 & -A.2 & -A.3));
(B.3 => (-A.0 & -A.1 & -A.2 & -A.3 & -A.4));
(B.4 => (-A.0 & -A.1 & -A.2 & -A.3 & -A.4));
(B.5 => (-A.0 & -A.1 & -A.2 & -A.3 & -A.4));
(B.6 => (-A.0 & -A.1 & -A.2 & -A.3 & -A.4));

/* Fictional Variable */
#(1,1,[BC.00,BC.01,BC.02,BC.03,BC.11,BC.12,BC.13,BC.21,BC.22,BC
    .23]);
/* Add constraints for fictional BC variable */
(BC.00 => B.0);
(BC.01 => B.0);
(BC.02 => B.0);
(BC.03 => B.0);
(BC.11 => B.1);
(BC.12 => B.2);
(BC.13 => B.3);
(BC.21 => B.2);
(BC.22 => B.4);
(BC.23 => B.6);
```

Listing 5.4: $\mathcal{M}'_{GRI}$ described as a textual configuration problem

## 5.7 Challenges Revisited

We have presented our approach for detecting cardinality inconsistencies in cardinality-based feature models when editing them and described a formal approach to automate such detections. Let us now revisit the challenges identified in Section 5.2 and discuss how our approach faces them.

1. **Detect cardinality inconsistencies.** To face this challenge, we first provide a catalog of cardinality-based feature model edits, thus describing which edits may lead to an inconsistency. Knowing which edits do not lead to inconsistent feature cardinalities

is useful because it provides the possibility to save effort by not checking the feature model consistency after one of those edits has been performed. Then, we define two kinds of cardinality inconsistency, local and global range inconsistency. While the former depends on the considered feature cardinality range, the latter takes into account the hierarchy of ranges.

2. **Explain the inconsistencies.** To face this challenge, our approach is twofold. First, we rely on existing mechanisms of explanation used in constraint programming to describe why an inconsistency arise. Second, as the explanation may not be precise enough to understand which configuration is not allowed and the number of involved instances, we use a fictional variable representing both feature instances and feature range combinations. Using this variable, one can know which range combination cannot be done and which related instances number cannot be reached.

3. **Provide an automated support.** To face this challenge, we rely on the notion of global inverse consistency in the area of constraint satisfaction problem [BFL13]. As detecting a local range inconsistency is equivalent to detect a non-global inverse consistent value in the constraint satisfaction problem representing the feature model, one can reuse existing tools used to maintain global inverse consistency. We thus describe a way to translate a feature model $\mathcal{M}$ with a global range consistency to a feature model $\mathcal{M}'$ with a local range consistency.

## 5.8 Summary

This chapter presented our approach to detect and explain range inconsistencies in cardinality-based feature models. As revealed by the catalog of edits to cardinality-based feature models we proposed, two kinds of inconsistency may arise, either local or global range inconsistencies. While the former is related to the range of a feature, the latter takes into account the hierarchy of feature ranges.

We then propose to rely on an existing approach to automate these inconsistency detections. This approach checks the global inverse consistency of a constraint satisfaction problem, which is actually similar to checking the local range consistency of a feature model. We thus provide a way to translate a feature model $\mathcal{M}$ to a normalized feature model $\mathcal{M}'$ containing only local range inconsistency. Automating the detection of inconsistencies in $\mathcal{M}'$ is then straightforward.

Finally, we support the user when designing the cardinality-based feature model by giving her/him explanations about the inconsistency, *i.e.*, where and why it arises. We go further in case of global range inconsistencies by providing feedback about which configuration cannot be established regarding the defined cardinality ranges and the related number of feature instances that cannot be reached.

In the next chapter of this dissertation, we present the SALOON platform, our software product lines-based approach for selecting and configuring cloud computing environments.

Chapter 6

# SALOON, a Model-based Approach for Selecting and Configuring Cloud Environments

**Contents**

## 6.1 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 providers available 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* (IaaS) to *Software* including *Platform* (PaaS) [AFG+09, BYV+09]. At 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, database configuration, etc. Regarding platforms provided by PaaS clouds, the configuration concern only focuses on software that compose this platform: which database(s), application server(s), compilation tool, libraries, etc. This layered model therefore offers many configuration and dimension choices, for the application to be deployed as well as the configurable runtime environments [MG11]. Thus, 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. Selecting a suitable cloud environment and dealing with its variability leads to complex and error-prone configuration choices that are usually made in an *ad hoc* manner.

To address these issues, we have developed a platform called SALOON, for *SoftwAre product Lines for clOud cOmputiNg*. As its name suggests, SALOON relies on software product lines principles to help its user selecting and configuring cloud environments according to its requirements. In particular, SALOON leverages software product lines to ensure reliability in those selection and configuration processes, relying on CardEx feature models to describe cloud variability and using cloud environment configuration files as software artifacts to derive a proper cloud configuration. This chapter presents SALOON. Since 2013, SALOON is used in particular as a cloud environments selection and configuration platform by the European IP PaaSage project [Paa13].

The chapter is structured as follows: Section 6.2 describes the challenges that have to be faced when dealing with clouds variability. Section 6.3 provides an overview of SALOON, describing the different roles and responsibilities as well as the different concerns of the platform. Section 6.4 gives more details about the architecture of SALOON, in particular the different metamodels involved and how they are linked together. Section 6.5 explains the cost estimation and derivation engines of SALOON, illustrated with IaaS and PaaS environments configurations. Finally, Section 6.6 discusses several concerns regarding the platform while Section 6.7 concludes the chapter.

## 6.2 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 help those developers selecting and configuring a suitable cloud environment. To achieve these objectives, we identify the following challenges SALOON must helps the developers to deal with:

1. **Find 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.

2. **Find 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.

3. **Ensure a reliable configuration.** Once a cloud environment is selected and there exists one 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.

## 6.3 SALOON in a Nutshell

This section provides a global description of SALOON, presenting its architecture and explaining how different actors interact with the platform. Figure 6.1 depicts an overview of SALOON.

SALOON relies on three main characteristics. First, cloud environments are described as feature models. More precisely, feature models extended with attributes and cardinality as described in Chapter 4, *i.e.*, CardEx feature models. Second, cloud configuration files and configuration scripts are used as assets of the feature models. Therefore, there is one software product line per cloud environment, since the feature model and the related assets are dedicated to one specific cloud environment. Finally, the reification and gathering of cloud environment provided functionalities into a Cloud Knowledge Model, which are mapped to each cloud feature model to automate the feature selection process.

### 6.3.1 Roles

Our approach distinguishes between two roles, *domain architects* and *developer*.

- **Domain Architect.** The architect is expert in the particular domain targeted by the software product line, the cloud computing one in SALOON. He/She is responsible for

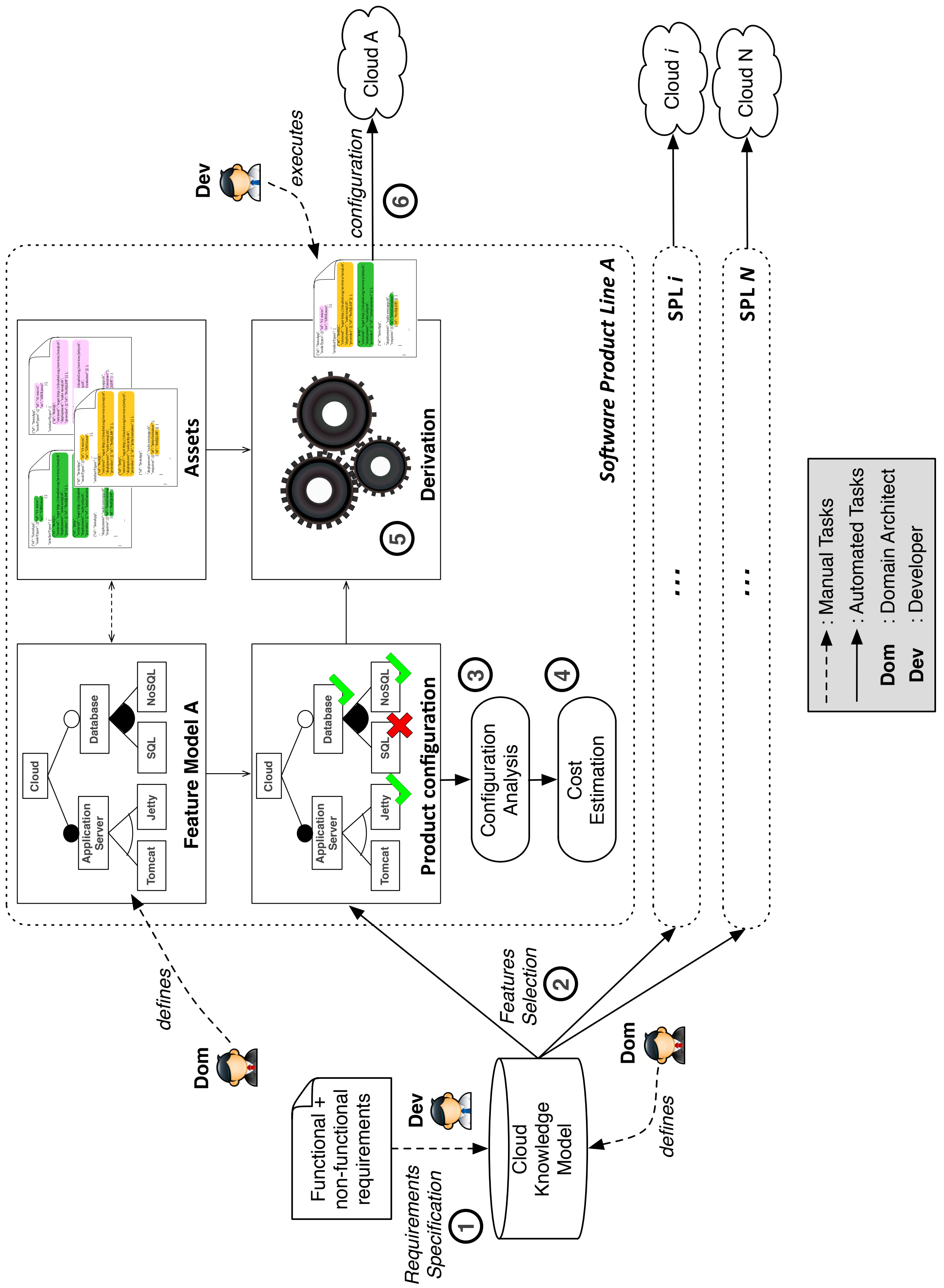


Figure 6.1: SALOON overview

defining both the cloud feature models and the Cloud Knowledge Model. A domain architect has all the 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 used in all SALOON feature models.

- **Developer.** 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.

### 6.3.2 Stages

In addition to the roles, Figure 6.1 also illustrates the different steps to get from a requirements set to a configured cloud environment. These steps are presented below.

① **Requirements specification.** The developer uses the Cloud Knowledge Model to select the required cloud services, *i.e.*, cloud services that comply with the functional and non-functional requirements of the application to deploy, *e.g.*, the Tomcat application server.

② **Features selection.** Regarding elements selected in the Cloud Knowledge Model, the related features are selected in the different feature models, if provided by the cloud environment. For instance, the Tomcat feature will be selected in all feature models providing this feature. The features selection process is an automated step, relying on defined mapping relationships between the Cloud Knowledge Model and the different feature models.

③ **Configuration analysis.** When at least one feature has been selected in a given feature model, SALOON provides a support to search if a valid product configuration exists with this feature, *e.g.*, if there exists a valid configuration for the Heroku cloud providing the Tomcat application server support. This step is automated as well, relying on the translation from CardEx feature models to CSP as defined in Chapter 4, Section 4.4.

④ **Cost estimation.** Thanks to a pricing model linked to each feature model, SALOON automatically estimates the cost of the configuration found at step ③. This estimation can help the user selecting a cloud when several ones are suitable to host the application.

⑤ **Product derivation.** Once a cloud environment has been selected by the user, the derivation engine associated with this cloud is launched to derive the related configuration files and scripts, *e.g.*, a *system.properties* file and a set of configuration commands to be executed.

⑥ **Configuration scripts execution.** While configuration files are required for a proper configuration of the cloud environment, configuration scripts have to be executed by the user to configure the environment, *e.g.*, through a command line interface.

In the following sections, we describe with more details the different concerns of our approach.

## 6.4 From Requirements Specification to Features Selection

As described in the previous section SALOON relies on various kinds of models, *i.e.*, feature models and the Cloud Knowledge Model. In addition to these models, mapping relationships between them used to automate the features selection process (stage ②) are described as models as well. This section describes in details SALOON models and metamodels and illustrates their use with several cloud environment concrete examples.

### 6.4.1 A Model-based Approach

The SALOON platform relies on three metamodels: the *Domain Knowledge Metamodel*, the *Mapping Metamodel* and the *CardEx Metamodel*, as depicted by Figure 6.2 in green, pink and orange respectively. For the sake of simplicity, only meta-classes involved in relationships with the *Mapping Metamodel* are depicted regarding the *CardEx Metamodel*, but a full picture of it can be found in Chapter 4, Section 4.3.1.

#### The Domain Knowledge Metamodel

The root of the metamodel is the `DomainKnowledge` meta-class, which contains a set of `Concepts`. Each `Concept` is given a *name* and may have `subConcepts`. A `Concept` can be of type `Concept`, *e.g.*, Java, `CountableElement` or `QuantifiableElement`. `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. The *selected* attribute of the `Concept` meta-class is only defined for the developer use purpose. When an instance of this metamodel is defined, *e.g.*, the Cloud Knowledge Model, it is used by the developer to select all required functionalities. This is done through the *selected* Boolean attribute. `Constraints` can also be defined, to specify whether a concept `Implies` or `Excludes` the selection of another concept. For instance, if the application to deploy is written in *ASP.NET*, then the application server involved in the configuration cannot be Jetty. Then, the selection of the former in the Cloud Knowledge Model implies the latter not to be selected.

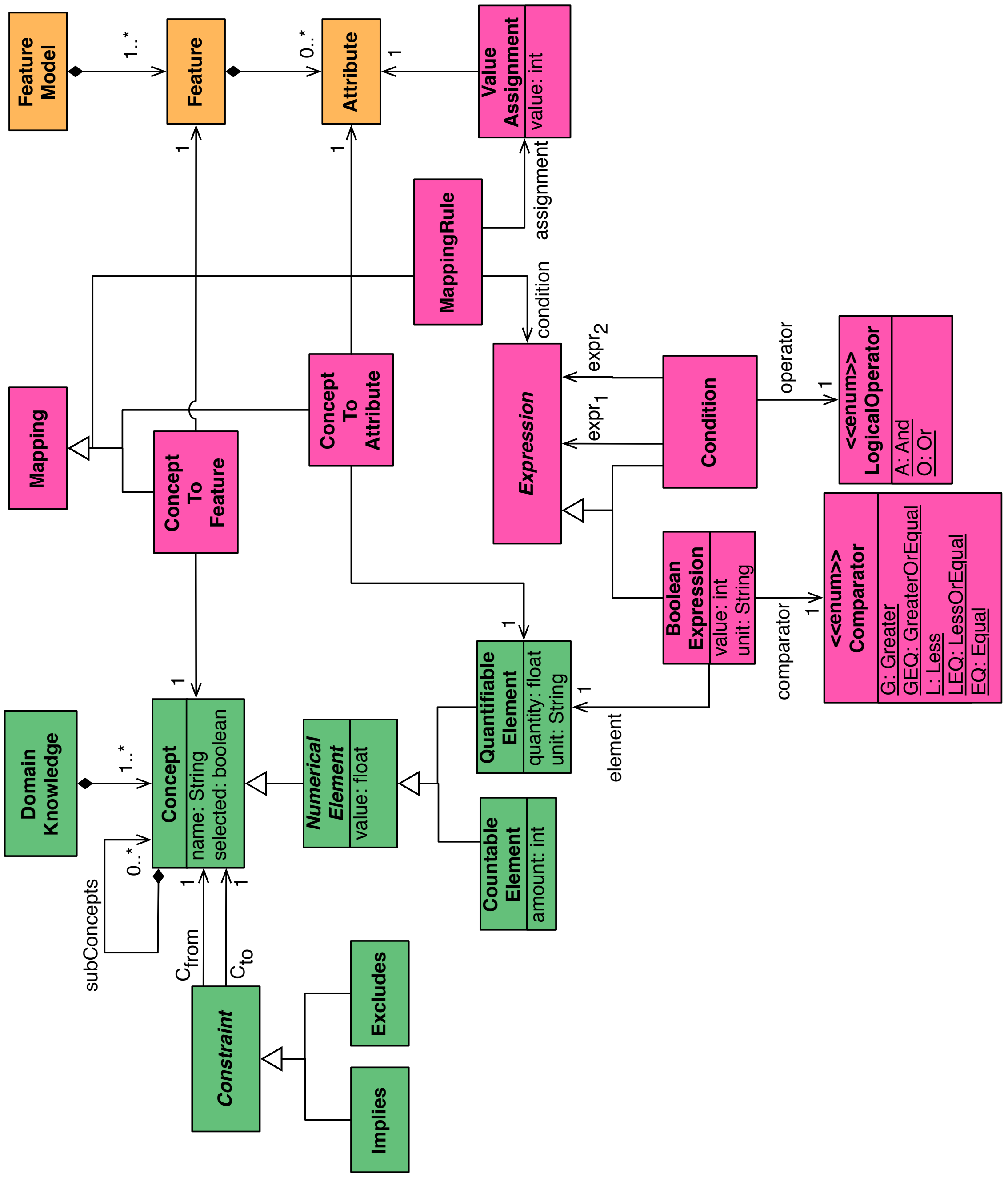


Figure 6.2: The SALOON metamodels

## The Mapping Metamodel

The `Mapping` relationships link concepts in the Cloud Knowledge Model with features or attributes in the feature models. There are threes types of relationships: `ConceptToFeature`, `ConceptToAttribute` and `MappingRule`. The two formers are used, without surprise, to link a concept with a feature or a concept with an attribute respectively. For instance, the concept *Java* in the Cloud Knowledge Model is linked with the *Java* feature in the feature model describing the Jelastic cloud environment. Note that the name of the concept and the feature's one are not necessarily the same. For instance, the *Load Balancer* concept can be linked to the *Nginx* feature, the *Load balancer* feature or the *HAProxy* feature in the Jelastic, Heroku and OpenShift feature models respectively. Therefore, the knowledge of the semantics of feature has to be known for such links to be established. Thus, such a role is given to the domain architects who, once the feature model is described, provide the related relationship links to the SALOON platform. The last type of mapping, `MappingRule`, is used to link concepts with attributes whose type is an enumeration. For instance, consider a `QuantifiableElement`, *e.g.*, *RAM*, and a feature attribute, *e.g.*, *memorySize*. If the developer requires *RAM* = 500 MB in the Cloud Knowledge Model, and *memorySize* is provided as memory block of 512, 1024 or 2048 MB. Then, should the relationship link be set for 1024 or 2048 (512 does not fit the requirements)? In such a case, one cannot know which value to use unless a mapping rule is defined. A *condition* must be described for a value to be assigned to the mapping rule. The condition is a `BooleanExpression` or a set of `BooleanExpression`s connected with `LogicalOperator`s. A `BooleanExpression` is satisfied if the relationship between the `QuantifiableElement` *quantity* attribute, the `BooleanExpression` *value* attribute and the defined `Comparator` is true. An example is depicted below.

#### The Feature Model Metamodel

This metamodel describes meta-classes used to define CardEx feature models, *i.e.*, feature models with attributes and cardinalities and their related constraints. CardEx feature models and the related metamodel are described in Chapter 4.

### 6.4.2 Illustrative Example

To illustrate our approach, let us now take as example the three following cloud environments: Windows Azure, Heroku and OpenShift. The former is a IaaS environment, the two latter are PaaS ones. Figure 6.3 depicts the Cloud Knowledge Model, mapping relationships and the cloud feature models. All of them are instances of the previously described metamodels and, since depicted for illustrative purpose, are not exhaustive (extended versions of these models are available in Appendix A).

The Cloud Knowledge Model is depicted based on the ontology formalism, which is "*a formal, explicit specification of a shared conceptualization*" [CFLGP06]. Thus, every concept is a *Thing*, and concepts are linked together through inheritance relationships or constraints [GDD09]. For instance, the *RAM* concept inherits the *QuantifiableElement* one, while the defined constraint describes that if *ASP.NET* is part of the requirements set, then the *Jetty* application server cannot be used to host the application. For ease of reading, mapping relationships are described as textual format, where the concept name is written in a bold font,

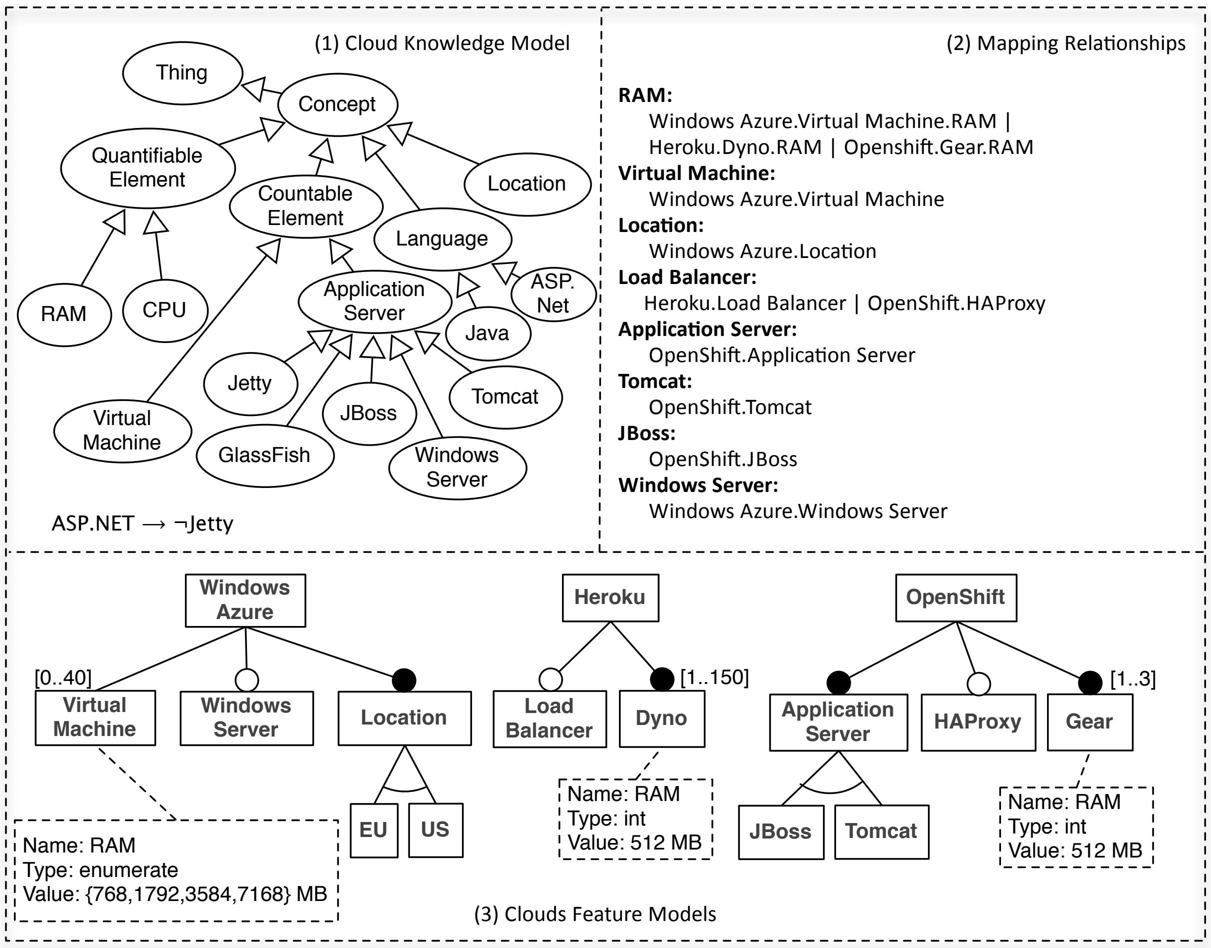


Figure 6.3: Mappings between the Cloud Knowledge Model and the feature models

while features or attributes are written in a normal font, separated by vertical lines when several ones are linked to the same concept.

Let us now consider the following sets of requirements, REQ$_1$: {Windows Server} and REQ$_2$: {8 GB RAM} a developer may define through the Cloud Knowledge Model. Regarding REQ$_1$, only Windows Azure fulfills this requirement. Indeed, it is the only one cloud to provide such a support, as depicted by its related feature model, and there is only one mapping relationship from the *Windows Server* concept to the Windows Azure *Windows Server* feature. Based on this mapping relationship, the *Windows Server* feature is automatically selected in the Windows Azure feature model. Since only Windows Azure provides such a support, there is no need to search for a valid configuration in the OpenShift and Heroku feature models. Thus, using such mapping relationships reduces the range of feature models to configure by acting like a filter. Indeed, it avoids searching for a valid configuration for certain feature models whose provided features cannot cope with the requirements set.

Regarding REQ$_2$, the developer relies on the `QuantifiableElement` meta-class attributes (Figure 6.2) to specify the required quantity and unit of *RAM*, *i.e.*, 8 and GB respectively. The *RAM* concept is linked to the *RAM* attribute in each of the three feature models. When a concept is linked to an attribute, the feature selection process is twofold. First, an algorithm determines the required amount of feature instances to fulfill the requirement. Second, when applicable, the feature is selected and its feature cardinality is set to the required value. Listing 6.1 illustrates such a process. Line 7, the value of the required concept is converted to a value in the same unit than the attribute one, *e.g.*, from 8 to 8000 regarding REQ$_2$. Then, the number of feature instances is computed regarding the required amount and the provided attribute value (line 8 and lines 19 to 27). Finally, the feature cardinality is set according to the value found in the previously called method. Thus, for REQ$_2$ to be satisfied, the cardinality of the *Gear* and *Dyno* features must be set to at least 16 (other features or constraints may require some more instances) in the OpenShift and Heroku feature models respectively, since the defined *RAM* attributes provide 512 MB, and 16*512 MB $\geq$ 8 GB. However, this is not possible for OpenShift, since at most 3 *Gear*s can be provided (we consider in this example the OpenShift *Online* offer public cloud). Then, such a cloud environment could not be configured regarding the developer's requirements.

What about Windows Azure? According to Listing 6.1, there are several ways to fulfill REQ$_2$, *e.g.*, two *Virtual Machine*s providing 7168 MB of RAM, three *Virtual Machine*s providing 3584 MB of RAM, etc. The aim of defining a `MappingRule` is to remove such an ambiguity. For instance, Figure 6.4 depicts a mapping rule for this situation as an object diagram that conforms to the metamodels depicted in Figure 6.2.

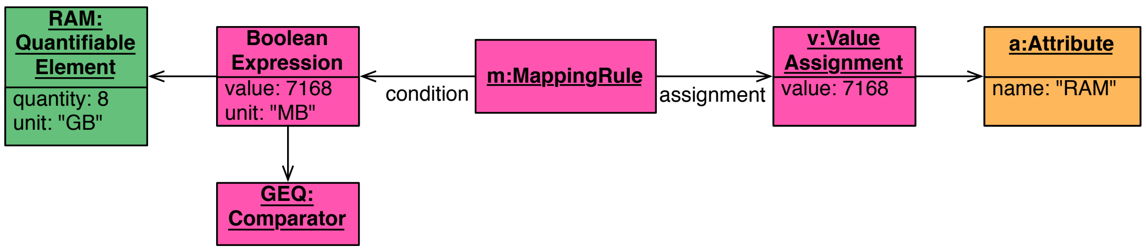


Figure 6.4: A sample mapping rule for the Windows Azure feature model

This mapping rule defines that if the required RAM quantity is greater or equal to 7168 MB, then the value that has to be taken into consideration for the RAM attribute enumeration is 7168. Thus, relying on this mapping rule and applying Listing 6.1, the number of instances of *Virtual Machine*s must be set to 2 to fulfill REQ$_2$. Such mapping rules must be defined by the domain architects, who know how and why such rules must be set, *e.g.*, for technical reason or pricing policy.

Using such a mapping between the Cloud Knowledge Model and the feature models has mainly three benefits. First, it automates the feature selection process. The developer thus does not have to select features by hand in every feature models, which is considerably tedious and error-prone, but simply defines the application requirements once in the

```
/**
 * quant is defined in the Cloud Knowledge Model, e.g. RAM 8 GB.
 * attr is an attribute of a feature in a given feature model.
 */
void mapAttribute(QuantifiableElement quant, Attribute attr) {
   // Convert the quant value to the same unit than attr
   int quantValue = convert(quant, attribute.getUnit());
   int card = getCardinality(attr.getValue(), quantValue);
   // Set the cardinality of the attribute parent feature
   attr.getFeature().setCardinality(card);
}

/**
 * Precondition: attributeValue and requiredValue
 * are expressed in the same unit.
 * attributeValue is the provided resource value.
 * requiredValue is the value defined in the Cloud Knowledge Model
 */
int getCardinality(int attributeValue, int requiredValue) {
   int card = 1;
   int referenceValue = attributeValue;
   while (attributeValue < requiredValue) {
       card++;
       attributeValue = attributeValue + referenceValue;
   }
   return card;
}
```

Listing 6.1: Assignment of feature cardinality regarding attribute value

Cloud Knowledge Model. Second, it bridges the semantic gap between cloud environments by mapping one Cloud Knowledge Model concepts to features in different feature models with the same semantics. For example, features *Load Balancer* and *HAProxy* in Figure 6.3 are mapped to the same Cloud Knowledge Model concept *Load Balancer*, since they are semantically equivalent even if their names differ. Finally, it reduces the range of feature models to be configured by acting like a filter. Indeed, it avoids checking the validity of certain feature models whose configuration can not cope with the requirements set. When a concept cannot be mapped to a feature, then the related feature model is not considered for the rest of the configuration process, since the related cloud environment is unsuitable.

## 6.5 Automated Configuration and Product Derivation

Even if the selection of features in the different feature models is automated regarding the defined mapping relationships, the developer still has to select the final cloud environment. Indeed, several cloud feature model configurations may be valid regarding the given requirements. In such a case, the developer selects a cloud that fulfills her/his requirements relying on additional criteria, *e.g.*, the configuration price.

### 6.5.1 Cost Estimation for Cloud Selection

Once features are selected in the different feature models regarding the defined set of requirements and mapping relationships, SALOON searches for a valid configuration involving those selected features. For instance, regarding $\text{REQ}_2$: {8 GB RAM}, SALOON searches for a valid configuration involving 2 *Virtual Machine* instances or 16 *Dyno* instances in the *Windows Azure* and *Heroku* feature models respectively. This is done by translating the cloud feature models, which are CardEx ones, to the related constraint satisfaction problem as defined in Chapter 4, Section 4.4 and by reasoning on this CSP using a CSP solver.

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. Knowing the price of a configuration represents one of the main concerns when selecting a cloud environment [DHTCB14]. 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. Therefore, SALOON 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 6.2 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.

SALOON can therefore be used as a decision-making tool by the developer, providing an automated support to find suitable cloud configurations regarding a set of functional and non-functional requirements. Then, relying on the SALOON cost estimation engine, those configurations can be filtered, *e.g.*, to select the less expensive one. However, reducing the configuration cost often means reducing the cloud provided resources as well. Thus, in the end, the final choice for the cloud environment to be configured comes to the developer, who then asks SALOON to derivate the related product.

```java
public interface IProviderCostEstimator
                        extends IProviderCostEstimatorFactory{
    public double estimateCost(Configuration conf);
    public double estimateCostPerHour(Configuration conf);
    public double estimateCostPerMonth(Configuration conf);
    public double estimateCostPerYear(Configuration conf);
}
```

Listing 6.2: SALOON cost estimation engine interface

### 6.5.2 Product Derivation in SALOON

In a software product line, features hold as assets reusable software artifacts that are put together to derive the final product during application engineering, *i.e.*, after assets have been implemented (a.k.a. domain engineering). 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. Features can hold none, one or several assets, while an asset can be shared among several features. 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 feature model holds its own dedicated assets, since a configuration file is not valid for two different cloud environments. Even though assets are different from a cloud feature model to another one, the derivation process remains the same for each of them *i.e.*, load the feature model, get selected features and derive the configuration files regarding their assets. SALOON thus provides an automated support to reason on feature model configurations and derive the related products. We illustrate this support with two cloud environments.

#### The Heroku PaaS Case Study

We illustrate the use of such assets through the Heroku PaaS example, depicted by Figure 6.5. For instance, the `Java 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.

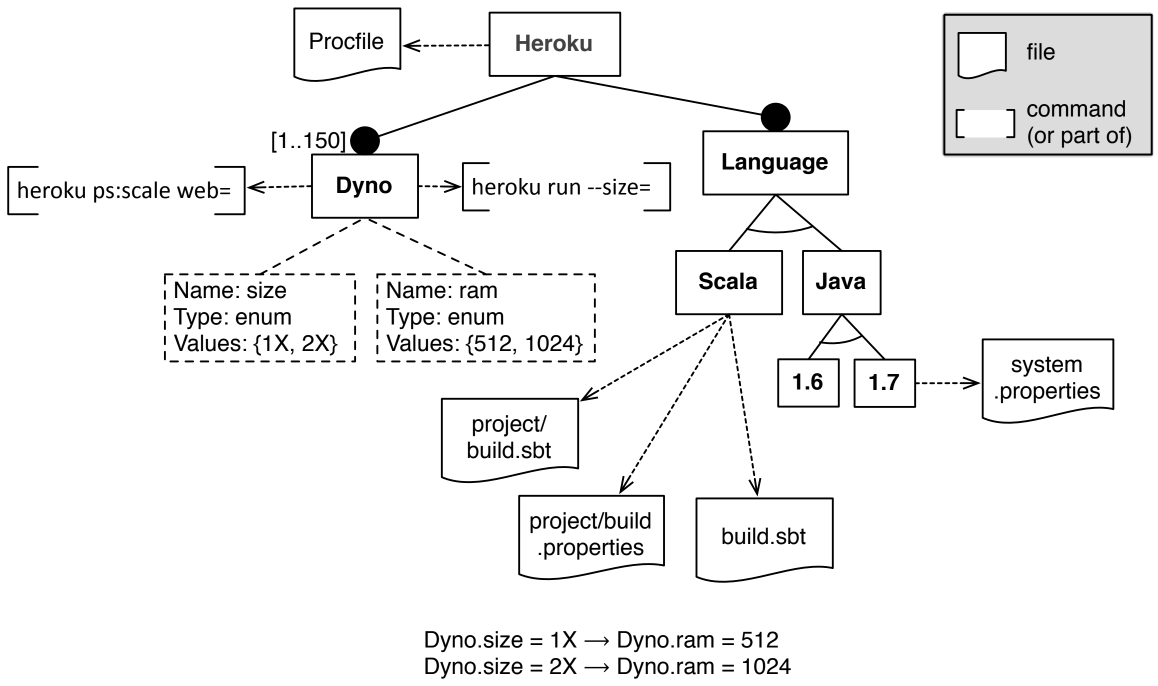
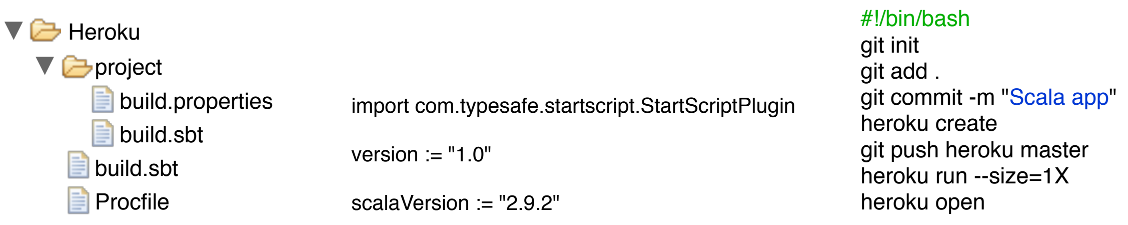


Figure 6.5: The Heroku feature model and its assets (excerpt)

Feature may also holds as asset configuration commands. Regarding the Heroku example, those commands are commands provided by the Heroku SDK, accessible from the developer's command shell. For instance, the `Dyno` feature holds as asset 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 derivates 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 REQ3: {Scala}. 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 derivates the related *commands.sh* file to automate the configuration of the environment regarding the requirements. Figure 6.6 depicts the different files derived by SALOON regarding REQ3.



```
▼ Heroku
    ▼ project
          build.properties
          build.sbt
      build.sbt
      Procfile
```

(a) derived files

```
import com.typesafe.startscript.StartScriptPlugin

version := "1.0"

scalaVersion := "2.9.2"
```

(b) *build.sbt* (excerpt)

```
#!/bin/bash
git init
git add .
git commit -m "Scala app"
heroku create
git push heroku master
heroku run --size=1X
heroku open
```

(c) *commands.sh*

Figure 6.6: derived files to configure Heroku for hosting a Scala application

Figure 6.6a illustrates the derived tree view, with the 4 required configuration files. An excerpt of the *build.sbt* file is depicted by Figure 6.6b, while Figure 6.6c 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` Figure 6.6b. 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 Figure 6.6 (c).

### The Windows Azure IaaS Case Study

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 derivates 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. These configuration steps can be automated, *e.g.*, by relying on the Windows Azure REST API. For instance, the request defined in Listing 6.3 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 6.3: 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 6.4, 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 6.4: *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*, that is, 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, derivate 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 6.5.

```
1 $ sudo apt-get update
2 $ sudo apt-get install tomcat7
```

Listing 6.5: 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.

## 6.6 Challenges Revisited and Discussion

We have presented SALOON, our platform for selecting and configuring cloud environment, relying on software product lines. Let us now revisit the challenges identified in Section 6.2 and discuss how our approach face them.

1. **Find a suitable environment.** To face this challenge, SALOON relies on three main characteristics. First, the modeling of cloud environment provided functionalities as CardEx feature models. Second, the gathering of cloud features, reified as concepts, into a Cloud Knowledge Model allowing the developer to specify functional and non-functional requirements. Finally, the use of mapping relationships between the Cloud Knowledge Model and the different feature models, providing an automated support to search for suitable cloud feature models.

2. **Find a proper configuration.** To face this challenge, SALOON leverages the use of CardEx feature models to describe cloud environment. Indeed, once translated to CSP, finding a valid configuration regarding the set of selected features is well-known, *e.g.*, using off-the-shelf CSP solvers. Then, SALOON provides a cost estimation engine, computing the cost of a given configuration The developer can thus select the configuration that best suits the defined functional and non-functional requirements.

2. **Ensure a reliable configuration.** To face this challenge, SALOON relies on the derivation process of the software product line whose feature model's configuration has been selected by the developer. SALOON thus derives cloud configuration files, together with a set of commands that must be executed by the developer for the cloud environment to be configured, *e.g.*, commands from a cloud SDK. Therefore, relying on SALOON, the developer avoids a tedious and error-prone process usually done manually.

**On the use of a model-based approach.**
The SALOON platform relies on three metamodels, as explained in Section 6.4. The advantage of such an approach is its modularity, since models can be added, removed or updated without impacting other elements. Moreover, such an approach can be used for any domain where several feature models are required to describe the domain variability, and where an automated support is required for features selection and configuration analysis. Thus, what make SALOON tailored for cloud environments selection and configuration are *(i)* the Cloud Knowledge Model and cloud feature models and *(ii)* the artifacts used to yield the final software product, *i.e.*, cloud configuration files and scripts. In addition, such a modularity enables the feature model configuration analysis support to be used independently from the rest of the SALOON platform, as explained in Chapter 4. On the other hand, the main drawback of such an approach is that designing models represents a significant investment. Moreover, those models have to be maintained over time for the platform to work properly, and such a maintenance is error-prone. There are two possible solutions to overcome the first shortcoming. The first one is to reverse-engineer cloud feature models from their web configurator [AAHC14]. The second one is to provide SALOON as a service, where domain architects would have a backend entry point, allowing them to design new cloud feature models and update existing ones. Regarding the second shortcoming about maintaining existing models, SALOON precisely integrates an automated support for reasoning on cardinality-based feature model consistency (see Chapter 5), thus helping architects in this

maintenance.

**On the use of constraints in the Cloud Knowledge Model.**
A question that may arise while reading this chapter is: since constraints can be defined in the Cloud Knowledge, then what is the difference between the Cloud Knowledge Model and the feature models, and why the configuration analysis cannot be properly handled by the Cloud Knowledge Model? Constraints defined in the Cloud Knowledge Model are constraints that are not cloud-specific. Thus, these constraints are shared among every cloud environment. On the other hand, constraints defined in the feature models are cloud-specific, and thus can not be defined in the Cloud Knowledge Model. For instance, a constraint implying the selection of a feature, which is valid for a given feature model, may lead to inconsistencies in another feature model, *e.g.*, the feature does not exist. Cloud configurations thus cannot be analyzed in the Cloud Knowledge Model.

**On the use of a plugin-based approach for the cost estimation engine.**
In SALOON, the configuration cost is computed relying on a dedicated engine. The reader may thus 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.

## 6.7 Summary

This chapter presented SALOON, our approach based on software product lines principles to select and configure cloud environments. SALOON relies on several pillars. First, cloud environments variability is described in CardEx features models, with one CardEx feature model per cloud. As the number of cloud environments is significant, and therefore the number of feature models in SALOON, selecting features manually in each feature model is a tedious and error-prone process. To face this issue, SALOON provides a Cloud Knowledge Model, together with mapping relationships between this model and the feature models. Such mappings automate the features selection in the different feature models, while the SALOON end-user, *i.e.*, the developer, relies on the Cloud Knowledge Model as the entry-point of the platform. The Cloud Knowledge Model, the mapping relationships and the feature models being described as models, it brings an high flexibility and modularity to the platform. Finally, SALOON supports the developer by computing a cost estimation for suitable cloud configurations, and by derivating cloud configuration files and scripts, defined as assets of the feature models.

This chapter concludes the contribution of this dissertation, which started in Chapter 4 with a description of our support for designing and reasoning on CardEx feature models. We have then explained in Chapter 5 how evolving cardinality-based feature models may lead to inconsistencies regarding their cardinalities, and how to detect and explain those inconsistencies. Finally, Chapter 6 described SALOON, our platform to automatically select and configure cloud environments relying on software product lines principles.

The next part of this dissertation is dedicated to give more details on the implementation of our approach, as well as to describe the experimentations we led to evaluate our work.

# Part IV

# Validation

Chapter 7

# Validation

**Contents**

## 7.1 Introduction

In this chapter, we describe the implementation details of SALOON. This includes all the tools related to the different SALOON phases described in Chapter 6, enabling the developer to configure a cloud environment, starting with application requirements specification and obtaining at the end the configuration files and executable scripts, as well as the consistency analysis support we described in Chapter 5. We also report on some experiments we conducted to evaluate SALOON. This evaluation investigates in particular the soundness, the scalability and the practicality of our approach when dealing with numerous cloud environments.

The chapter is structured as follows: Section 7.2 describes all the implementation details about the tools we implemented for SALOON. In Section 7.3, we report on some experiments we conducted to evaluate the platform. Section 7.4 discusses on the advantages and limitations of our approach. Finally, Section 7.5 summarizes and concludes the chapter.

## 7.2 Tool Support

This section presents the different tools and implementation details used at different phases in the software product line engineering process SALOON relies on, from requirements specification to products derivation.

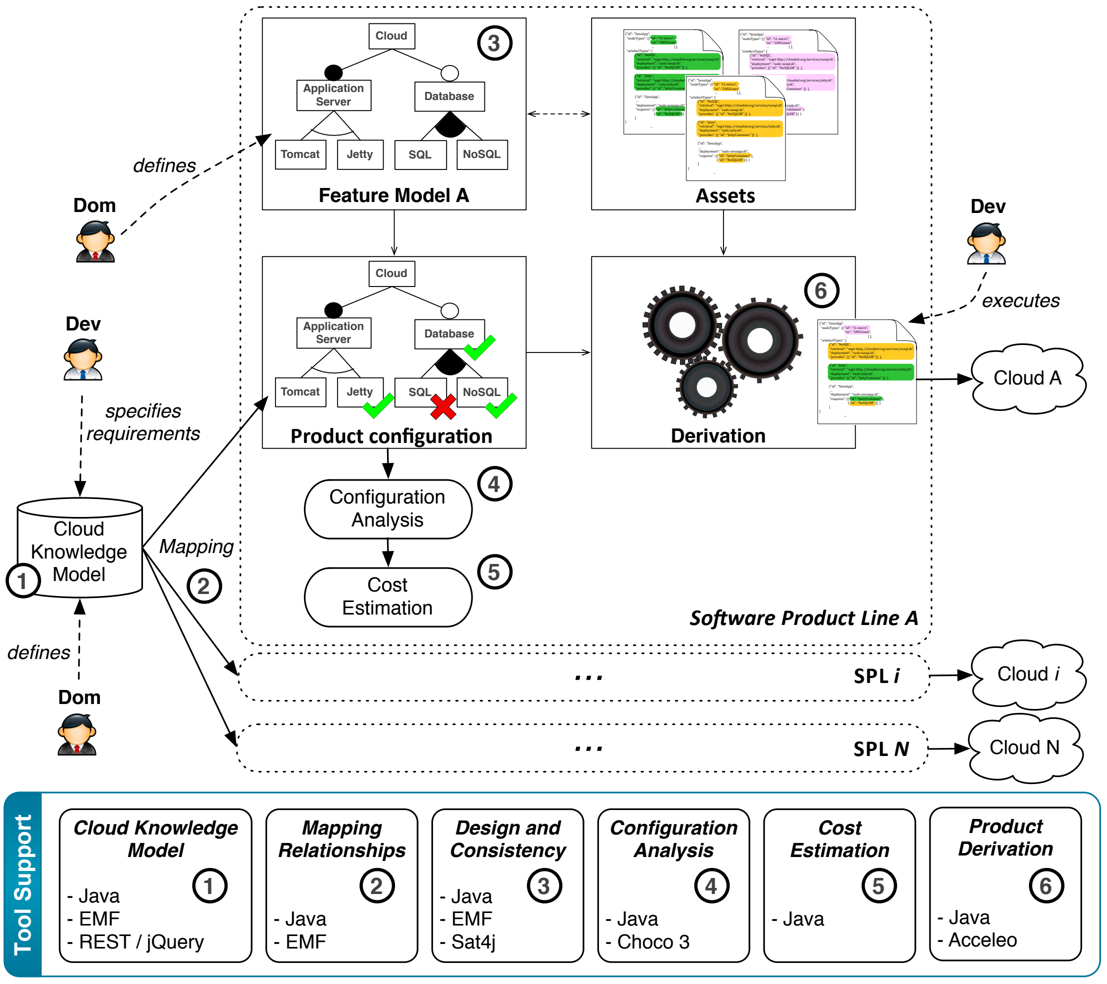


Figure 7.1: Tool support overview

Figure 7.1 depicts the different tool support parts and how they correspond to the different phases of the SALOON architecture. There are six parts in total as follows: ① Cloud Knowledge Model, to specify the application's requirements; ② Mapping Relationships, to link concepts in the Cloud Knowledge Model with features one the feature models; ③ Design and Consistency, that covers feature models definition and analysis of their consistency regarding cardinalities; ④ Configuration Analysis, used to search for a valid configuration regarding selected features; ⑤ Cost Estimation, which provides a mean to estimate the cloud configuration cost; and finally ⑥ Product Derivation, to obtain the configuration files and scripts of the products. We describe in details those different parts in the following sections. However, as ①, ② and ③ share the same technology to define the different involved models, we start with a description of such tool support.

### 7.2.1 The SALOON Model-based Architecture

The SALOON platform relies on three metamodels: the *Domain Knowledge Metamodel*, the *Mapping Metamodel* and the *CardEx Metamodel*, as described in Chapter 6. These metamodels are used to create instances such as the Cloud Knowledge Model, as well as mapping models and feature models. To define those models and metamodels, we rely on the *Eclipse Modeling Framework* (EMF) [SBPM09], which is one of the most widely accepted metamodeling technologies. Each metamodel is thus described as an *ecore* file while dynamic instances, *e.g.*, cloud feature models, are defined as *XMI* models. The XMI format is used to support model persistence, in particular for SALOON to store the different models in a dedicated repository.

Listing 7.1 illustrates the different tags and elements of the EMF representation of the Cloud Knowledge Model. For instance, *Tomcat* is a *CountableElement*, sub-concept of the *Application Server* concept. This inheritance relationship is described using EMF references, *e.g.*, *inherits="//@concepts.0/@subConcept.1"* line 19, where numbers are indexes. Thus, *@concepts.0* is the first concept of the list, *i.e.*, *TechnicalElement*, and *@subConcept.1* is the second sub-concept of *@concepts.0*, *i.e.*, *Application Server*. Finally, lines 30 to 33 illustrates *QuantifiableElement*s, where a value is given to the *Memory* and *CPU* concepts, 4 and 1 GB respectively.

### 7.2.2 The Cloud Knowledge Model Interface

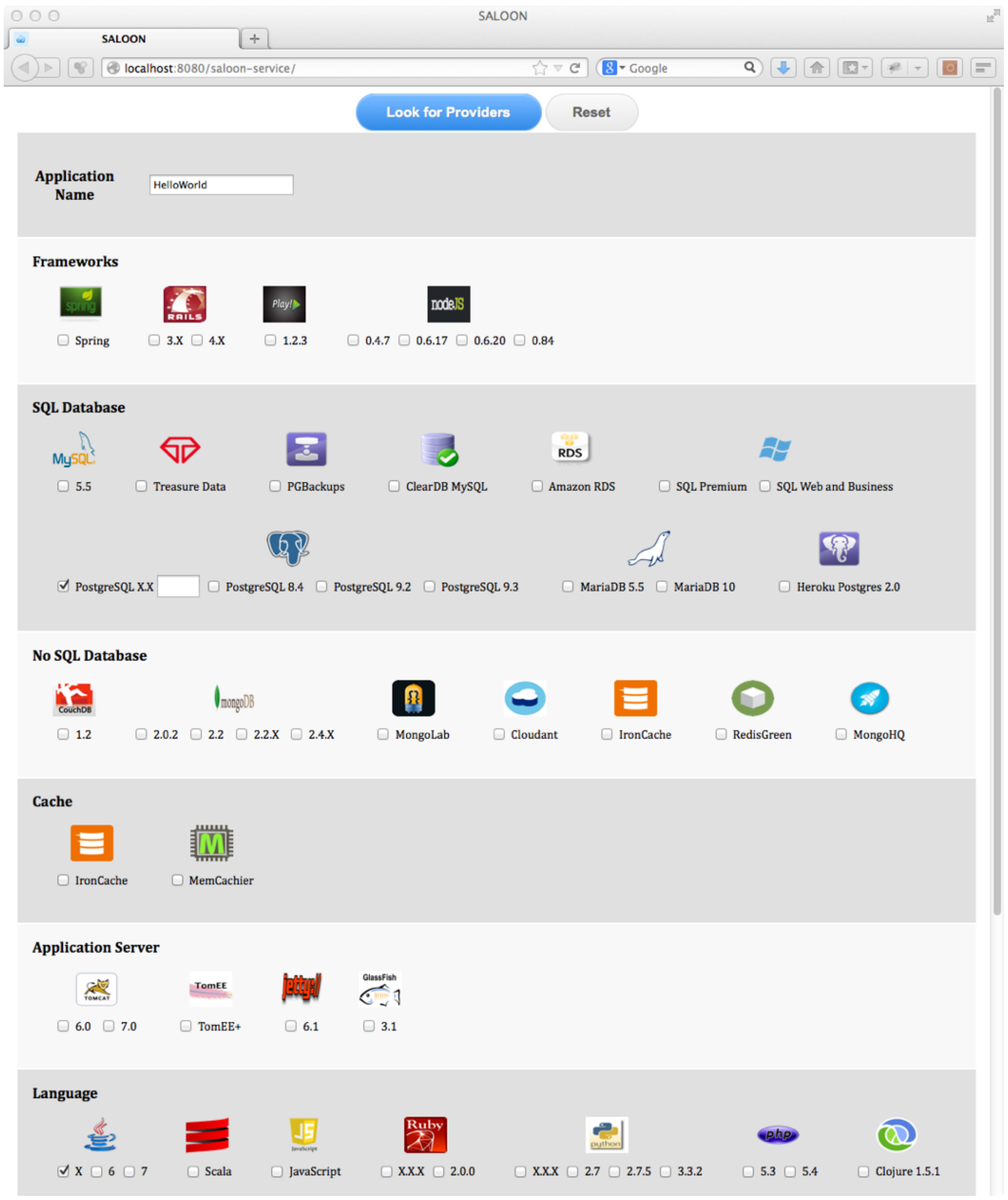
The configuration of cloud feature models is done in an automated way using the Cloud Knowledge Model as entry point of the SALOON platform. To make SALOON easier to handle (the EMF interface to manage models is not user-friendly), we expose this Cloud Knowledge Model as a RESTful service that allows the developer to select concepts regarding the application requirements. The Cloud knowledge model is thus exposed as an HTML

```
<?xml version="1.0" encoding="ASCII"?>
<ckm:DomainKnowledgeModel xmi:version="2.0"
 xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:dkm ="http://fr.saloon.model.dkm"
 xsi:schemaLocation="http://fr.saloon.model.dkm ../metamodel/DomainKnowledgeModel.ecore">
 <concepts name="TechnicalElement">
  <subConcept name="Language">
   <subConcept name="Java" inherits="//@concepts.0/@subConcept.0">
    <subConcept name="Java 6"
     inherits="//@concepts.0/@subConcept.0/@subConcept.0" />
    <subConcept name="Java 7"
     inherits="//@concepts.0/@subConcept.0/@subConcept.0" />
   </subConcept>
   <subConcept name="Ruby" inherits="//@concepts.0/@subConcept.0" />
   <subConcept name="Python" inherits="//@concepts.0/@subConcept.0" />
  </subConcept>
  <subConcept name="Application Server">
   <subConcept xsi:type="ckm:CountableElement" name="Tomcat"
    inherits="//@concepts.0/@subConcept.1">
    <subConcept xsi:type="ckm:CountableElement" name="Tomcat 6.0"
     inherits="//@concepts.0/@subConcept.1/@subConcept.0" />
    <subConcept xsi:type="ckm:CountableElement" name="Tomcat 7.0" />
   </subConcept>
   <subConcept xsi:type="ckm:CountableElement" name="Jetty"
    inherits="//@concepts.0/@subConcept.1" />
  </subConcept>
 </concepts>
 <concepts name="Provisioning">
  <subConcept name="Resource">
   <subConcept xsi:type="ckm:QuantifiableElement" name="Memory"
    inherits="//@concepts.1/@subConcept.0" quantity="4.0" unit ="GB"/>
   <subConcept xsi:type="ckm:QuantifiableElement" name="CPU"
    inherits="//@concepts.1/@subConcept.0" quantity="1.0" unit ="GB"/>
  </subConcept>
 </concepts>
</ckm:DomainKnowledgeModel>
```

Listing 7.1: CloudKnowledgeModel.xmi

client, invoking SALOON Java methods dedicated to mapping through RESTful services using jQuery, a fast and feature-rich JavaScript library [jqu]. More precisely, SALOON relies on the 1.10.2 version of the jQuery library. As the Cloud Knowledge Model evolves *e.g.*, a new feature model is added to the platform, SALOON automatically generates the HTML client regarding the current version of the Cloud Knowledge Model. For instance, for a basic concept, SALOON generates an HTML table cell with its name and a checkbox, for this concept to be selected if required.

As depicted by Figure 7.2, the client proposes the different concepts defined in the Cloud Knowledge Model. They can be selected and, in some cases, values need to be specified. For instance, when *PostgreSQL X.X* and *Tomcat 6.0* are selected, the developer can specify the required size or the number of application server instances respectively.



Application Name: HelloWorld

SQL Database: [✓ PostgreSQL X.X]

Language: [✓ X]

Figure 7.2: The Cloud Knowledge Model interface

### 7.2.3 The Mapping Relationships

Once concepts are selected and values defined in the Cloud Knowledge Model, SALOON provides, relying on mapping relationships, an automated support to select the related features in each feature model. Listing 7.2 depicts the *mapCKMWithFM* method, called for each feature model to address this feature selection.

```
1  void mapCKMWithFM(EObject ckm, EObject mappingModel, EObject featureModel) {
2   TreeIterator<Concept> concepts = (TreeIterator<Concept>) ckm.eAllContents();
3   while (concepts.hasNext()) {
4     Concept concept = concepts.next();
5     if (concept.isSelected()) {
6       TreeIterator<Mapping> mappings = (TreeIterator<Mapping>) mappingModel.eAllContents();
7       while (mappings.hasNext()) {
8         Mapping mapping = mappings.next();
9         if (mapping instanceof ConceptToFeature) {
10          ConceptToFeature conToFeat = (ConceptToFeature) modelElement;
11          if (conToFeat.getFrom().getName().equals(concept.getName())) {
12            Feature featureToSelect = conToFeat.getTo();
13            selectFeature(featureToSelect, featureModel);
14          }
15        }  else if (mapping instanceof ConceptToAttribute) {
16             ConceptToAttribute conceptToAttribute = (ConceptToAttribute) modelElement;
17             QuantifiableElement quant = conceptToAttribute.getFrom();
18             Attribute attr = conceptToAttribute.getTo();
19             mapAttribute(quant, attr);
20        } else if (mapping instanceof MappingRule) {
21             MappingRule rule = (MappingRule) modelElement;
22             boolean conditionRespected = dealWithCondition(rule.getCondition());
23             if (conditionRespected) {
24               ValueAssignment assignment = rule.getAssignment();
25               dealWithAssignment(assignment);
26             }
27        }
28      }
29    }
30  }
31 }
```

Listing 7.2: Features selection regarding selected concepts

For each concept in the Cloud Knowledge Model (lines 2-4), if the concept is selected (line 5), then the algorithm searches for an existing mapping between this concept and a feature of the feature model given as parameter. If a mapping is found, it is either a *Concept-ToFeature*, a *ConceptToAttribute* or a *MappingRule* one (lines 9, 15 and 20). In the first case, the related feature must be selected (line 13). In the second situation, the *mapAttribute* method is called, already discussed in Chapter 6, Listing 6.1 (line 19). Finally, if the condition of the *MappingRule* is satisfied, then the related assignment must be satisfied as well (lines 23-25).

### 7.2.4 Feature Model Consistency Checking

As described in Chapter 5 Section 5.5, feature models are translated into the Aralia textual format of the BR4CP project [BR4]. This translation is done through the SALOON Java class *BR4CPBuilder*. This class provides several methods to translate an EMF feature model into the Aralia format, *e.g.*, *translateFeaturesIntoVariables* described in Listing 7.3.

If the EMF container of the feature is the feature model (line 7), then this feature is the root feature of the feature model and is mandatory. The only value for the domain of its related variable is then 1 (line 8). For all other features, this domain is built regarding their

```
1  void translateFeaturesIntoVariables(FeatureModel fm) {
2    FeatCardinality featureCardinality;
3    int min, max;
4    String varName;
5    for (Feature feature : fm.getFeatures()) {
6      varName = feature.getName();
7      if (feature.eContainer() instanceof FeatureModel) {
8        String varString = "#(1,1,[" + varName + ".1]);";
9        br4cpVariables.add(varString);
10     } else {
11       featureCardinality = feature.getFeatureCardinality();
12       min = featureCardinality.getCardinalityMin();
13       max = featureCardinality.getCardinalityMax();
14       String varString = "#(1,1,[" + varName + "." +  min;
15       for (int i = min+1; i <= max; i++) {
16         varString = varString + "," + varName + "." +  i;
17       }
18       varString = varString + "]);";
19       br4cpVariables.add(varString);
20     }
21   }
22 }
```

Listing 7.3: From EMF to Aralia

cardinality range, *i.e.*, the lower and upper bounds (lines 11-18). These variables are stored into the *br4cpVariables* list (lines 9 and 19) to be written into the related Aralia file. Then, this file is given as input to the Sat4j solver, for the consistency of the feature model to be checked.

### 7.2.5 Feature Model Configuration Analysis

To reason on CardEx feature models, and determine wether a configuration is valid or not, feature models are translated to CSP. In particular, SALOON relies on the Choco solver [JRL08]. We selected Choco because of its maturity and spread usage in research, education and industry. However, the architecture of SALOON is flexible enough to facilitate the support for any Java CSP solver. The version of Choco used in SALOON is Choco 3.1.1. Listing 7.4 illustrates an excerpt of how CardEx feature models are translated into a Choco model.

The first method is used to translate features into variables. With Choco 3, variables require a name, the lower and upper bounds for its domain and the solver used to reason on these variables. Building a variable relies on the *VariableFactory* (VF) provided by Choco (line 5). Lines 10 to 25 describe how a *ValueOperation* is built (see Chapter 4 for its description). First, the involved *ConstrainableElement* is retrieved. We suppose it to be stored somewhere, *e.g.*, features are stored in a map (line 6) and retrieved using the *getCSPVariableFromFeature* method. Then, the operation itself is built, either when no upper bound is defined (line 20) or

```
1  void buildFeatureVariable(FeatureModel fm) {
2    Solver solver = new Solver();
3    for (Feature feature : fm.getFeatures()) {
4      FeatCardinality card = feature.getFeatureCardinality();
5      IntVar variable = VF.bounded(feature.getName(), 0, card.getMax(), solver);
6      mapFeatToVariables.put(variable.getName(), variable);
7    }
8  }
9
10 Constraint buildValueOperation(ValueOperation valueOperation) {
11   IntVar var;
12   ConstrainableElement consElem = valueOperation.getConstrainableElement();
13   if (consElem instance of Attribute) {
14     var = getCSPVariableFromAttribute((Attribute) consElem);
15   } else {
16     var = getCSPVariableFromFeature((Feature) consElem);
17   }
18
19   if (valueOperation.getMax() == -1) {
20     return ICF.arithm(var, ">=", valueOperation.getMin());
21   } else {
22     return LCF.and(ICF.arithm(var, ">=", valueOperation.getMin()),
23         ICF.arithm(var, "<=", valueOperation.getMax()));
24   }
25 }
```

Listing 7.4: From EMF feature models to Choco

when both bounds are defined (lines 22-23). In the first case, it relies on the *IntConstraintFactory* (ICF), while the second case requires both the ICF and *LogicalConstraintFactory* provided by Choco.

In our approach, attributes are translated to variables as well. The type of Choco variable depends on the type of attribute to translate, defined as follows:

- `VF.bounded(attributeName, intValue, intValue, solver)` for an attribute of type int;
- `VF.bounded(attributeName, 0, 1, solver)` for an attribute of type boolean;
- `VF.real(attributeName, floatValue, floatValue, floatPrecision, solver)` for an attribute of type real;
- `VF.enumerated(attributeName, values, solver)` for an attribute of type enum;
- `VF.bounded(attributeName, intMin, intMax, solver)` for an attribute of type bounded;

An attribute is taken into consideration when reasoning on a configuration if its parent feature is also part of the configuration. Thus, the relationship between an attribute and its parent feature is the same as the optional feature relationship, and is described in Listing 7.5.

```
LCF.ifThenElse(ICF.arithm(featureVar, ">", 0),
        ICF.arithm(attributeVar, ">", 0),
        ICF.arithm(attributeVar), "=", 0)));
```

Listing 7.5: Relationship between an attribute and its parent feature with Choco

### 7.2.6 Cost Estimation Engine

As described in Chapter 6 Section 6.5.1, 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. Listing 7.6 illustrates two of those plugins, gathered here for the sake of simplicity, and described using the *JavaScript Object Notation* (JSON) format [jso].

```
"AmazonEC2": {
  "location": "US East",
  "vm": [
     { "size": "M", "os": "Ubuntu", "rate": "hour", "price": "0.084" },
     { "size": "L", "os": "Ubuntu", "rate": "hour", "price": "0.17" },
     { "size": "XL", "os": "Ubuntu", "rate": "hour", "price": "0.33" },
  ]
}
"ElasticHosts": {
  "resources": [
     {
        "memory": [
           { "size": "512", "rate": "hour", "price": "0.019" },
           { "size": "512", "rate": "month", "price": "6.99" },
           { "size": "512", "rate": "year", "price": "69.9" },
        ]
     }
  ]
}
```

Listing 7.6: Pricing models using JSON format

For instance, regarding the *Amazon EC2* environment, the price depends on the size of the selected virtual machine *vm*, and is proportional with respect to this size. Contrarily to *Amazon EC2*, *Elastic Hosts* proposes a discount according to the duration of the *resources* allocation. Thus, for the same duration, it is cheaper to select the *year* rate than pay twelve times the *month* rate, since a discount equivalent to two months is proposed.

### 7.2.7 Product Derivation

SALOON supports the developer when configuring a cloud environment by deriving the related cloud configuration scripts and files. This derivation process relies on the Acceleo environment [acc]. Acceleo is an implementation of the MetaObject Facility (MOF) [MOF] model-to-text standards. It provides a flexible and simple environment to design and develop a variety of code generators, using simple and standard templates. Furthermore, it is compatible with the Eclipse environment, and thus with our EMF-based models. Figure 7.3 depicts some Acceleo mechanisms.

```
[query public getSelected(fm : FeatureModel) : Sequence(Feature) =
eAllContents()->select(oclIsTypeOf(Feature))->select(featureCardinality.configValue >= 1) /]

[template public procfile(language: String, mainProcess: String)]
[file ('Procfile', false, 'UTF-8')]
    [if (language.equalsIgnoreCase('java'))]
        web: java -cp target/classes:target/dependency/* [mainProcess/]
    [elseif (language.equalsIgnoreCase('python'))]
        web: gunicorn [mainProcess.toLower()/]:app
    [elseif (language.equalsIgnoreCase('scala'))]
        web: target/start [mainProcess/]
    [elseif (language.equalsIgnoreCase('ruby'))]
        web: bundle exec ruby [mainProcess.toLower()/].rb -p $PORT
    [/if]
[/file]
[/template]
```

Figure 7.3: The Acceleo generation engine

One of these mechanisms relies on OCL to execute requests on a model element. For instance, a feature model is given as parameter to the request, used to retrieve selected features in this feature model. For all elements of the model (*eAllContents()*), only elements of type *Feature* are selected. Then, among these features, those present in the configuration are selected. In addition, templates are used to generate text. If the *file* tag is defined, then the generated text will be contained in this file, *e.g.*, *Procfile*. In this example, depending on the language, different instructions will be generated.

## 7.3 Experiments and Evaluation

This section reports on some experiments we conducted to evaluate the SALOON platform. The aim of these experiments is to evaluate the usefulness of SALOON for selecting and configuring a well-suited cloud environment. Moreover, adding new constraints support for CardEx feature models brings an additional cost when running the platform. This section thus investigates whether the extra capabilities in terms of configuration introduced in the platform penalize its performance or not. This evaluation aims at investigating the three following criteria:

**C1:** ***Soundness***. Is the platform, and our approach for CardEx feature models in particular, well-suited to support cloud environment modeling and configuration?

**C2:** ***Scalability***. Is SALOON still performing when handling feature models with a substantial amount of features and constraints and when selecting among tens of cloud environments? What about consistency checking?

**C3:** ***Practicality*.** Does running SALOON when configuring cloud environments to deploy applications improve reliability and efficiency?

We describe in details in the following sections the experiments we conducted to evaluate our approach with respect to these criteria.

### 7.3.1 Soundness

The aim of this evaluation is to empirically assess the soundness of our approach, by using the CardEx metamodel 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 feature model which conforms to the CardEx metamodel. We define this set of 10 cloud feature models in the following as the $Cloud_{corpus}$. All feature models from the $Cloud_{corpus}$ are available in Appendix A. The selection of these 10 cloud environments 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.
- *Data 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 7.1 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 feature model, the number of **`Attributes`**, and the number of **`Constraints`**. For features and constraints, it gives details on the amount of features and constraints with cardinalities and attributes, $F_{card}$, $F_{attr}$ and $C_{card}$, $C_{attr}$ respectively. To assess the soundness of our approach, we determine how often do cardinalities and attributes occur in cloud environments feature models, 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 *data 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 feature model with about 3 features with cardinality, 5

| | | 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 7.1: Modeled cloud environments

with attributes and about 19 `CardExConstraints`. There 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.

### 7.3.2 Scalability

The aim of the following experiments is to evaluate the performances of the SALOON platform with respect to its scalability. This evaluation is twofold. First, we evaluate the performances of SALOON in the whole, *i.e.*, from a set of requirements to a configuration analysis by the CSP solver. Then, we evaluate how our approach for detecting cardinality inconsistencies presented in Chapter 5 performs with large feature models. All these experiments were performed on a MacBook Pro with a 2,6 GHz Intel Core i7 processor and 8 GB of DDR3 RAM.

**From Requirements to Configuration Analysis**

This evaluation is divided into three parts. 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 feature models 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 threats to scalability of SALOON if taking too much computation time. Third, 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 CardEx feature model 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 exist 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, the algorithm randomly assigns a given amount of child features, if possible. In the previous example, if feature $f4$ 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, *e.g.*, a `ValueOperation`, 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 feature models, that is, feature models with at least one valid configuration. Indeed, our algorithm sometimes generates void feature models by unfortunate generation of constraints.

For the first experiment, we generate random feature models with 10, 50, 100, 500, 1000, 5000 and 10000 features and we perform 50 random generations for each feature amount. 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 feature model with feature cardinality set to [0..1] or [1..1]. Figure 7.4 depicts the time taken by SALOON to find a valid configuration, computed as the average time for each feature amount, and displayed using logarithmic scale.

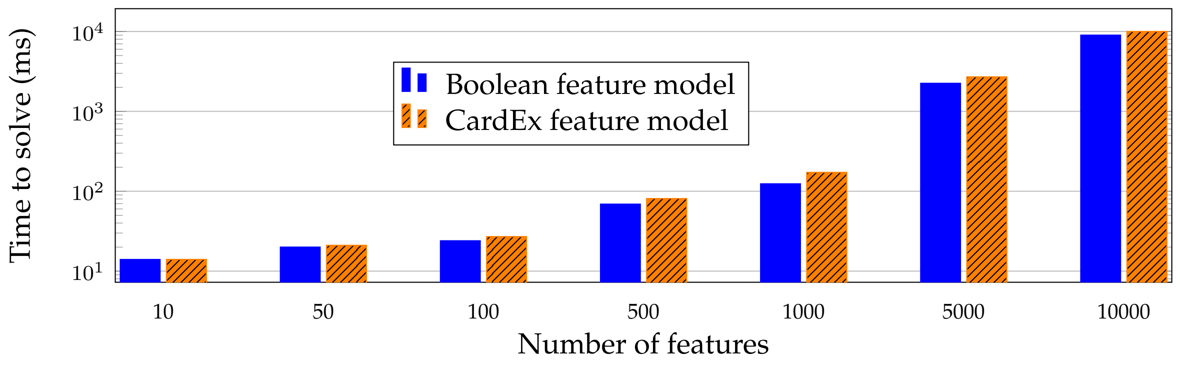


Figure 7.4: Time to find a valid configuration

For the second experiment, we measure the time taken by SALOON to translate XMI feature models into CSP constraints. We thus check if this translation is not a threat to SALOON scalability, in particular regarding large feature models ($nbFeat > 500$). 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 Figure 7.5, using logarithmic scale.

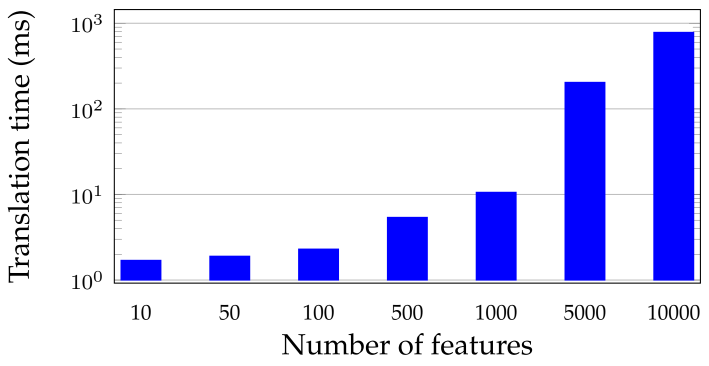


Figure 7.5: Time to translate from XMI to CSP

For the third experiment, we measure the time taken by SALOON for selecting features in the different feature models regarding a set of requirements 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 feature model set size, as described by Table 7.2. In this experiment, generated feature models size is lower than 100 features, to be as close as possible to cloud environment feature models described in the $Cloud_{corpus}$.

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

Table 7.2: Feature selection and configuration analysis time

**Result Analysis.** Regarding the first experiment, the aim was to compute the verification time overhead for the same randomly generated feature model, either with cardinality, attributes and constraints over them or considerer as a boolean feature model. As illustrated by Figure 7.4, the support for our CardEx 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 finding a valid configuration is done within a few milliseconds for feature models with less than $5000$ features, which are the majority of them. Moreover, 10 seconds remains fairly reasonable for feature models with $10000$ features.

Second, as shown by Figure 7.5, 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 [spl]. Then, one of the biggest existing feature model, which is the Linux feature model, has over 5000 features [SLB+11]. 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 the whether the configuration is valid or not within a few seconds, even for 200 feature models, as illustrated by Table 7.2. This time is negligible and is not a threat to the scalability of SALOON, compared to the time taken to configure those feature models 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 set of requirements.

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 feature models, with a substantial number of features and constraints, either boolean or CardEx ones.

### Cardinality Inconsistencies Detection

The intent of this evaluation is to assess the scalability of our approach when evolving large cardinality-based feature models and checking their consistency. As a reminder, note that there is no attribute in these feature models, and they only hold as constraints boolean ones or CardEx constraints with a `ValueOperation` on a feature cardinality, further referred to as cardinality-based constraints. As we did not find large cardinality-based feature models in the literature, we thus implemented an algorithm to randomly generate such feature models. This algorithm, given *nbFeatures* and *cardMax*, generates a random feature model with *nbFeatures* features, whose cardinality is randomly assigned a range $\mathcal{R}$, with $\mathcal{R} \subseteq [0..cardMax]$. This algorithm creates *nbFeatures* features, then builds the tree hierarchy as described in the previous scalability experiment. Then, the algorithm generates cross-tree constraints, where two features are selected randomly, with one constraint for every 10 features as proposed by Thüm *et al.* [TBK09]. Either boolean or cardinality-based constraints are generated (50% probability each). For the latter, the constraint range or required instances amount is built to fit within the feature cardinality such as the generated feature model is consistent. Here again, we only consider non-void feature models, that is, feature models with at least one valid configuration.

To measure the performance of our approach while detecting cardinality inconsistencies, we also generate feature model edits. In particular, regarding the edits described in Chapter 5, Section 5.3, we implemented the following operations: (1) move a feature, (2) add a cross-tree constraint, (3) update a feature cardinality and (4) update a cross-tree constraint. Operation (1) picks at random an existing feature, and create a new feature assigned randomly as a child or a parent of this feature. In (2), the algorithm generates a random constraint, while a feature cardinality is randomly changed by operation (3). Finally, operation (4) makes a boolean constraint a cardinality-based constraint or update the range of an existing cardinality-based constraint. For each generated feature model, we generate one or several edits leading, on purpose, to inconsistency. For each edit, we thus know what kind of inconsistency will result. For instance, we update a cross-tree constraint and check the consistency of the targeted feature, which should be local range inconsistent. We then evaluate how our approach performs when handling cardinalities, in particular regarding the potential combinatorial explosion. Indeed, the number of combinations that the solver has to examine grows significantly when using cardinalities. To reduce this issue and improve our inconsistency detection mechanism, we adapt the algorithm described in Chapter 5, Definition 2. In our experiments, to check the global range consistency of a feature $f$, the translation algorithm does not update the feature cardinality range considering all $f$

parent features, but only one of them. More precisely, it is updated using its direct parent feature if its cardinality upper bound is greater than one, or the algorithm search for another parent feature whose cardinality upper bound is greater than one, until the root feature. If no feature matches this criteria, then the local range consistency of *f* is checked. This improvement is used only for these experiments, as the approach described in Chapter 5 handles any cardinality-based feature models. We argue that this adaptation is fair since, in all cardinality-based feature models we found in the literature, including ours, we never found any feature model whose cardinality upper bound is greater than one for more than two features hierarchically linked, *e.g.*, A:[0..3], B:[1..5] and C:[0..4] with A parent of B and B parent of C.

*Experiment 1.* For the first experiment, we measure the computation time required to find an inconsistency with *(i)* one random edit leading to a local or global range inconsistency, *(ii)* a fixed value for *cardMax* and *(iii)* an increasing number of features *nbFeatures*, thus varying the size of the feature model. We set the value of *cardMax* to 10. We argue that this value is a fair value to evaluate our approach. First, as explained above, we never found any feature model whose cardinality upper bound is greater than one for more than two features hierarchically linked. Second, in the case we found a feature with a high *cardMax* number, its parent feature was either an optional or mandatory feature with *cardMax* set to 1. Our algorithm does not take into consideration these findings and generates feature models with a random cardinality for each feature, *i.e.*, feature models whose combinatory grows significantly with feature model size.

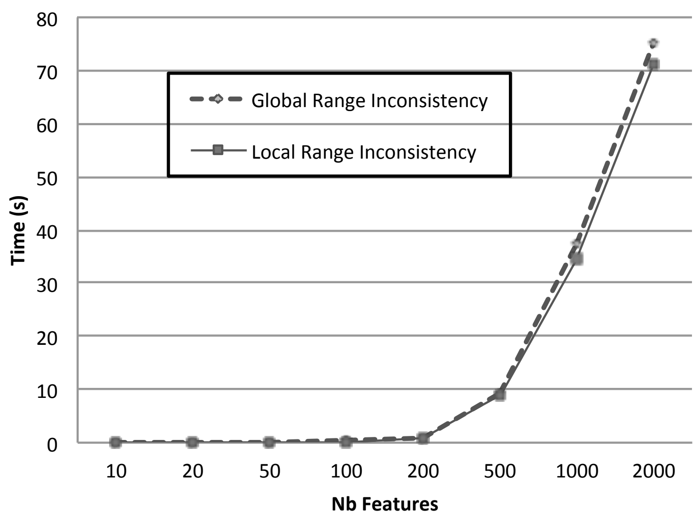


Figure 7.6: Detecting an inconsistency in feature models with maximum upper bound cardinality set randomly from 1 to 10.

In this experiment, we vary the size of the feature model from 10 to 2000 features which is, once again, more significant than the cardinality-based feature models we found, including ours. Generating random feature models leads to important differences regarding computation time to detect inconsistency for the same feature model size. This is due to the randomly generated tree hierarchy and constraints as well as the random edits performed. To deal with this issue, we performed 200 generation runs for each feature amount and computed the average time. As shown in Figure 7.6, our detection algorithms perform the same way to detect both local and global range inconsistencies, even if detecting a global range inconsistency generates a small overhead compared to detecting a local one, *e.g.*, 2,5 seconds for 1000 features. This overhead results from the additional variables and constraints the solver has to handle, due to the way we translate the feature model into the configuration problem textual format as described in Chapter 5, Section 5.5.

Thus, for the same generated feature model, there are more variables and constraints to take into consideration when checking the global range consistency than when checking the local one. Moreover, even though the detection time seems important regarding the feature model size (in particular compared to boolean feature models, *e.g.*, about 1 second to find an edit for a feature model with 2000 features [TBK09]), it is explained by the amount of variables the solver has to handle in such cases. Given a feature $f$ with a cardinality range [0..8], then 9 variables are required to reason on this feature, *i.e.*, one per cardinality value. For instance, for cardinality-based feature models with 2000 features and 200 constraints, the solver has to reason on an average model of 15670 variables, 11910 constraints and 16120 variables, 12240 constraints to detect local and global range inconsistencies respectively. Overall, for cardinality-based feature models with less than 200 features, the computation time is less than 1 second. This time then increases significantly for larger feature models, with an average time of 9 seconds for 500 features, 36 seconds for 1000 features and up to 73 seconds for 2000 features.

*Experiment 2.* For the second experiment, we measure the computation time required to find an inconsistency with *(i)* one random edit leading to a local or global range inconsistency, *(ii)* an increasing value for *cardMax* and *(iii)* a fixed number of features, with *nbFeatures* = 200. We consider this value is fair for this experiment, as this is larger than the biggest cardinality-based feature model we found. To generate feature models with features whose cardinality is the one expected, we modified our algorithm such as a feature is given a cardinality [m..n], with m ∈ {0,1} and n = *cardMax*. The aim of this experiment is to evaluate how our approach performs in presence of several features with a high cardinality. We thus vary *cardMax* from 10 to 45 and compare the computation time required to detect either a local or global range inconsistency. We performed 500 generation runs for each *cardMax* value and computed the average time. Figure 7.7 depicts the results of this experiment.

As expected, the time required to detect an inconsistency increases as the value of *cardMax* does, from less than one second for *cardMax* = 10 to more than 18 seconds for *cardMax* = 45 (18 and 21,8 seconds in average to detect a local or a global range inconsistency respectively). As in the previous experiment, detecting a global range inconsistency generates an overhead compared to detect a local one, *e.g.*, one second for *cardMax* = 25 or two seconds for *cardMax* = 35.

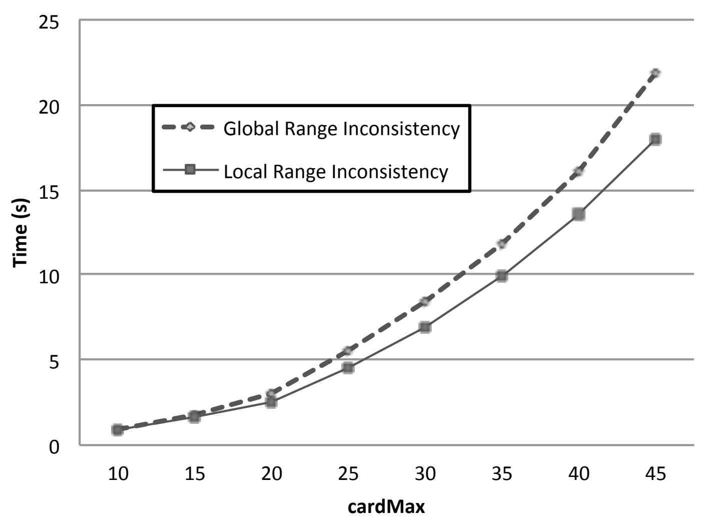


Figure 7.7: Detecting inconsistency while varying the maximum upper bound cardinality (FM size = 200).

*Experiment 3.* For the third experiment, we measure the computation time required to find inconsistencies with *(i) nbEdits* random edits leading to local or global range inconsistencies, with *nbEdit* $\in$ [1..*nbFeatures*/20], *(ii)* a random value for *cardMax* with *cardMax* $\in$ [1..30] and *(iii)* an increasing number of features *nbFeatures*, thus varying the size of the feature model. Our generation algorithm is thus slightly evolved to generate random feature models whose one feature out of ten has its cardinality upper bound set to *cardMax*. Other features are either optional or mandatory features one usually meet in boolean feature models. The aim of this experiment is to evaluate how our approach performs when checking cardinality-based feature models whose structure is as close as possible as the one we met in the literature. Moreover, this is not limited to one edit, but several random ones, as it occurs in classic feature oriented development, *e.g.*, staged configuration [CHE05b]. We thus generate random edits and compute the time required to find an inconsistency, if any. Figure 7.8 depicts the results of these experiments, where the time is an average computation time over 200 generation runs.

The resulting trend matches the one depicted in Figure 7.6, but this approach is more than twice more efficient to detect inconsistencies, *e.g.*, 15 seconds for 1000 features in this experiment in comparison to 36 seconds for the same features amount in the first experiment. We would expect such a result, as there are less features with a high cardinality upper bound value. The time required to detect an inconsistency remains however quite significant. This is due to the value of *cardMax* (200 features with a cardinality upper bound set up to 30 for a feature model with 2000 features) and to the random number of edits, *e.g.*, up to 50 for a feature model with 1000 features.

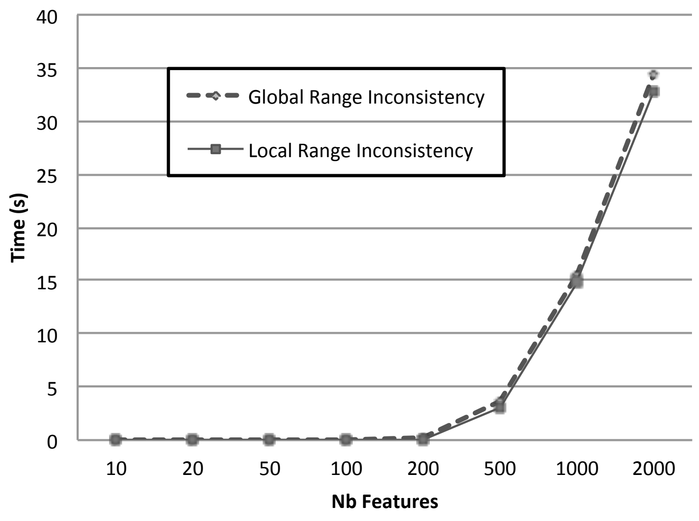


Figure 7.8: Detecting an inconsistency in feature models whose one feature out of ten has its cardinality upper bound greater than one.

Overall, our approach is well-suited to detect inconsistencies in cardinality-based feature models whose size is lower than 500 features (about 9 seconds in the worst case, 3 seconds otherwise). For larger feature models, the number of features as well as the cardinality upper bound value as an important impact on the computation time. However, although we did not define a threshold for this experiment, we can fairly argue that these results are acceptable, as we did not find in the literature such large cardinality-based feature models.

### 7.3.3 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 developers. 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 7.3 describes the results of this first experiment.

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 developer

| 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 7.3: Configuring Heroku and deploying the application

and two researchers. They were asked quite the same task: *Use* SALOON *to configure an Heroku environment, upload a web application and add a PostgreSQL support.* Table 7.4 describes the results of this second experiment.

| 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 7.4: Configuring Heroku and deploying the application using SALOON

**Result Analysis.** The task asked 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 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 28 minutes with a manual process (for the latter, we consider the average value for the five successful deployments), leading to a time reduction of 72%.

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.

## 7.4 Discussion

In this section, we present and discuss several concerns of our evaluation that may form threats to its validity.

### On the Scalability Experiments

For those experiments, we tested our approach on randomly generated feature models. Even though it seems fair, we did implement the generation algorithm and thus expect the feature models to conform a certain structure, *e.g.*, with a given number of constraints. Moreover, those feature models conform to the CardEx metamodel we also defined (more precisely, to the CardEx metamodel without meta-classes related to attributes for experiments about detecting cardinality inconsistencies). This might not be representative of the usage of such feature models, in particular in the industrial domain. Moreover, the algorithm does not generate completely random feature models, since they are consistent after being generated, and since we only consider non-void feature models. The generation process is thus guided to fit once again a certain structure. Regarding our evaluation about detecting cardinality inconsistencies, one could also argue that our evaluation is not complete as it does not take into account global range consistency for the whole list of parents, but only one of them. However, such a result would only serve as comparison, as we did not find any cardinality-based feature model matching this pattern.

**On the Selected Cloud Environments**

The choice of the $Cloud_{corpus}$ threatens the evaluation validity. We first selected a substantial number of cloud providers from many sources (our knowledge, case studies in cloud related conference papers, web comparator, etc.). However, the intrinsic nature of cloud computing, in particular the variability of cloud environments, was a limiting factor. Cloud environments provide a shared pool of highly configurable computing resources, in several configuration levels. It was thus impossible for us to fully reify the cloud environments and we had to limit our feature modeling to features which are explicitly released by cloud providers, since implicit features and constraints finding and modeling are far more complex. Feature models used in our experiments were thus not exhaustive. Moreover, the number of cloud environments studied in the $Cloud_{corpus}$ is relatively low regarding the real number of existing providers. However, as explained in Section 7.3.1, we try to be as fair as possible when selecting our case studies for them to be representative of the domain. Finally, due to the evolutive nature of cloud computing, *e.g.*, cloud providers that appear/disappear or existing environments evolving, $Cloud_{corpus}$ case studies might not be valid over the long term.

## 7.5 Summary

In this chapter, we have presented a validation for the SALOON framework. We presented the implementation details of various elements of the platform, and then evaluated different concerns of our approach. Overall, as our empirical evaluation shows, we observe that SALOON is well-suited to handle the configuration of cloud environments. Relying on CardEx feature models 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. Moreover, our approach is well-suited to detect inconsistencies in cardinality-based feature models whose size is lower than 500 features (about 9 seconds in the worst case, 3 seconds otherwise). For larger feature models, the number of features as well as the cardinality upper bound value have an important impact on the computation time. 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 set of requirements remains fairly low, thus not hindering its usability. Finally, automating the feature model configuration files generation improve the reliability of the deployment process compared to manual and error-prone process.

This chapter concludes the third part of this document, which was dedicated to the validation of our approach. In the following chapter, we summarize the main contributions of this dissertation, present the conclusions of the research work, and define a set of perspectives for future work.

# Part V

# Conclusion

# Chapter 8

# Conclusion and Perspectives

**Contents**

In this chapter, we summarize our thesis dissertation by discussing the challenges and goals addressed, and we outline our contributions. Then, we discuss our short-term and long-term perspectives related to the work presented in this dissertation.

## 8.1 Summary of the Dissertation

Modern software and computer configurations are increasingly distributed, and use a variety of software services, in particular with the adoption of the cloud computing. Cloud computing represents a promising paradigm to achieve the utility computing vision, where developers access services on-demand, based on their requirements, without regard to where the services are hosted or how they are delivered. Those resources include computational and memory blocks, as well as virtual machines, application servers, databases, or any required library. Developers thus take advantage of the flexibility of this provisioning model and rely on cloud environments to host their applications. In these new usages, dealing with the variety of available cloud environments and services they provide requires a significant knowledge. Developers thus need to be supported in their cloud selection and configuration choices.

However, offering an approach for helping developers selecting and configuring cloud environments is not straightforward. Numerous clouds environment are available, each one providing several functionalities with different non-functional properties. Moreover, there is no consensus on how these environments are documented, nor how their services can be configured. In addition, these environments evolve over time, making their knowledge, and therefore their configuration, difficult to manage without a dedicated support. In this thesis, we addressed these challenges and provided cloud models relying on a well-known formalism, together with a set of approaches and tools to help the developer face these issues.

The solution we propose is based on well accepted and defined technologies and standards. In particular, our platform, called SALOON, relies on the principles of software product line engineering. We leverage these principles to provide an automated support for cloud selection and configuration. In SALOON, cloud environment commonalities and variabilities are described using feature models. Feature models enable the definition of both coarse-grained and fine-grained cloud functionalities, and provide a reasoning support dedicated to configuration analysis. As one feature model describes one cloud environment, it would be tedious and error-prone to select features manually for each feature model. We thus propose in SALOON a model reifying all cloud elements present in each feature model, together with mapping relationships from these elements to the related features. This model, called the Cloud Knowledge Model, is also used by the developer to specify its functional and non-functional requirements. To cope with the evolution of cloud environments, and therefore of their related feature models, we provide means to make them evolve properly, by checking their consistency and explaining, when necessary, why the evolution edit introduced an inconsistency. Finally, SALOON exploits the software product line derivation process to generate configuration files and scripts related to a valid configuration established in a given feature model.

Thus, from the developer's perspective, SALOON acts as a black-box: the developer defines the requirements using the Cloud Knowledge Model as entry point of the platform, and retrieves the configuration files according to these requirements. In the next section, we present an overview of the contributions associated with our SALOON platform.

## 8.2 Contributions

The contributions of this thesis, made as part of our work around the SALOON platform, are summarized as follows:

### Support for CardEx Feature Models

In Chapter 4, we introduced our CardEx metamodel. This metamodel provides an abstract syntax to define feature models with attributes and cardinalities, together with constraints over them. These extensions are required to describe the variability of cloud environments, and existing feature modeling approaches only provide a partial support for such extensions. In particular, we introduce new constraint expressions regarding cardinalities, and describe their semantics. This metamodel reifies relevant concepts and elements from existing feature modeling approaches so that existing feature models, even without extension, are natively managed by SALOON. The metamodel is simple and easy to use, and through our experimentation we have demonstrated that it is expressive enough to model different kinds of relationships required when describing cloud environments. In addition, we describe translation rules for mapping a CardEx feature model into a Constraint Satisfaction Problem (CSP). Thus, reasoning on CardEx feature models *e.g.*, search for a valid configuration, can be automated relying on any existing CSP solver. Compare to Boolean feature models, the addition of our CardEx expressions generates a small increase in the required time to perform such a reasoning, which remains negligible.

**An evolution support**

As cloud environments evolve over time, their feature models have to evolve as well to be kept up-to-date. However, extending feature models with cardinalities make their evolution complex, as inconsistencies may arise regarding those cardinalities. In Chapter 5, we thus describe atomic edits that can be performed while evolving those feature models, and distinguish two kinds of possible cardinality inconsistencies. We then show how such inconsistencies can be related to the recently defined notion of global inverse consistency in the area of CSP. The benefit of relating our approach to the one on global inverse consistency is to allow us reusing existing CSP tools to detect inconsistencies in cardinality-based feature models. We thus provide an automated support for both detecting and explaining *where*, *how* and *why* did an inconsistency arise when evolving the feature model. Even though the time required to detect an inconsistency is quite significant, SALOON is the first approach addressing this support. With such a support, SALOON helps developers evolving cardinality-based feature models and thus, makes cloud feature models edition more reliable.

**An automated support**

SALOON encapsulates the CardEx metamodel and the evolution support, and provide in addition several automated supports to help developers selecting and configuring cloud environments. Developers specify their requirements using the Cloud Knowledge Model, to finally retrieve configuration files corresponding to these requirements and the selected cloud. In between, all processes are automated, as explained in Chapter 6. First, from the set of requirements and thanks to mapping relationships, features are selected in the different cloud feature models, feature cardinalities are computed and defined, and attributes are given a certain value, avoiding a tedious an error-prone manual definition. Then, SALOON determines whether the selected features represent a valid configuration or not in those feature models, relying on a CSP solver. When a valid configuration is found, SALOON estimates the cost of such a configuration using the cost estimation engine, which relies on defined cost models. Finally, by using configuration files and executable commands as assets of cloud feature models, SALOON automates the generation of fulfilled configuration files, together with an executable script specific to the targeted cloud. We showed that using such an automated support lead to a time reduction of 73% and improve the configuration success rate of 50%, even for a simple Java application deployment.

## 8.3 Perspectives

Although the work presented in this dissertation covers the needs of selecting and configuring cloud environments, there is still some work that could be done to improve our research. In this section, we thus discuss some short-term and long-term perspectives that should be considered in the continuation of this work.

### 8.3.1 Short-term Perspectives

#### Improve SALOON Practicality

As explained in Chapter 7, feature models are defined within SALOON using the Eclipse Modeling Framework (EMF). Even though EMF is a powerful dedicated modeling tool, it provides limited support for defining feature models, either graphically or textually. Feature models are thus defined in a tree view way, or using an XML-based textual format. The practicality of SALOON regarding feature model description would benefit from a dedicated Graphical User Interface (GUI) or a DSL, since textual variability modeling approaches seem to be a well-suited alternative, as explained in Section 3.4, Chapter 3. Moreover, the GUI could be used when evolving cardinality-based feature models to improve the inconsistency understanding, *e.g.*, by propagating the explanations given by the solver to the feature model graphical editor and highlighting model elements leading to the inconsistency.

#### Improve Interoperability

Currently, the definition of feature models is done within SALOON using EMF. An interesting improvement would be to provide means to import existing feature models defined with state-of-the-art approaches, and thus support different formats. Such feature models would thus benefit from the CardEx expressions we support *e.g.*, by improving their semantics or enabling the search for a valid configuration regarding a given property defined using attributes. A translation engine would be required, to parse the existing feature model and generate one conforming to the CardEx metamodel. A close-related idea is to generate configurations that conform to the *Open Virtualization Format* (OVF), an open-source standard for packaging and distributing software and applications for virtual machines. Numerous cloud environments use or are about to use OVF, thus improving the portability of an application from a cloud to another one.

#### Deal with Cardinalities at Solution Space Level

We propose in this dissertation an approach for modeling and reasoning on cardinality-based feature models. Those processes are done at problem space level, but issues may arise at solution space level, *i.e.*, when managing concrete software artifacts (see Section 2.3, Chapter 2). For example, when deriving several instances of the same feature, one may need to explicitly refer to a precise instance, *e.g.*, one which has a particular property and which differs from the other. A possible solution is to represent the children of a feature by a set of arrays of instances, as proposed by Cordy *et al.* [CSHL13]. The current implementation of SALOON does not provide such a capability, as it is not required. Indeed, in our approach dedicated to cloud environments, cardinalities are used to get the final value required in the configuration *e.g.*, 2 `Dyno`s, but not for referring to a specific element. However, it could be required if SALOON was used for another domain of study.

### 8.3.2 Long-term Perspectives

#### Consider Multi-Cloud Configurations

In our approach, we address the configuration of cloud environments regarding a given set of requirements. 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 those requirements. However, a multi-cloud configuration may be better suited regarding those requirements [PHM+12]. We believe that our approach could also be relevant to address a multi-cloud configuration *i.e.*, a subset of the application deployed on one cloud, and another one on another cloud. A multi-cloud configuration is useful in several cases *e.g.*, data stored on a private cloud while computing processes run on a public one. The main issue to cope with when considering such configurations while relying on feature models is to know how and where these feature models can be sliced, to get partial configurations [ACLF11b]. However, such a challenge has only be faced for Boolean feature models until now.

#### Towards a DSPL Approach

Another domain of extension of the SALOON platform is to leverage *Dynamic Software Product Lines* (DSPL), thus adding support for product adaptation at runtime [HHPS08]. Indeed, changes can occur in the application context requiring the cloud 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 software product line has to be well-suited to handle this adaptation, *e.g.*, defining a feedback control loop [KC03] that monitors the metrics, detects adaptation situations and execute the required actions in order to search for a better suited configuration. Traceability links would also be useful to support this kind of reconfiguration, as we need to store a trace of the history of previous configurations as well as to store the set of requirements. Such links could also be used to update the cloud feature model regarding the running environment *e.g.*, update an attribute related to a non-functional property, such as the cost of a feature.

**SaaS: SALOON-as-a-Service**

As described in Chapter 6, SALOON distinguishes between two roles: domain architects and developers. Domain architects define cloud feature models, while developers use SALOON to configure a cloud environment regarding a set of requirements. In this dissertation, we played the role of the architects by defining feature models ourselves. By providing SALOON as a service with two entry points, we allow architects in real life to come and define their own cloud environments. Thus, it can be targeted by SALOON and bring more customers to the cloud provider. Another possibility is to reverse engineering feature models from existing cloud environments, even if such an approach still needs some improvement [AAHC14]. On the other hand, the second entry point is used by developers, who benefit from this new delivery model since the platform now proposes more cloud environments. Therefore, there is a greater chance to find a suitable configuration. It can be seen as a virtuous circle, since if there is a more cloud environment described and therefore a greater chance to find a proper configuration, then there will be more developers interested in using SALOON. If there are more developers using SALOON, then more cloud providers will be interested in getting their cloud environment described within SALOON.

# Appendices

# Appendix A

# SALOON models

We report here the models used in the current version of the SALOON platform, *i.e.*, the Cloud Knowledge Model and the cloud environment feature models.

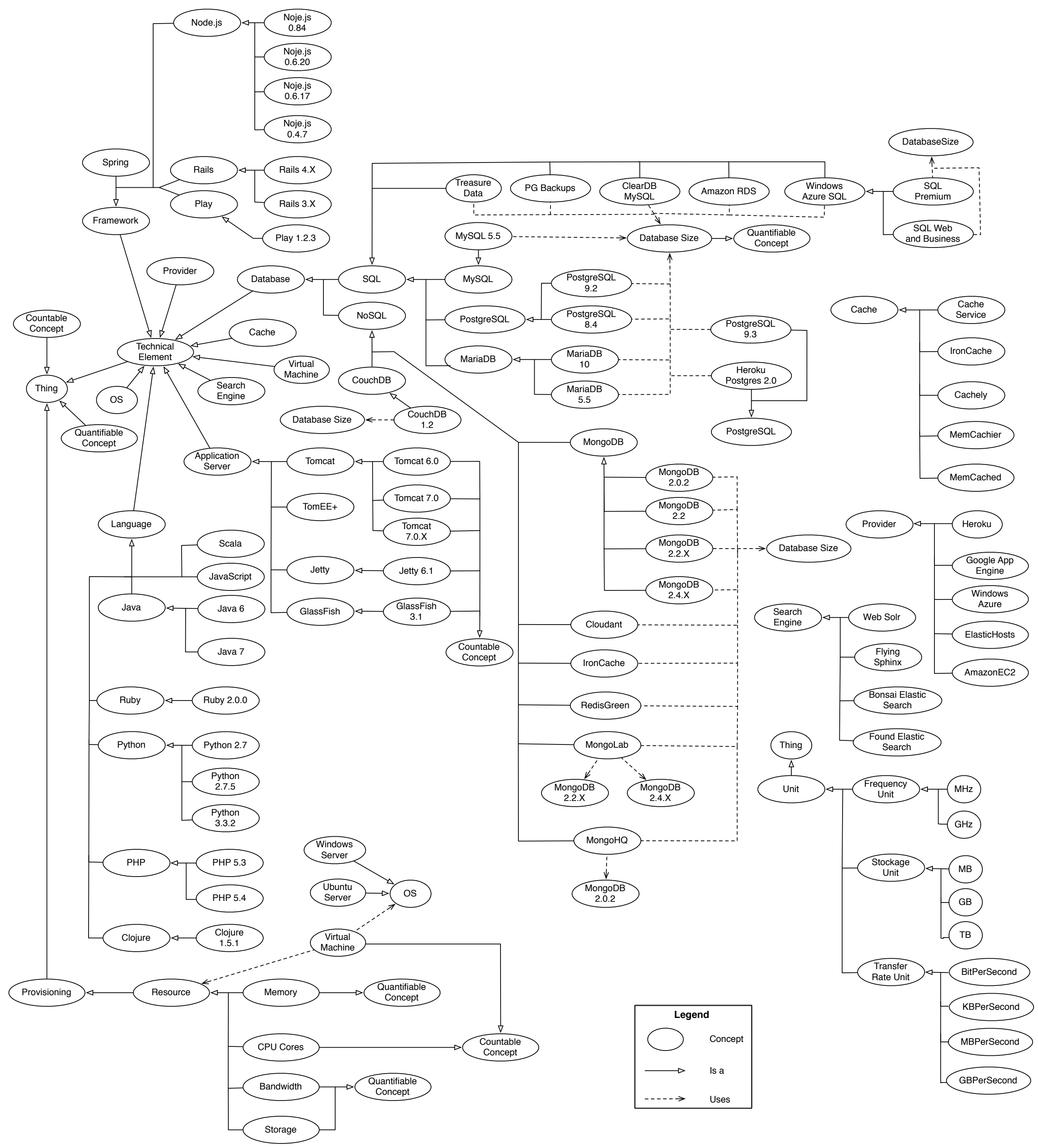
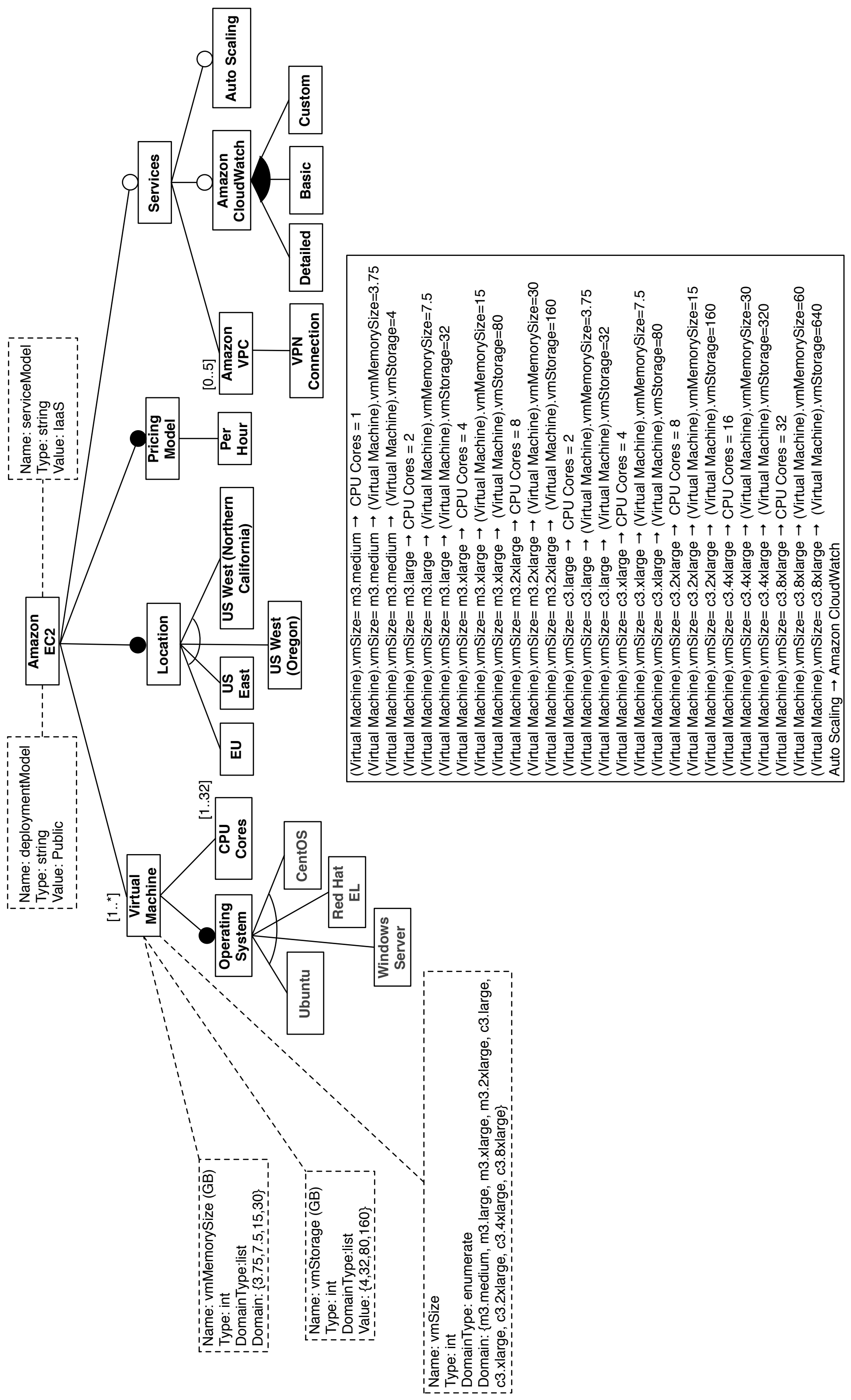
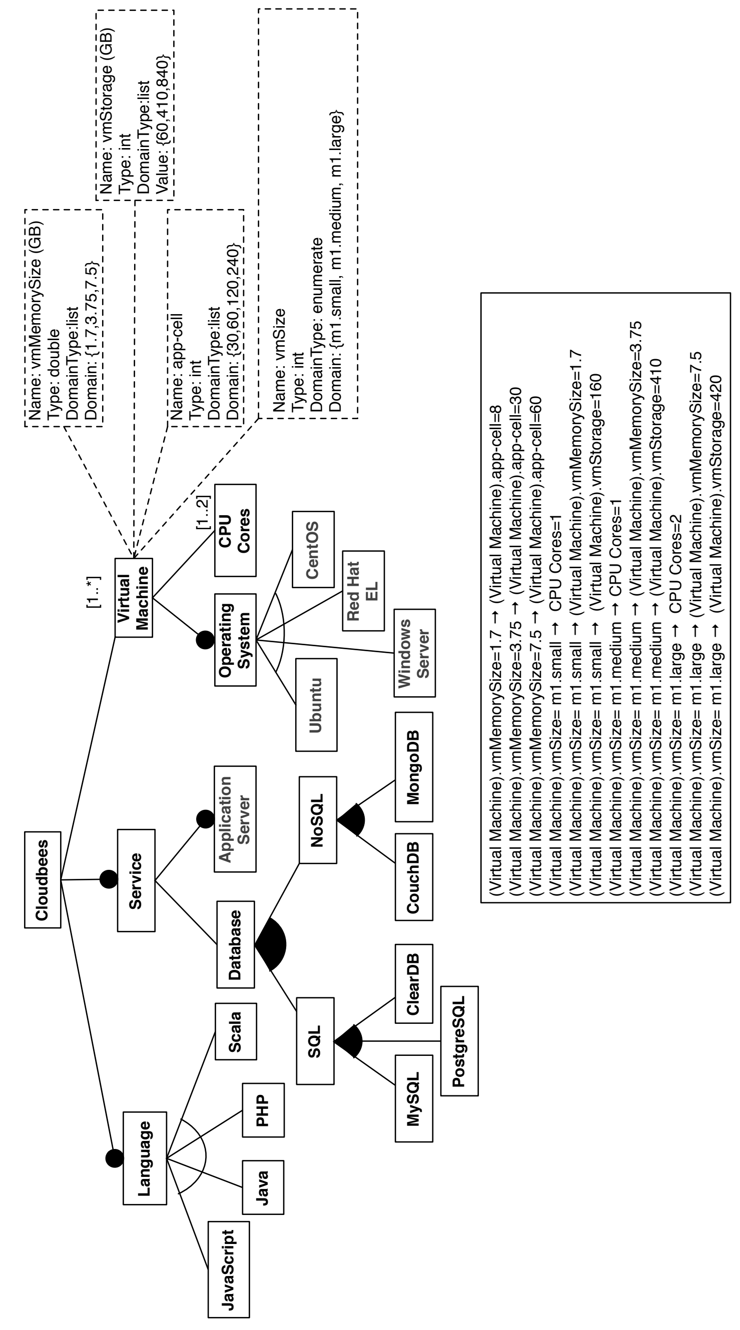


Figure A.1: The Cloud Knowledge Model.



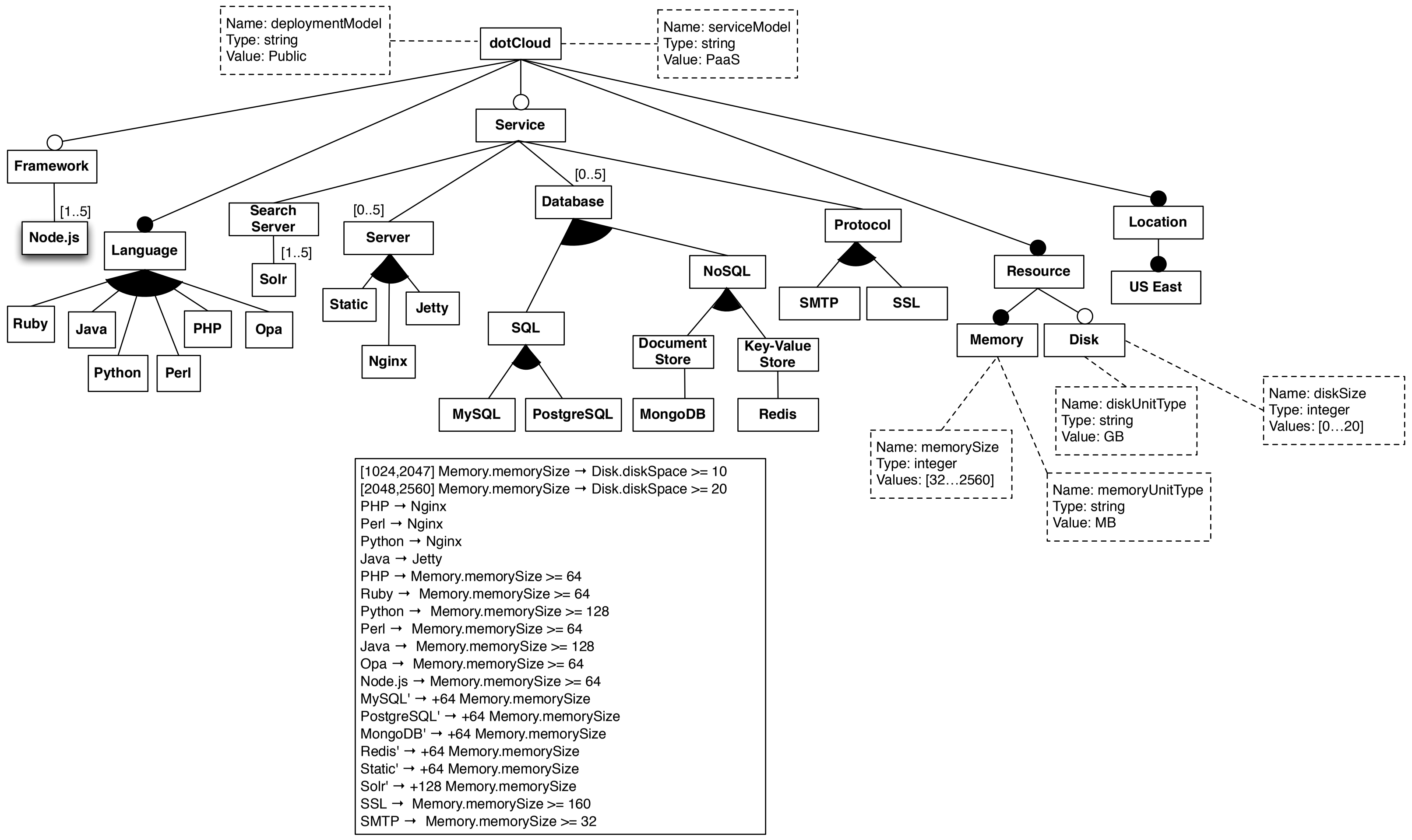
(Virtual Machine).vmSize= m3.medium → CPU Cores = 1
(Virtual Machine).vmSize= m3.medium → (Virtual Machine).vmMemorySize=3.75
(Virtual Machine).vmSize= m3.medium → (Virtual Machine).vmStorage=4
(Virtual Machine).vmSize= m3.large → CPU Cores = 2
(Virtual Machine).vmSize= m3.large → (Virtual Machine).vmMemorySize=7.5
(Virtual Machine).vmSize= m3.large → (Virtual Machine).vmStorage=32
(Virtual Machine).vmSize= m3.xlarge → CPU Cores = 4
(Virtual Machine).vmSize= m3.xlarge → (Virtual Machine).vmMemorySize=15
(Virtual Machine).vmSize= m3.xlarge → (Virtual Machine).vmStorage=80
(Virtual Machine).vmSize= m3.2xlarge → CPU Cores = 8
(Virtual Machine).vmSize= m3.2xlarge → (Virtual Machine).vmMemorySize=30
(Virtual Machine).vmSize= m3.2xlarge → (Virtual Machine).vmStorage=160
(Virtual Machine).vmSize= c3.large → CPU Cores = 2
(Virtual Machine).vmSize= c3.large → (Virtual Machine).vmMemorySize=3.75
(Virtual Machine).vmSize= c3.large → (Virtual Machine).vmStorage=32
(Virtual Machine).vmSize= c3.xlarge → CPU Cores = 4
(Virtual Machine).vmSize= c3.xlarge → (Virtual Machine).vmMemorySize=7.5
(Virtual Machine).vmSize= c3.xlarge → (Virtual Machine).vmStorage=80
(Virtual Machine).vmSize= c3.2xlarge → CPU Cores = 8
(Virtual Machine).vmSize= c3.2xlarge → (Virtual Machine).vmMemorySize=15
(Virtual Machine).vmSize= c3.2xlarge → (Virtual Machine).vmStorage=160
(Virtual Machine).vmSize= c3.4xlarge → CPU Cores = 16
(Virtual Machine).vmSize= c3.4xlarge → (Virtual Machine).vmMemorySize=30
(Virtual Machine).vmSize= c3.4xlarge → (Virtual Machine).vmStorage=320
(Virtual Machine).vmSize= c3.8xlarge → CPU Cores = 32
(Virtual Machine).vmSize= c3.8xlarge → (Virtual Machine).vmMemorySize=60
(Virtual Machine).vmSize= c3.8xlarge → (Virtual Machine).vmStorage=640
Auto Scaling → Amazon CloudWatch

Figure A.2: The Amazon EC2 feature model.



(Virtual Machine).vmMemorySize=1.7 → (Virtual Machine).app-cell=8
(Virtual Machine).vmMemorySize=3.75 → (Virtual Machine).app-cell=30
(Virtual Machine).vmMemorySize=7.5 → (Virtual Machine).app-cell=60
(Virtual Machine).vmSize= m1.small → CPU Cores=1
(Virtual Machine).vmSize= m1.small → (Virtual Machine).vmMemorySize=1.7
(Virtual Machine).vmSize= m1.small → (Virtual Machine).vmStorage=160
(Virtual Machine).vmSize= m1.medium → CPU Cores=1
(Virtual Machine).vmSize= m1.medium → (Virtual Machine).vmMemorySize=3.75
(Virtual Machine).vmSize= m1.medium → (Virtual Machine).vmStorage=410
(Virtual Machine).vmSize= m1.large → CPU Cores=2
(Virtual Machine).vmSize= m1.large → (Virtual Machine).vmMemorySize=7.5
(Virtual Machine).vmSize= m1.large → (Virtual Machine).vmStorage=420

Figure A.3: The Cloudbees feature model.



```
[1024,2047] Memory.memorySize → Disk.diskSpace >= 10
[2048,2560] Memory.memorySize → Disk.diskSpace >= 20
PHP → Nginx
Perl → Nginx
Python → Nginx
Java → Jetty
PHP → Memory.memorySize >= 64
Ruby →  Memory.memorySize >= 64
Python →  Memory.memorySize >= 128
Perl →  Memory.memorySize >= 64
Java →  Memory.memorySize >= 128
Opa →  Memory.memorySize >= 64
Node.js → Memory.memorySize >= 64
MySQL' → +64 Memory.memorySize
PostgreSQL' → +64 Memory.memorySize
MongoDB' → +64 Memory.memorySize
Redis' → +64 Memory.memorySize
Static' → +64 Memory.memorySize
Solr' → +128 Memory.memorySize
SSL →  Memory.memorySize >= 160
SMTP →  Memory.memorySize >= 32
```

Figure A.4: The Dotcloud feature model.

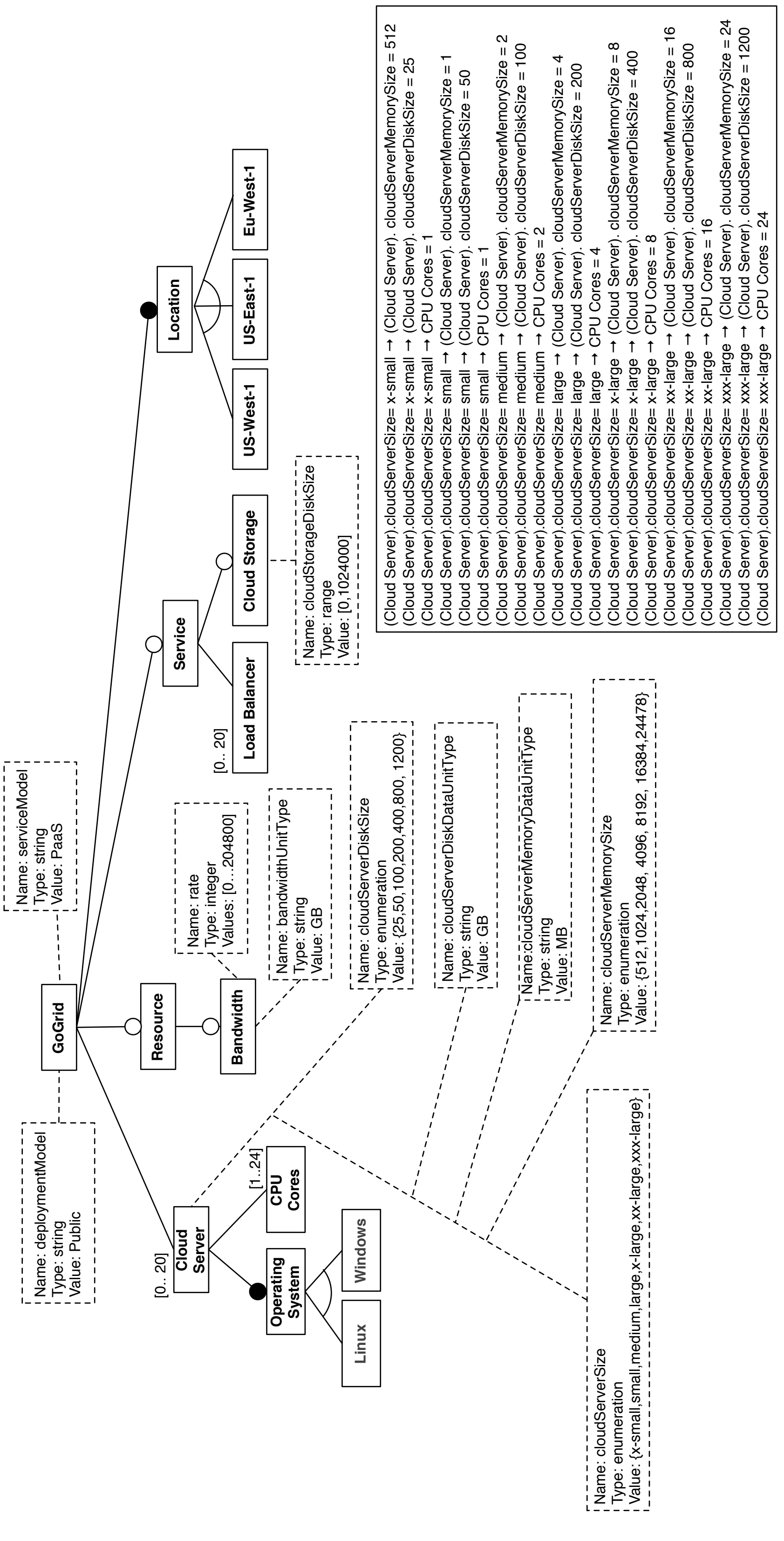

GoGrid
Name: deploymentModel
Type: string
Value: Public
Name: serviceModel
Type: string
Value: PaaS
Resource
Bandwidth
Name: rate
Type: integer
Values: [0...204800]
Name: bandwidthUnitType
Type: string
Value: GB
Service
[0.. 20]
Load Balancer
Cloud Storage
Name: cloudStorageDiskSize
Type: range
Value: [0,1024000]
Location
US-West-1
US-East-1
Eu-West-1
[0.. 20]
Cloud Server
[1..24]
CPU Cores
Operating System
Linux
Windows
Name: cloudServerDiskSize
Type: enumeration
Value: {25,50,100,200,400,800, 1200}
Name: cloudServerDiskDataUnitType
Type: string
Value: GB
Name:cloudServerMemoryDataUnitType
Type: string
Value: MB
Name: cloudServerMemorySize
Type: enumeration
Value: {512,1024,2048, 4096, 8192, 16384,24478}
Name: cloudServerSize
Type: enumeration
Value: {x-small,small,medium,large,x-large,xx-large,xxx-large}
(Cloud Server).cloudServerSize= x-small → (Cloud Server). cloudServerMemorySize = 512
(Cloud Server).cloudServerSize= x-small → (Cloud Server). cloudServerDiskSize = 25
(Cloud Server).cloudServerSize= x-small → CPU Cores = 1
(Cloud Server).cloudServerSize= small → (Cloud Server). cloudServerMemorySize = 1
(Cloud Server).cloudServerSize= small → (Cloud Server). cloudServerDiskSize = 50
(Cloud Server).cloudServerSize= small → CPU Cores = 1
(Cloud Server).cloudServerSize= medium → (Cloud Server). cloudServerMemorySize = 2
(Cloud Server).cloudServerSize= medium → (Cloud Server). cloudServerDiskSize = 100
(Cloud Server).cloudServerSize= medium → CPU Cores = 2
(Cloud Server).cloudServerSize= large → (Cloud Server). cloudServerMemorySize = 4
(Cloud Server).cloudServerSize= large → (Cloud Server). cloudServerDiskSize = 200
(Cloud Server).cloudServerSize= large → CPU Cores = 4
(Cloud Server).cloudServerSize= x-large → (Cloud Server). cloudServerMemorySize = 8
(Cloud Server).cloudServerSize= x-large → (Cloud Server). cloudServerDiskSize = 400
(Cloud Server).cloudServerSize= x-large → CPU Cores = 8
(Cloud Server).cloudServerSize= xx-large → (Cloud Server). cloudServerMemorySize = 16
(Cloud Server).cloudServerSize= xx-large → (Cloud Server). cloudServerDiskSize = 800
(Cloud Server).cloudServerSize= xx-large → CPU Cores = 16
(Cloud Server).cloudServerSize= xxx-large → (Cloud Server). cloudServerMemorySize = 24
(Cloud Server).cloudServerSize= xxx-large → (Cloud Server). cloudServerDiskSize = 1200
(Cloud Server).cloudServerSize= xxx-large → CPU Cores = 24

Figure A.5: The GoGrid feature model.

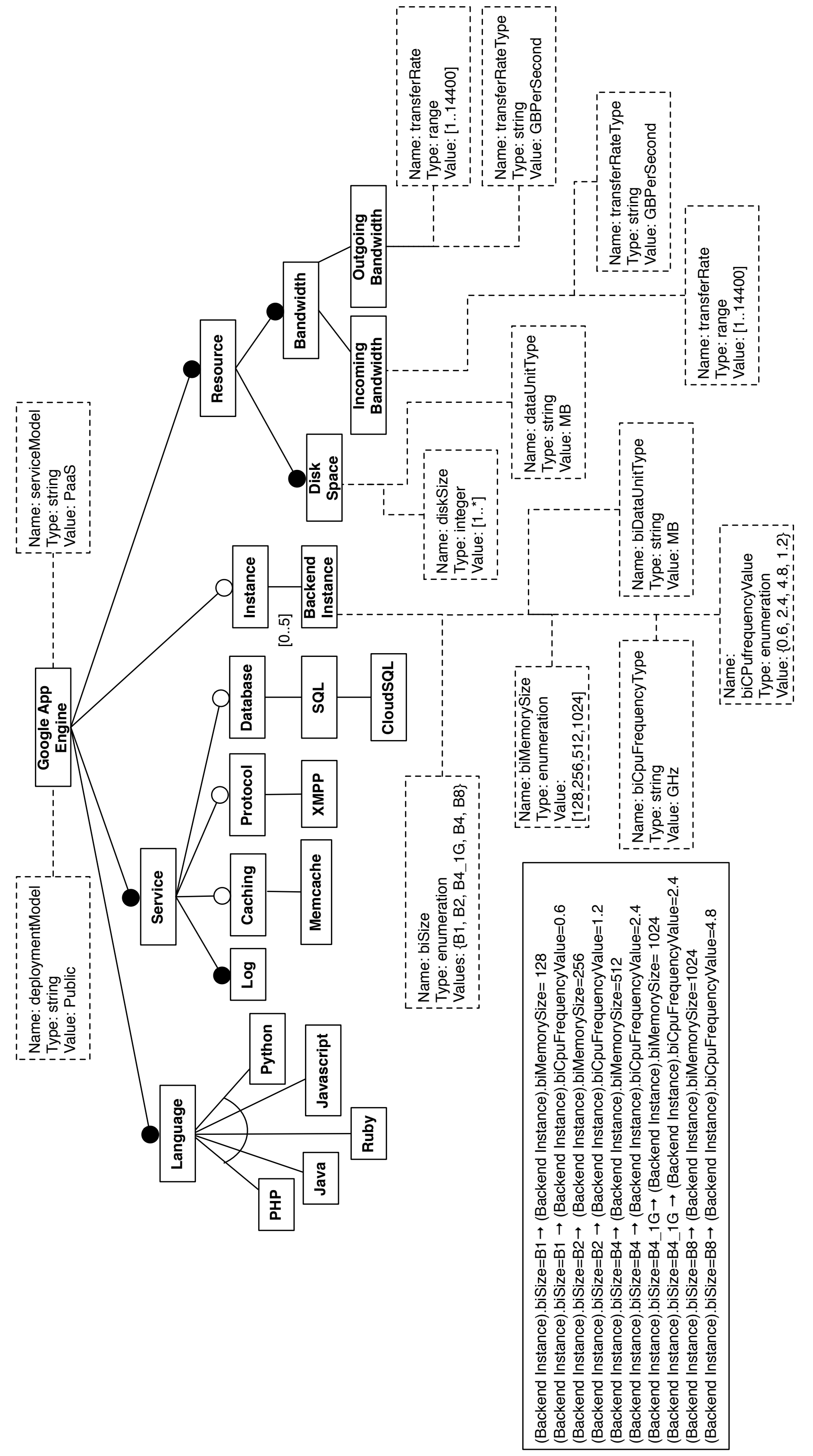

Google App Engine
Name: serviceModel
Type: string
Value: PaaS
Name: deploymentModel
Type: string
Value: Public
Language
Python
Javascript
Ruby
Java
PHP
Service
Log
Caching
Memcache
Protocol
XMPP
Database
SQL
CloudSQL
Instance
[0..5]
Backend Instance
Resource
Disk Space
Bandwidth
Incoming Bandwidth
Outgoing Bandwidth
Name: transferRate
Type: range
Value: [1..14400]
Name: transferRateType
Type: string
Value: GBPerSecond
Name: transferRateType
Type: string
Value: GBPerSecond
Name: transferRate
Type: range
Value: [1..14400]
Name: dataUnitType
Type: string
Value: MB
Name: diskSize
Type: integer
Value: [1..*]
Name: biDataUnitType
Type: string
Value: MB
Name: biCPufrequencyValue
Type: enumeration
Value: {0.6, 2.4, 4.8, 1.2}
Name: biSize
Type: enumeration
Values: {B1, B2, B4_1G, B4, B8}
Name: biMemorySize
Type: enumeration
Value: [128,256,512,1024]
Name: biCpuFrequencyType
Type: string
Value: GHz
(Backend Instance).biSize=B1 → (Backend Instance).biMemorySize= 128
(Backend Instance).biSize=B1 → (Backend Instance).biCpuFrequencyValue=0.6
(Backend Instance).biSize=B2 → (Backend Instance).biMemorySize=256
(Backend Instance).biSize=B2 → (Backend Instance).biCpuFrequencyValue=1.2
(Backend Instance).biSize=B4 → (Backend Instance).biMemorySize=512
(Backend Instance).biSize=B4 → (Backend Instance).biCpuFrequencyValue=2.4
(Backend Instance).biSize=B4_1G → (Backend Instance).biMemorySize= 1024
(Backend Instance).biSize=B4_1G → (Backend Instance).biCpuFrequencyValue=2.4
(Backend Instance).biSize=B8 → (Backend Instance).biMemorySize=1024
(Backend Instance).biSize=B8 → (Backend Instance).biCpuFrequencyValue=4.8

Figure A.6: The Google App Engine feature model.

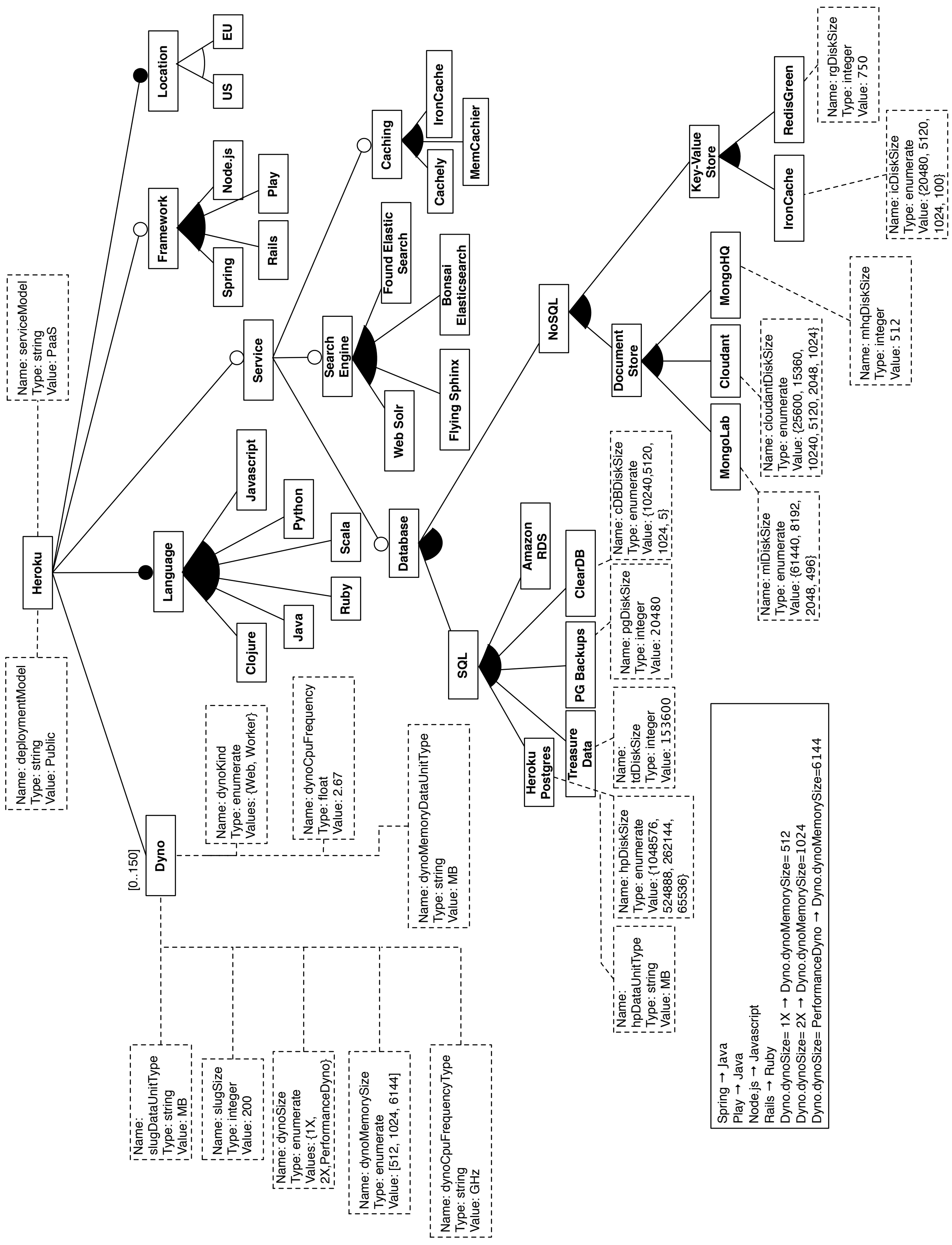


Figure A.7: The Heroku feature model.

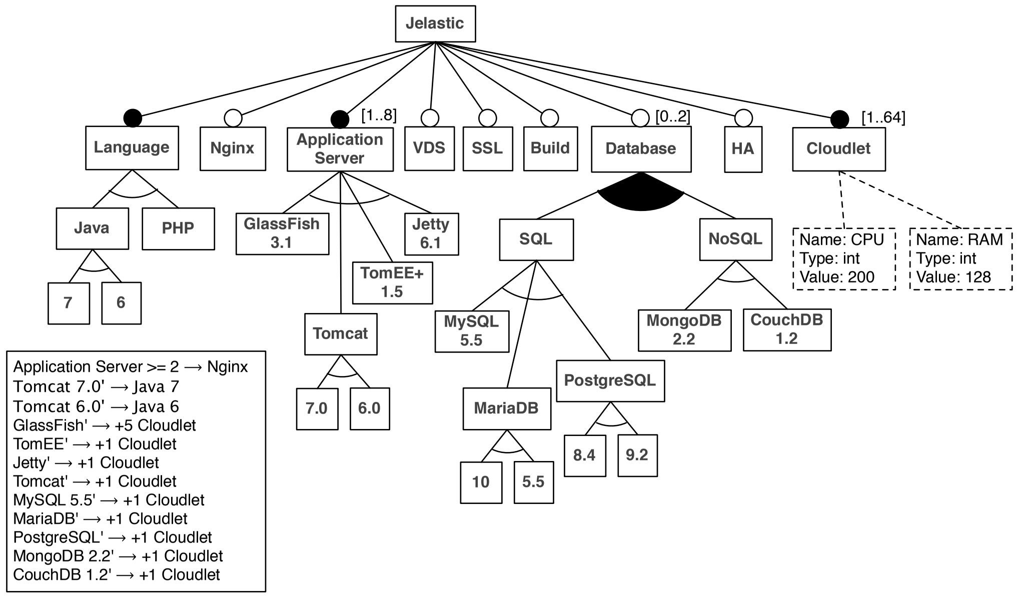
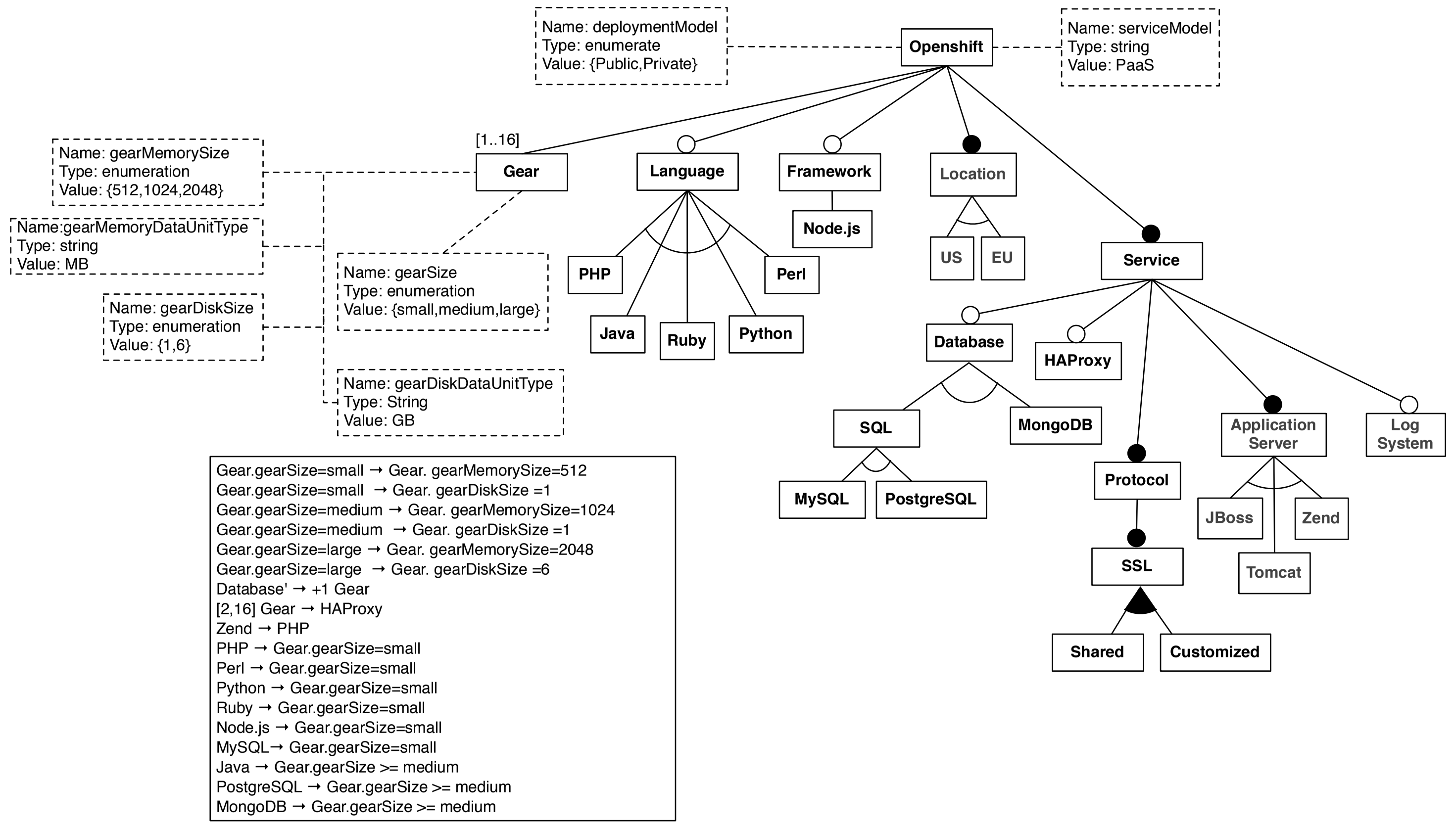


Figure A.8: The Jelastic feature model.



Gear.gearSize=small → Gear. gearMemorySize=512
Gear.gearSize=small → Gear. gearDiskSize =1
Gear.gearSize=medium → Gear. gearMemorySize=1024
Gear.gearSize=medium → Gear. gearDiskSize =1
Gear.gearSize=large → Gear. gearMemorySize=2048
Gear.gearSize=large → Gear. gearDiskSize =6
Database' → +1 Gear
[2,16] Gear → HAProxy
Zend → PHP
PHP → Gear.gearSize=small
Perl → Gear.gearSize=small
Python → Gear.gearSize=small
Ruby → Gear.gearSize=small
Node.js → Gear.gearSize=small
MySQL→ Gear.gearSize=small
Java → Gear.gearSize >= medium
PostgreSQL → Gear.gearSize >= medium
MongoDB → Gear.gearSize >= medium

Figure A.9: The OpenShift feature model.

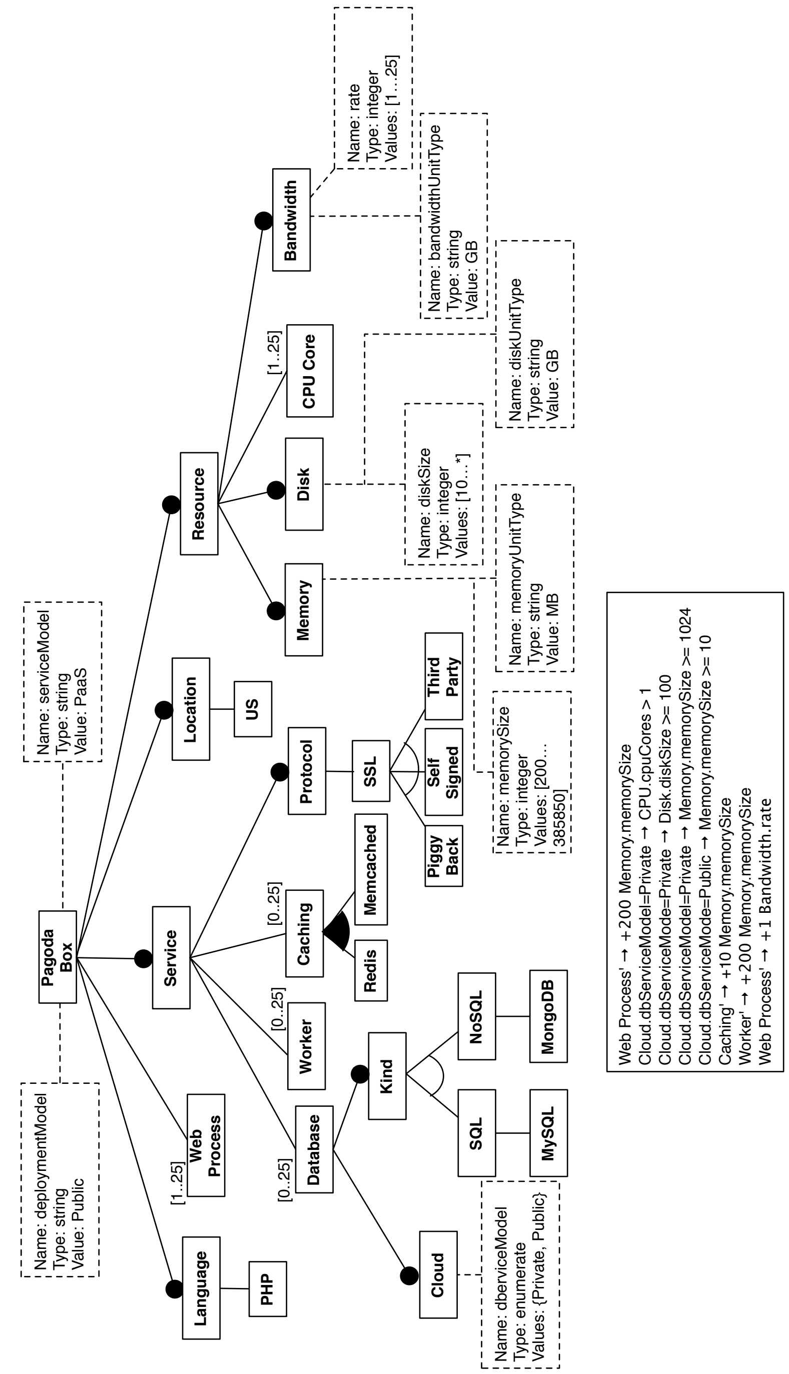
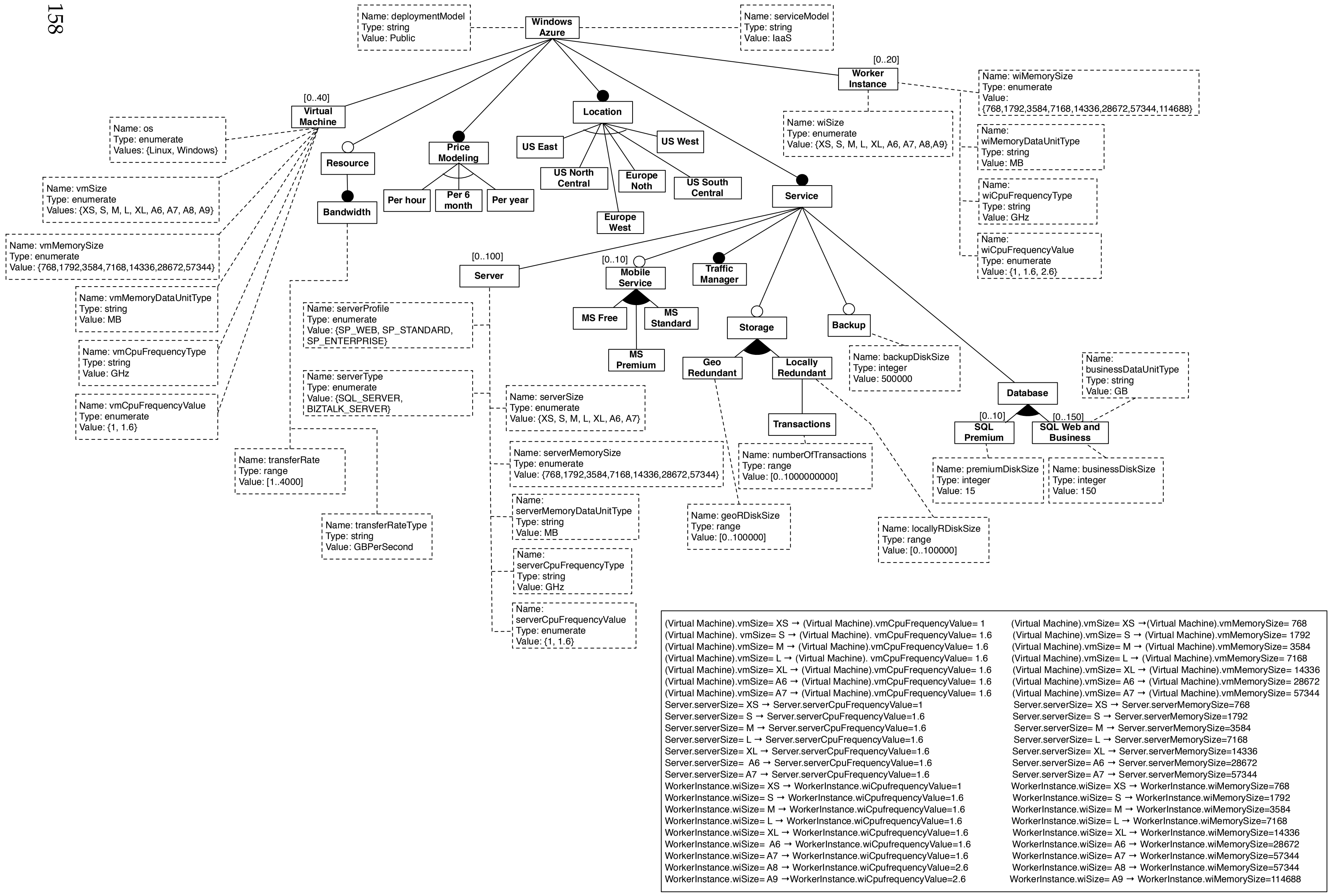


Figure A.10: The Pagoda Box feature model.



```
(Virtual Machine).vmSize= XS → (Virtual Machine).vmCpuFrequencyValue= 1        (Virtual Machine).vmSize= XS →(Virtual Machine).vmMemorySize= 768
(Virtual Machine). vmSize= S → (Virtual Machine). vmCpuFrequencyValue= 1.6     (Virtual Machine).vmSize= S → (Virtual Machine).vmMemorySize= 1792
(Virtual Machine).vmSize= M → (Virtual Machine).vmCpuFrequencyValue= 1.6       (Virtual Machine).vmSize= M → (Virtual Machine).vmMemorySize= 3584
(Virtual Machine).vmSize= L → (Virtual Machine). vmCpuFrequencyValue= 1.6      (Virtual Machine).vmSize= L → (Virtual Machine).vmMemorySize= 7168
(Virtual Machine).vmSize= XL → (Virtual Machine).vmCpuFrequencyValue= 1.6      (Virtual Machine).vmSize= XL → (Virtual Machine).vmMemorySize= 14336
(Virtual Machine).vmSize= A6 → (Virtual Machine).vmCpuFrequencyValue= 1.6      (Virtual Machine).vmSize= A6 → (Virtual Machine).vmMemorySize= 28672
(Virtual Machine).vmSize= A7 → (Virtual Machine).vmCpuFrequencyValue= 1.6      (Virtual Machine).vmSize= A7 → (Virtual Machine).vmMemorySize= 57344
Server.serverSize= XS → Server.serverCpuFrequencyValue=1                       Server.serverSize= XS → Server.serverMemorySize=768
Server.serverSize= S → Server.serverCpuFrequencyValue=1.6                      Server.serverSize= S → Server.serverMemorySize=1792
Server.serverSize= M → Server.serverCpuFrequencyValue=1.6                      Server.serverSize= M → Server.serverMemorySize=3584
Server.serverSize= L → Server.serverCpuFrequencyValue=1.6                      Server.serverSize= L → Server.serverMemorySize=7168
Server.serverSize= XL → Server.serverCpuFrequencyValue=1.6                     Server.serverSize= XL → Server.serverMemorySize=14336
Server.serverSize=  A6 → Server.serverCpuFrequencyValue=1.6                    Server.serverSize= A6 → Server.serverMemorySize=28672
Server.serverSize= A7 → Server.serverCpuFrequencyValue=1.6                     Server.serverSize= A7 → Server.serverMemorySize=57344
WorkerInstance.wiSize= XS → WorkerInstance.wiCpufrequencyValue=1               WorkerInstance.wiSize= XS → WorkerInstance.wiMemorySize=768
WorkerInstance.wiSize= S → WorkerInstance.wiCpufrequencyValue=1.6              WorkerInstance.wiSize= S → WorkerInstance.wiMemorySize=1792
WorkerInstance.wiSize= M → WorkerInstance.wiCpufrequencyValue=1.6              WorkerInstance.wiSize= M → WorkerInstance.wiMemorySize=3584
WorkerInstance.wiSize= L → WorkerInstance.wiCpufrequencyValue=1.6              WorkerInstance.wiSize= L → WorkerInstance.wiMemorySize=7168
WorkerInstance.wiSize= XL → WorkerInstance.wiCpufrequencyValue=1.6             WorkerInstance.wiSize= XL → WorkerInstance.wiMemorySize=14336
WorkerInstance.wiSize=  A6 → WorkerInstance.wiCpufrequencyValue=1.6            WorkerInstance.wiSize= A6 → WorkerInstance.wiMemorySize=28672
WorkerInstance.wiSize= A7 → WorkerInstance.wiCpufrequencyValue=1.6             WorkerInstance.wiSize= A7 → WorkerInstance.wiMemorySize=57344
WorkerInstance.wiSize= A8 → WorkerInstance.wiCpufrequencyValue=2.6             WorkerInstance.wiSize= A8 → WorkerInstance.wiMemorySize=57344
WorkerInstance.wiSize= A9 →WorkerInstance.wiCpufrequencyValue=2.6              WorkerInstance.wiSize= A9 → WorkerInstance.wiMemorySize=114688
```

Figure A.11: The Windows Azure feature model.

# Bibliography

[AAHC14] Ebrahim Khalil Abbasi, Mathieu Acher, Patrick Heymans, and Anthony Cleve. Reverse Engineering Web Configurators. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering, CSMR-WCRE'14*, pages 264–273, Feb 2014. 107, 144

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013. 4

[AC04] Michal Antkiewicz and Krzysztof Czarnecki. FeaturePlugin: Feature Modeling Plug-in for Eclipse. In *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange*, Eclipse'04, pages 67–72, New York, NY, USA, 2004. ACM. 41

[acc] Acceleo. http://www.eclipse.org/acceleo. 121

[Ach11] Mathieu Acher. *Managing Multiple Feature Models: Foundations, Language, and Applications*. PhD thesis, University of Nice Sophia Antipolis, Nice, France, sep 2011. 38, 64, 70

[ACLF11a] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. A Domain-specific Language for Managing Feature Models. In *Proceedings of the ACM Symposium on Applied Computing*, SAC'11, pages 1333–1340, New York, NY, USA, 2011. ACM. 40

[ACLF11b] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Slicing Feature Models. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 424–427, Washington, DC, USA, 2011. IEEE Computer Society. 143

[ADM14] ADM. Architecture-Driven Modernization (ADM). http://adm.omg.org, 2014. 32

[Aeo13] Aeolus. Aeolus: Mastering the Complexity of the Cloud. http://www.aeolus-project.org, 2013. 30, 35, 36, 37

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. 92

[AGM+06] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Refactoring Product Lines. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE'06, pages 201–210, New York, NY, USA, 2006. ACM. 45, 48

[ama] Amazon EC2. `http://aws.amazon.com/ec2`. 18, 19

[APS+10] Andreas Abele, Yiannis Papadopoulos, David Servat, Martin Törngren, and Matthias Weber. The CVM Framework - A Prototype Tool for Compositional Variability Management. In *Proceedings of the 4th International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS'10, pages 101–105. Universität Duisburg-Essen, 2010. 42

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg, 2005. Springer-Verlag. 39, 64

[BB01] Felix Bachmann and Len Bass. Managing Variability in Software Architectures. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, SSR'01, pages 126–132, New York, NY, USA, 2001. ACM. 24

[BB10] Barry Boehm and Kent Beck. Perspectives [The changing nature of software evolution; The inevitability of evolution]. *Software, IEEE*, 27(4):26–29, July 2010. 69

[BCM+03] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003. 65

[BCW11] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. Feature and Meta-models in Clafer: Mixed, Specialized, and Coupled. In *Proceedings of the Third International Conference on Software Language Engineering*, SLE'10, pages 102–122, Berlin, Heidelberg, 2011. Springer-Verlag. 39

[BFG+02] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J. Henk Obbink, and Klaus Pohl. Variability Issues in Software Product Lines. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE '01, pages 13–21, London, UK, UK, 2002. Springer-Verlag. 23

[BFL13] Christian Bessiere, Hélène Fargier, and Christophe Lecoutre. Global Inverse Consistency for Interactive Constraint Satisfaction. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, volume 8124 of *Lecture Notes in Computer Science*, pages 159–174. Springer Berlin Heidelberg, 2013. 80, 84, 89

[Big14] BigLever. BigLever Software Gears. http://www.biglever.com/solution/product.htm, 2014. 41

[BLMS12] Anton Belov, Inês Lynce, and João Marques-Silva. Towards efficient MUS extraction. *AI Communications*, 25(2):97–116, 2012. 85

[BP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–64, 2010. 84

[BP14] Goetz Botterweck and Andreas Pleuss. Evolution of software product lines. In Tom Mens, Alexander Serebrenik, and Anthony Cleve, editors, *Evolving Software Systems*, pages 265–295. Springer Berlin Heidelberg, 2014. 45, 70

[BR4] BR4CP. www.irit.fr/~Helene.Fargier/BR4CP/BR4CP.html. 83, 118

[BSR03] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-wise Refinement. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society. 39

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Inf. Syst.*, 35(6):615–636, September 2010. 27, 38, 64, 65, 72

[BSTC07] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz Cortés. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proceedings of the 1st International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS'07, pages 129–134, 2007. 40, 65

[BSTRC06] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. Using Java CSP Solvers in the Automated Analyses of Feature Models. In *Proceedings of the 2005 International Conference on Generative and Transformational Techniques in Software Engineering*, GTTSE'05, pages 399–408, Berlin, Heidelberg, 2006. Springer-Verlag. 45, 57, 65

[BTC05] David Benavides, Pablo Trinidad, and Antonio Ruiz Cortés. Using Constraint Programming to Reason on Feature Models. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering, SEKE'2005*, pages 677–682, 2005. 65

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated Reasoning on Feature Models. In *Proceedings of the 17th International Conference* *on Advanced Information Systems Engineering*, CAiSE'05, pages 491–503, Berlin, Heidelberg, 2005. Springer-Verlag. 27, 54

[Buy09] Rajkumar Buyya. Market-Oriented Cloud Computing: Vision, Hype, and Reality of Delivering Computing As the 5th Utility. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 1–1, Washington, DC, USA, 2009. IEEE Computer Society. 3, 16

[BYV$^+$09] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Gener. Comput. Syst.*, 25:599–616, June 2009. 3, 92

[CABA09] Lianping Chen, Muhammad Ali Babar, and Nour Ali. Variability Management in Software Product Lines: A Systematic Review. In *Proceedings of the 13th International Software Product Line Conference*, SPLC'09, pages 81–90, Pittsburgh, PA, USA, 2009. Carnegie Mellon University. 24

[CBH11] Andreas Classen, Quentin Boucher, and Patrick Heymans. A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL. *Science of Computer Programming*, 76(12):1130–1143, December 2011. 42, 44

[CBK13] Rafael Capilla, Jan Bosch, and Kyo Chul Kang, editors. *Systems and Software Variability Management, Concepts, Tools and Experiences*. Springer, 2013, 2013. 23

[CBUE02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. Generative Programming for Embedded Software: An Industrial Experience Report. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 156–172. Springer Berlin Heidelberg, 2002. 26, 54

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. 23, 25

[CFLGP06] Oscar Corcho, Mariano Fernández-López, and Asunción Gómez-Pérez. Ontological Engineering: Principles, Methods, Tools and Languages. In Coral Calero, Francisco Ruiz, and Mario Piattini, editors, *Ontologies for Software Engineering and Software Technology*, pages 1–48. Springer Berlin Heidelberg, 2006. 98

[CHE05a] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005. 27, 53, 57, 75

[CHE05b] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005. 27, 131

[CK05] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *International Workshop on Software Factories at OOPSLA'05*. ACM, San Diego, California, USA, 2005. 27, 53

[Cla14] Clafer. Clafer, Lightweight Modeling Language. `http://www.clafer.org`, 2014. 39

[CN01] Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., 2001. 4, 20, 21, 23

[CODA$^+$11] Joanna Dobroslawa Chimiak-Opoka, Birgit Demuth, Andreas Awenius, Dan Chiorean, Sebastien Gabel, Lars Hamann, and Edward D. Willink. OCL Tools Report based on the IDE4OCL Feature Model. *ECEASST*, 44, 2011. 68

[Con14a] ConPaaS. ConPaaS: Integrated Runtime Environment for Elastic Cloud Applications. `http://www.conpaas.eu`, 2014. 30, 35

[Con14b] Contrail. Contrail: Open Computing Infrastructures for Elastic Services. `http://contrail-project.eu`, 2014. 30

[CSHL13] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Beyond Boolean Product-line Model Checking: Dealing with Feature Attributes and Multi-features. In *Proceedings of the International Conference on Software Engineering*, ICSE '13, pages 472–481, Piscataway, NJ, USA, 2013. IEEE Press. 27, 143

[CVM14] CVM. CVM, Compositional Variability Management. `http://www.cvm-framework.org`, 2014. 42

[Dav87] Stanley M. Davis. *Future Perfect*. Addison-Wesley Reading, Mass, 1987. 20

[DCLT$^+$13] Roberto Di Cosmo, Michaël Lienhardt, Ralf Treinen, Stefano Zacchiroli, and Jakub Zwolakowski. Optimal Provisioning in the Cloud. Technical report of the Aeolus project. Technical report, Preuves, Programmes et Systèmes - PPS , FOCUS - INRIA Sophia Antipolis, March 2013. 30

[DCZZ12] Roberto Di Cosmo, Stefano Zacchiroli, and Gianluigi Zavattaro. Towards a Formal Component Model for the Cloud. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 156–171, Berlin, Heidelberg, 2012. Springer-Verlag. 30, 33, 34

[DHTCB14] Johan Den Haan, Mark Thiele, Nicholas Chase, and Matt Butcher. The 2014 Cloud Platform Research Report. Technical report, DZone, 2014. 102

[Don08] Don Batory. The GUIDSL Tool. `http://www.cs.utexas.edu/~schwartz/ATS/fopdocs/guidsl.html`, 2008. 39

[dot] DotCloud. https://www.dotcloud.com. 60

[DR97] Yves Dutuit and Antoine Rauzy. Exact and Truncated Computations of Prime Implicants of Coherent and non-Coherent Fault Trees within Aralia. *Reliability Engineering and System Safety*, 58(2):127–144, 1997. 83

[DSB05] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product Derivation in Software Product Families: A Case Study. *Journal of Systems and Software*, 74(2):173–194, January 2005. 22

[DWS12] Brian Dougherty, Jules White, and Douglas C. Schmidt. Model-driven Auto-scaling of Green Cloud Computing Infrastructure. *Future Gener. Comput. Syst.*, 28(2):371–378, February 2012. 44

[ES13] Holger Eichelberger and Klaus Schmid. A Systematic Analysis of Textual Variability Modeling Languages. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 12–21, New York, NY, USA, 2013. ACM. 39

[euc] Eucalyptus. https://www.eucalyptus.com. 19

[FAM14] FAMA. FaMa Tool Suite. http://www.isa.us.es/fama, 2014. 40

[Fea14] FeatureIDE. FeatureIDE, An Eclipse plug-in for Feature-Oriented Software Development. http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide, 2014. 40

[FFH13] Sören Frey, Florian Fittkau, and Wilhelm Hasselbring. Search-based Genetic Optimization for Deployment and Reconfiguration of Software in the Cloud. In *Proceedings of the International Conference on Software Engineering*, ICSE '13, pages 512–521, Piscataway, NJ, USA, 2013. IEEE Press. 32, 35

[FHS13] Sören Frey, Wilhelm Hasselbring, and Benjamin Schnoor. Automatic conformance checking for migrating software systems to cloud infrastructures and platforms. *Journal of Software: Evolution and Process*, 25(10):1089–1115, 2013. 30, 32, 34, 35, 36, 37

[FMP14] FMP. fmp: Feature Modeling Plug-in. http://gsd.uwaterloo.ca/fmp, 2014. 41

[FRC$^+$13] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems. In *Proceedings of the IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 887–894, Washington, DC, USA, 2013. IEEE Computer Society. 31, 34, 35, 37

[GAE] Google App Engine. https://appengine.google.com. 18

[GDD09] Dragan Gasevic, Dragan Djuric, and Vladan Devedzic. *Model Driven Engineering and Ontology Development (2. ed.)*. Springer, 2009. 98

[Gla] GlassFish support in Jelastic. http://docs.jelastic.com/glassfish. 62, 71

[Gla01] Robert L. Glass. Frequently Forgotten Fundamental Facts About Software Engineering. *IEEE Software*, 18(3):111–112, May 2001. 20

[GR10] Abel Gómez and Isidro Ramos. Cardinality-Based Feature Modeling and Model-Driven Engineering: Fitting them Together. In *Proceedings of the 4th International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS'10, pages 61–68. Universität Duisburg-Essen, 2010. 27

[GSK13] Alexander Gunka, Stepan Seycek, and Harald Kühn. Moving an Application to the Cloud: An Evolutionary Approach. In *Proceedings of the International Workshop on Multi-cloud Applications and Federated Clouds*, MultiCloud '13, pages 35–42, New York, NY, USA, 2013. ACM. 31

[GWTB12] Jianmei Guo, Yinglin Wang, Pablo Trinidad, and David Benavides. Consistency Maintenance for Evolving Feature Models. *Expert Systems with Applications*, 39(5):4987–4998, April 2012. 46, 48, 70

[her] Heroku. https://www.heroku.com. 18

[HHPS08] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, April 2008. 143

[HP03] Günter Halmans and Klaus Pohl. Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, 2(1):15–36, 2003. 23

[jel] Jelastic. http://jelastic.com. 18, 26, 54

[JO01] Narendra Jussien and Samir Ouis. User-friendly explanations for constraint programming. In *Proceedings of the Eleventh Workshop on Logic Programming Environments, WLPE'01*, 2001. 84

[jqu] The jQuery library. http://jquery.com. 116

[JRL08] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. Choco: an Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, Paris, France, France, 2008. 119

[jso] JavaScript Object Notation. http://www.json.org. 121

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003. 143

[KC12] Filip Křikava and Philippe Collet. On the use of an internal dsl for enriching emf models. In *Proceedings of the 12th Workshop on OCL and Textual Modelling*, OCL '12, pages 25–30, New York, NY, USA, 2012. ACM. 68

[KC13] Charles Krueger and Paul Clements. Systems and software product line engineering with biglever software gears. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC '13 Workshops, pages 136–140, New York, NY, USA, 2013. ACM. 41

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) - Feasibility Study. Technical report, The Software Engineering Institute, 1990. 4, 24, 27, 53

[Kle03] Leonard Kleinrock. An internet vision: the invisible global infrastructure. *Ad Hoc Networks*, 1(1):3–11, 2003. 3

[Kru92] Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992. 20

[Leh80] Manny Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept 1980. 69

[LMZ13] Tudor A. Lascu, Jacopo Mauro, and Gianluigi Zavattaro. A Planning Tool Supporting the Deployment of Cloud Applications. In *Proceedings of the IEEE 25th International Conference on Tools with Artificial Intelligence*, ICTAI '13, pages 213–220, Washington, DC, USA, 2013. IEEE Computer Society. 30

[LSB+10] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. Evolution of the Linux Kernel Variability Model. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC'10, pages 136–150, Berlin, Heidelberg, 2010. Springer-Verlag. 46, 48, 70

[Man02] Mike Mannion. Using First-Order Logic for Product Line Model Validation. In *Proceedings of the Second International Conference on Software Product Lines*, SPLC'02, pages 176–187, London, UK, UK, 2002. Springer-Verlag. 64

[Mat14] Mathieu Acher. FAMILIAR Project. http://familiar-project.github.io, 2014. 40

[MBC09] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 761–762, New York, NY, USA, 2009. ACM. 42

[MCHB11] Raphael Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. A Formal Semantics for Feature Cardinalities in Feature Diagrams. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS'11, pages 82–89, New York, NY, USA, 2011. ACM. 27, 53, 63, 67, 75

[MG11] Peter M. Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2011. 16, 17, 92

[MOD14] MODAClouds. MODAClouds: MOdel-Driven Approach for design and execution of applications on multiple Clouds. http://www.modaclouds.eu, 2014. 30, 31, 35

[MOF] MOF. http://www.omg.org/mof. 121

[Mos13] Mosaic. Mosaic: Open Source API and Platform for Multiple Clouds. http://www.mosaic-cloud.eu, 2013. 30, 31, 36

[MP14] Andreas Metzger and Klaus Pohl. Software Product Line Engineering and Variability Management: Achievements and Challenges. In *Proceedings of the Future of Software Engineering*, FOSE 2014, pages 70–84, New York, NY, USA, 2014. ACM. 23

[MPH$^+$07] Andreas. Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *15th IEEE International Requirements Engineering Conference, RE '07*, pages 243–253, Oct 2007. 23

[MR12] Michael Menzel and Rajiv Ranjan. CloudGenius: Decision Support for Web Server Cloud Migration. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 979–988, New York, NY, USA, 2012. ACM. 30, 32, 35, 36

[MS11] Parastoo Mohagheghi and Thor Sæther. Software Engineering Challenges for Migration to the Service Cloud Paradigm: Ongoing Work in the REMICS Project. In *Proceedings of the IEEE World Congress on Services*, SERVICES '11, pages 507–514, Washington, DC, USA, 2011. IEEE Computer Society. 32, 34, 35, 36

[MSDLM11] Raúl Mazo, Camille Salinesi, Daniel Diaz, and Alberto Lora-Michiels. Transforming Attribute and Clone-enabled Feature Models into Constraint Programs over Finite Domains. In *Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering*, ENASE'11, pages 188–199. SciTePress, 2011. 65

[MSNT11] Michael Menzel, Marten Schönherr, Jens Nimis, and Stefan Tai. (MC2)2: A Generic Decision-Making Framework and its Application to Cloud Computing. *CoRR*, abs/1112.1851, 2011. 32, 34, 35

[NTS+11] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulezsa, and Paulo Borba. Investigating the Safe Evolution of Software Product Lines. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE'11, pages 33–42, New York, NY, USA, 2011. ACM. 46, 48

[ope] OpenStack. https://www.openstack.org/. 19

[Paa13] PaaSage. PaaSage: Model-based Cloud Platform Upperware. http://www.paasage.eu, 2013. 4, 62, 92

[Par76] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976. 20

[Par11] Carlos Parra. *Towards Dynamic Software Product Lines: Unifying Design and Runtime Adaptations*. These, Université des Sciences et Technologie de Lille - Lille I, March 2011. 57

[PBD+12] Andreas Pleuss, Goetz Botterweck, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. Model-driven Support for Product Line Evolution on Feature Level. *Journal of Systems and Software*, 85(10):2261–2274, October 2012. 47, 48

[PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. 4, 20, 21, 22, 23, 24, 57

[PCN+11] Dana Petcu, Ciprian Crăciun, Marian Neagul, Silviu Panica, Beniamino Di Martino, Salvatore Venticinque, Massimiliano Rak, and Rocco Aversa. Architecturing a Sky Computing Platform. In *Proceedings of the International Conference on Towards a Service-based Internet*, ServiceWave'10, pages 1–13, Berlin, Heidelberg, 2011. Springer-Verlag. 31, 35

[PCW12] Leonardo Passos, Krzysztof Czarnecki, and Andrzej Wąsowski. Towards a Catalog of Variability Evolution Patterns: The Linux Kernel Case. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, FOSD'12, pages 62–69, New York, NY, USA, 2012. ACM. 46, 70

[PDv12] Paulius Paskevicius, Robertas Damasevicius, and Vytautas Štuikys. Change Impact Analysis of Feature Models. In Tomas Skersys, Rimantas Butleris, and Rita Butkiene, editors, *Information and Software Technologies*, volume 319 of *Communications in Computer and Information Science*, pages 108–122. Springer Berlin Heidelberg, 2012. 70

[PGT+13] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski, and Paulo Borba. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In *Proceedings of the 17th International Software Product Line Conference*, SPLC'13, pages 91–100, New York, NY, USA, 2013. ACM. 46, 48, 70

[PHM+12] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy, and L. Seinturier. A Federated Multi-cloud PaaS Infrastructure. In *IEEE 5th International Conference on Cloud Computing, CLOUD'12*, pages 392–399, June 2012. 143

[PS12] Guillaume Pierre and Corina Stratan. ConPaaS: A Platform for Hosting Elastic Cloud Applications. *IEEE Internet Computing*, 16(5):88–92, 2012. 31, 34, 35

[pur14a] pure::systems. pure::variants: Variant Management. `http://www.pure-systems.com/pure_variants.49.0.html`, 2014. 41

[pur14b] pure::systems. Variant Management with pure::variants. `http://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf`, 2014. 41

[RBSP02] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending Feature Diagrams with UML Multiplicities. In *6th World Conference on Integrated Design & Process Technology, IDPT'02*, June 2002. 27, 53, 61

[REM13] REMICS. REMICS: Reuse and Migration of legacy applications to Interoperable Cloud Services. `http://www.remics.eu`, 2013. 30, 32, 35, 37

[RGD10] Rick Rabiser, Paul Grünbacher, and Deepak Dhungana. Requirements for Product Derivation Support: Results from a Systematic Literature Review and an Expert Survey. *Inf. Softw. Technol.*, 52(3):324–346, March 2010. 70

[RSTS11] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. Multi-dimensional Variability Modeling. In *Proceedings of the 5th Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '11, pages 11–20, New York, NY, USA, 2011. ACM. 42

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009. 115

[SHA12] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. Co-evolution of Models and Feature Mapping in Software Product Lines. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC'12, pages 76–85, New York, NY, USA, 2012. ACM. 47, 48, 70

[SJP12] Vlado Stankovski, Jernej Juzna, and Dana Petcu. Enabling Legacy Engineering Applications for Cloud Computing: Experience with the mOSAIC API and

Platform. In *Proceedings of the Third International Conference on Emerging Intelligent Data and Web Technologies*, EIDWT '12, pages 281–286, Washington, DC, USA, 2012. IEEE Computer Society. 31, 34

[SLB$^+$11] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 461–470, New York, NY, USA, 2011. ACM. 127

[spl] S.P.L.O.T. `http://www.splot-research.org/`. 41, 127

[SRG11] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In *Proceedings of the 5th Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS'11, pages 119–126, New York, NY, USA, 2011. ACM. 24

[SRP03] Detlef Streitferdt, Matthias Riebisch, and Ilka Philippow. Details of formalized relations in feature models using OCL. In *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 297–304, April 2003. 27

[SSA13] Christoph Seidl, Ina Schaefer, and Uwe Assmann. Capturing Variability in Space and Time with Hyper Feature Models. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS'14, pages 6:1–6:8, New York, NY, USA, 2013. ACM. 24

[SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A Taxonomy of Variability Realization Techniques: Research Articles. *Softw. Pract. Exper.*, 35(8):705–754, July 2005. 24

[TBK09] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning About Edits to Feature Models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 254–264, Washington, DC, USA, 2009. IEEE Computer Society. 47, 48, 68, 70, 128, 130

[TBRC$^+$08] Pablo Trinidad, David Benavides, Antonio Ruiz-Cortés, Sergio Segura, and Alberto Jimenez. FAMA Framework. In *Proceedings of the 12th International Software Product Line Conference*, SPLC'08, pages 359–359, Washington, DC, USA, 2008. IEEE Computer Society. 40

[TKB$^+$14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-oriented Software Development. *Science of Computer Programming*, 79:70–85, January 2014. 40

[TR09] P. Trinidad and A. Ruiz–Cortés. Abductive Reasoning and Automated Analysis of Feature Models: How are they connected? In *Proceedings of the Third International Workshop on Variability Modelling of Software-Intensive Systems.*, pages 145–153, 2009. 49, 72

[TVL14] TVL. TVL - A Text-based Variability Language. `https://staff.info.unamur.be/acs/tvl`, 2014. 42

[VDA10] Wil M. P. Van Der Aalst. Configurable Services in the Cloud: Supporting Variability While Enabling Cross-organizational Process Mining. In *Proceedings of the International Conference on On the Move to Meaningful Internet Systems - Volume Part I*, OTM'10, pages 8–25, Berlin, Heidelberg, 2010. Springer-Verlag. 38

[vDK02] Arie van Deursen and Paul Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002. 40

[VEL14] VELVET. The VELVET Modeling Language. `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/multiple/modeling.php`, 2014. 42

[VRMCL08] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds: Towards a Cloud Definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008. 16

[WDS09] Jules White, Brian Dougherty, and Douglas C. Schmidt. Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening. *J. Syst. Softw.*, 82(8):1268–1284, August 2009. 27

[Win] Windows Azure. `http://azure.microsoft.com`. 18, 19

[WKM12] Erik Wittern, Jörn Kuhlenkamp, and Michael Menzel. Cloud Service Selection Based on Variability Modeling. In *Proceedings of the 10th International Conference on Service-Oriented Computing*, ICSOC'12, pages 127–141, Berlin, Heidelberg, 2012. Springer-Verlag. 44

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product-line Engineering: A Family-based Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. 21, 24

[WLS$^+$05] Hai Wang, Yuan Fang Li, Jing Sun, Hongyu Zhang, and Jeff Pan. A Semantic Web Approach to Feature Modeling and Verification. In *In Workshop on Semantic Web Enabled Software Engineering, SWESE'05*, 2005. 65

[WSH$^+$12] Peter Wright, YihLeong Sun, Terence Harmer, Anthony Keenan, Alan Stewart, and Ronald Perrott. A constraints-based resource discovery model for multi-provider cloud environments. *Journal of Cloud Computing*, 1(1), 2012. 30, 33, 34, 35, 36