École Doctorale Sciences Pour l'Ingenieur, Université de Lille Nord-de-France

CRIStAL UMR 9189
Inria

A Software Product Lines-Based Approach for the Setup and Adaptation of Multi-Cloud Environments

THÈSE

préparée pour obtenir le grade de
Docteur en Informatique et applications de l'Université de Lille

présentée et soutenue publiquement par

Gustavo Sousa

le 5 juin 2018

Directeur de thèse:	Laurence Duchien, *Université de Lille*
Co-encadrant:	Walter Rudametkin, *Université de Lille*
Rapporteurs:	Philippe Collet, *Université Nice Sophia Antipolis*
	Camille Salinesi, *Université Paris 1 Panthéon-Sorbonne*
Examinateurs:	Danilo Ardagna, *Politecnico di Milano*
	Hélène Coullon, *IMT Atlantique*
	Patrick Heymans, *Université de Namur*

This manuscript was prepared by:
Gustavo Sousa
Université de Lille

Rapporteurs:
Philippe Collet, *Université Nice Sophia Antipolis*
Camille Salinesi, *Université Paris 1 Panthéon-Sorbonne*

Examinators:
Danilo Ardagna, *Politecnico di Milano*
Hélène Coullon, *IMT Atlantique*
Patrick Heymans, *Université de Namur*

Advisor:
Laurence Duchien, *Université de Lille*

Co-advisor:
Walter Rudametkin, *Université de Lille*

Abstract

Cloud computing is characterized by a model in which computing resources are delivered as services in a pay-as-you-go manner, which eliminates the need for upfront investments, reducing the time to market and opportunity costs. Despite its benefits, cloud computing brought new concerns about provider dependence and data confidentiality, which further led to a growing trend on consuming resources from multiple clouds. However, building multi-cloud systems is still very challenging and time consuming due to the heterogeneity across cloud providers' offerings and the high-variability in the configuration of cloud providers. This variability is expressed by the large number of available services and the many different ways in which they can be combined and configured. In order to ensure correct setup of a multi-cloud environment, developers must be aware of service offerings and configuration options from multiple cloud providers.

To tackle this problem, this thesis proposes a software product lines-based approach for managing the variability in cloud environments in order to automate the setup and adaptation of multi-cloud environments. The contributions of this thesis enable to automatically generate a configuration or reconfiguration plan for a multi-cloud environment from a description of its requirements. The conducted experiments aim to assess the impact of the approach on the automated analysis of feature models and the feasibility of the approach to automate the setup and adaptation of multi-cloud environments.

Keywords: software product lines, cloud computing, feature models, dynamic software product lines

Résumé

Le cloud computing est caractérisé par un modèle dans lequel les ressources informatiques sont fournies en tant qu'un service d'utilité, ce qui élimine le besoin de grands investissements initiaux. Malgré ses avantages, le cloud computing a suscité de nouvelles inquiétudes concernant la dépendance des fournisseurs et la confidentialité des données, ce qui a conduit à l'émergence des approches multi-cloud. Cependant, la construction de systèmes multi-cloud est toujours difficile en raison de l'hétérogénéité entre les offres des fournisseurs de cloud et de la grande variabilité dans la configuration des fournisseurs de cloud. Cette variabilité est caractérisée par le grand nombre de services disponibles et les nombreuses façons différentes de les combiner et de les configurer. Afin de garantir la configuration correcte d'un environnement multi-cloud, les développeurs doivent connaître les offres de services et les options de configuration de plusieurs fournisseurs de cloud.

Pour traiter ce problème, cette thèse propose une approche basée sur les lignes de produits logiciels pour gérer la variabilité dans les cloud afin d'automatiser la configuration et l'adaptation des environnements multi-cloud. Les contributions de cette thèse permettent de générer automatiquement un plan de configuration ou de reconfiguration pour un environnement multi-cloud à partir d'une description de ses exigences. Les expérimentations menées visent à évaluer l'impact de l'approche sur l'analyse automatisée des modèles de caractéristiques et la faisabilité de l'approche pour automatiser la configuration et l'adaptation des environnements multi-nuages.

Mots-clés : lignes des produits logiciels, informatique en nuage, modèles de caracteristiques, lignes de produits logiciels dynamiques

Table of contents

List of figures

List of tables

Chapter 1

Introduction

The cloud computing paradigm is characterized by a model in which computing resources can be rapidly provisioned and released with minimal management effort [1]. One of the main properties of cloud computing is the ability to pay for computing resources based on their utilization [2]. This *pay-as-you-go* model eliminates the need for upfront investments in computing infrastructure, turning it into an utility service like water, gas, electricity and telecommunication. From a business perspective, cloud computing is a disruptive technology that is changing the way software is developed and consumed [3]. By eliminating the need for upfront investments, cloud computing has the potential to reduce opportunity costs for new business and to foster the emergence of innovative ideas [2, 4, 5]. The ability to rapidly provision or release resources in cloud systems allows for dynamically changing the resources allocated to a software system. Hence, cloud computing can also be regarded as a highly flexible computing infrastructure that can be dynamically adapted to cope with changes in application demand.

Despite its benefits, cloud computing has also brought new concerns about provider dependence, vendor lock-in, and data confidentiality [2, 6]. Lack of standardization hinders cloud provider interoperability and increases the risk of being locked into proprietary solutions, making cloud customers vulnerable to unilateral changes by cloud providers. When sensitive data is at stake, organizations may be skeptical about moving their data to a computing infrastructure that is not under their control. Furthermore, cloud customers are powerless and completely dependent on the provider's actions in the case of failures or service interruptions.

These concerns have led to the growing trend of consuming resources from multiple cloud providers. Various approaches have emerged to deal with the complexities of using multiple cloud providers [7–10]. Among these approaches, *multi-cloud computing* is characterized by the use of services from multiple cloud providers when there is no agreement between them to provide integrated access to their resources [7]. In a multi-cloud system, it is up to the cloud customer or a third-party to integrate cloud services from different providers into a unified system. Multiple clouds are mainly used to reduce vendor dependence, comply with regulatory constraints, and better exploit cloud market offerings. By combining services from different private and public providers, cloud customers can build a cloud environment that better supports system requirements while reducing provider dependence and costs.

1.1 Motivation and Scope

Despite the benefits of multi-cloud computing, particularly in regards to provider independence, it introduces new challenges. The large number of available cloud providers as well as the heterogeneity across their service offerings make it very difficult for cloud customers to identify which providers and services are a best fit for their application. When setting up a multi-cloud environment, customers must be aware of the services offered by different providers and of the constraints governing their configuration. In addition, the maintenance and evolution of a system that relies on multiple providers, each one with their own sets of policies, concepts and management interfaces, is time consuming and error prone. Due to these reasons, it becomes difficult, or even unfeasible, to build multi-cloud systems without adequate tool support.

Over the past few years, many approaches have emerged to deal with aspects involved in the construction and management of multi-cloud systems [11–22]. These projects propose new ways of dealing with the issues related to multi-cloud systems, such as the discovery and selection of cloud resources, automated deployment, monitoring and management of heterogeneous clouds, among others. However, most of these approaches neglect the high-variability that can be found in the configuration of cloud providers. This variability is reflected in the wide range of services offered by cloud providers, each of which can be configured in a number of ways. For instance, the Amazon EC2 service, which is a single service, has a total of 16,991 possible configurations [23]. These configurations are obtained by combining the available configuration options in different ways. Cloud providers exhibit complex constraints that define how their offered services

can be combined and configured. Neglecting this variability in the configuration of cloud providers may lead to invalid configurations that do not match their actual service offerings.

Some works have been proposed [24–26, 23, 21] to manage variability in the configuration of cloud resources or environments. These works are based on Software Product Line Engineering (SPLE) [27], a systematic approach for building a family of related software systems from a set of reusable common assets. The main element of a software product line is a variability model, such as a feature model, that specifies how a set of common assets can be assembled into a software product. SPLE is mostly used to support the construction of customized software products with reduced costs and high-quality. In the cloud computing domain, SPLE has been used to set up customized cloud environments, tailored to applications [24–26, 23, 21]. Feature models are then used to define the variability of cloud configurations. Cloud services and configuration options are depicted as features organized in a hierarchical structure, while additional constraints define further dependencies among features. SPLE provides a large body of techniques and tools for the analysis of feature models and derivation of configurations.

However, in the existing works, variability is handled only in a limited context, such as a single cloud resource (e.g., virtual machines, storage, networking, etc), or for single-cloud applications. Only one of these works provides some level of support for multi-cloud systems [21], but variability management is limited to infrastructure resources. When variability is handled only for the configuration of a single resource or service, constraints that may exist between them are overlooked. Configurations that may look valid when analyzing the variability of each individual cloud service, may turn out to be inconsistent when multiple services and resources are set up together.

These works also have no regard for the dynamic variability that comes into play when reconfiguring a cloud environment to cope with changes in application needs. Cloud providers may exhibit complex rules governing how a running cloud environment can be reconfigured. The availability of configuration options may depend on previous configurations, and some reconfigurations may require executing extra operations. When adapting a running cloud environment, these constraints must also be taken into account to correctly reconfigure it to support new requirements. Lack of support for managing this complexity is a limiting factor for the adoption of multi-cloud computing.

The limitations in previous works point to a lack of support for the management of variability in multi-cloud systems. In this thesis we propose an approach for the setup

and adaptation of multi-cloud environments that acknowledges the inherent variability that arises from multi-cloud environments.

Building adaptive multi-cloud systems requires dealing with a wide range of issues related to the provisioning, development, monitoring and management of heterogeneous cloud environments [28, 29]. In this work, we focus on the particular problem of managing configuration variability and handling the heterogeneity across providers, with the goal of supporting the setup and adaptation of multi-cloud environments. Therefore, activities related to the development, provisioning, deployment and monitoring of multi-cloud systems are outside the scope of this thesis. The goal of this work is to manage the variability in the configuration of cloud providers in order to build a configuration or reconfiguration plan for a multi-cloud environment that conforms with cloud providers' configuration rules.

1.2 Objectives

In this thesis, we deal with the problem of managing variability in cloud providers to support the setup and adaptation of multi-cloud environments. More precisely, we look to **bridge the gap from a description of application needs to a tailored multi-cloud environment; and adapt this environment to cope with changes in application needs**. A multi-cloud environment is conceived of as a selection of cloud services from private and public providers, that are used to deploy and run an application. The goal is to shield developers from the complexities related to the heterogeneity and variability in cloud provider configurations and enable them to automatically build a multi-cloud environment configuration that correctly supports their applications.

As discussed earlier, many recent works have been proposed to deal with aspects related to the construction of multi-cloud systems. This work distinguishes itself by its support for **service-based applications**, its management of **heterogeneity** across cloud providers, and its handling of the **variability** in their configurations. Also, this approach addresses not only the initial setup of cloud environments, but also their adaptation as the application needs change over time. The remainder of this section discusses the importance of these aspects, as well as their implications and the challenges they introduce.

Service-based Application Model

We consider a service-based application model in which an application is composed of a set of collaborating services that can be distributed across multiple private or public clouds from different providers. We assume that application services can be developed by different teams, using different methods, and thus may require different services and resources from cloud providers. Also, each application service may have different requirements concerning scalability, redundancy and data privacy. For example, to comply with data protection policies a service that manages sensitive data may need to be placed in a private cloud or within the boundaries of a given country. Meanwhile, a critical service may need to be deployed to multiple providers to allow for extra redundancy and failure protection.

This application model encompasses the main types of architectures currently used in development of cloud applications, such as service-oriented architectures [30] and microservices [31]. Monolithic or multi-tiered applications can be represented as special cases of service-based applications and also benefit from the solutions we propose in this work.

The choice of a service-based application model has implications in the way variability is managed in a cloud environment. In previous works, variability is only handled at the resource level [24, 25, 23, 21], or only handled for tiered applications that have a predefined number of services (e.g., web server, application server, database) [26]. In the first case, variability is managed in a limited context and constraints that may arise between different resources are ignored. In the second case, variability is modeled in a way that does not support the setup of cloud environments for service-based applications because it is only possible to setup a single application server for the whole configuration.

When setting up a cloud environment for a service-based application, multiple application services may be deployed to the same cloud provider. The variability management mechanism should support variability modeling at the provider level, in a way that makes it possible to describe a cloud environment configuration for deploying multiple application services.

Heterogeneity across Cloud Providers

Cloud providers deliver computing resources as services at different levels of abstraction, the more popular being infrastructure (IaaS), platform (PaaS), and software (SaaS).

While some providers are focused on one abstraction level, such as Heroku PaaS or the Rackspace IaaS providers, others such as Google and Amazon offer services at multiple levels of abstraction. Even at the same abstraction level, providers employ different terminology, concepts and services that may not directly map to those of competing providers. As an example, for the IaaS virtual machine service the choice of operating systems, geographical region, processing power and memory can vary significantly across providers. At platform and software levels, cloud services are even more specialized and heterogeneous. For instance, support for Java applications can range from a simple runtime environment with a Java Development Kit installed, to a full-blown JavaEE server such as JBoss Enterprise Application Platform.

These differences across cloud provider offerings make it very difficult for cloud customers to establish a comparison between different offerings and identify the cloud services that are more suitable for their applications. Additionally, in service-based applications, each application service may have different needs concerning the functionality they require from cloud providers. Different application services may be designed to consume cloud services at varying levels of abstraction. For example, an application service developed using standardized Java Enterprise technology might only require a JavaEE EJB container service, while a standalone application service may be designed in a way that requires access to operating system functions and thus should be deployed directly into a virtual machine.

When working with multiple cloud providers it is necessary to identify across heterogeneous cloud services, which ones provide the functionality and properties required by application services. This requires a way to semantically compare application service requirements to actual cloud provider offerings in order to identify which cloud services support these requirements. This also requires to define application needs in a way that is independent of the actual cloud provider offerings.

Static and Dynamic Variability in Cloud Environments

Cloud providers often have complex rules that govern how their services can be configured and combined in a cloud environment. In order to correctly configure cloud providers, we need to represent this variability and ensure that generated configurations comply with these rules. In service-based applications, we must also take into account that a configuration for a cloud provider encompasses cloud services required by multiple application services. This requires considering variability, not only in the context of a

specific cloud service or a single application service, but for multiple application services at the provider level. Besides this static variability that defines the configuration rules for a cloud environment, we also have to consider the dynamic variability that defines how a running cloud environment can be adapted. Dynamic variability, also called runtime variability, defines how a system can be modified at runtime. Cloud providers may also have their own unique rules concerning how a running cloud environment can be updated. When reconfiguring a cloud environment, the available cloud services and configuration options may also depend on previous configuration choices.

To manage the variability found in the configuration of cloud computing providers we need means for representing this variability and evaluating it against application requirements in order to generate complying configurations.

The work presented in this dissertation aims to investigate the use of principles from software product lines to automate the setup and adaptation of multi-cloud environments. In particular, it is focused on the challenges introduced by the high variability that exists in the configuration of cloud environments. Following the aspects discussed above, this thesis addresses the following research questions.

The first research question is related to the need for managing variability in the configuration of cloud environments. The variability in cloud environments emerges from the large number of ways in which cloud services can be configured and combined together into valid configurations. The number of cloud services and their configuration options can vary greatly across providers and can be subject to complex constraints. Handling this variability is essential to ensuring the correct configuration of cloud providers.

RQ_1: Is it possible to use feature models to handle the variability in the configuration of cloud environments?

- Do current feature modeling approaches properly handle the variability of cloud environments when considering service-based applications in which multiple application services may be deployed together?
- How can feature modeling be extended to properly handle the variability in cloud environments?

During our investigation, we identified that existing feature modeling constructs were insufficient to properly model the variability that arises in the configuration of cloud

providers. This limitation motivated us to introduce the concept of *relative cardinalities*. This concept can be regarded as a generalization of the possible interpretations for cardinalities in feature models. We then analyzed the impact of relative cardinalities on the syntax and semantics of feature models and how to reason over them.

When adapting a running cloud environment, in addition to the configuration variability, extra constraints governing their reconfiguration must also be considered. The configuration options available during a reconfiguration are not necessarily the same as those available for an initial setup and may depend on previous configuration choices. This implies a change in variability over time that also needs to be taken into account in order to generate correct configurations. In addition, cloud providers may provide alternative ways to reconfigure a cloud environment and still achieve the same final configuration, each one of them with different impacts on costs, performance, quality of service, among other properties, of the running cloud environment.

RQ_2: Is it feasible to use temporal properties for handling reconfiguration constraints in feature models?

- Is it possible to model reconfiguration constraints that arise during adaptation of cloud environments as temporal properties?
- How can temporal properties analysis be integrated with feature model analysis?

To answer this question, we proposed an approach that integrates temporal properties and reconfiguration operations with variability modeling to better define the adaptive behavior in DSPLs.

In order to setup or adapt a multi-cloud environment, we need means for defining its requirements and mapping them to cloud services made available by heterogeneous providers. In addition, we need to ensure that all configuration and reconfiguration constraints of the selected providers are followed and that we generate correct configurations.

RQ_3: How to generate a proper multi-cloud environment configuration?

- How to describe multi-cloud environment requirements?
- How to handle the heterogeneity across cloud providers offerings?
- How to select cloud providers for setting up a multi-cloud environment?

To generate multi-cloud environment configurations, we integrate variability modeling with semantic reasoning to get from a description of application needs to a selection of cloud providers supporting these needs and proper configurations for these providers.

Our research addresses these questions in a flexible approach that, starting from a description of the application's needs, obtains valid multi-cloud configurations that take into consideration the extensive variability and heterogeneity of cloud providers. We also conduct experimental evaluations to assess the feasibility of our approach. We describe our contributions in more detail in the next subsection.

1.3 Contributions

As stated in Section 1.2, the goal of this work is to facilitate the configuration and maintenance of multi-cloud environments that are tailored to their applications. In particular, we are interested in the challenges related to the heterogeneity and variability of cloud providers. Thus, the contributions of this thesis are aimed at supporting the management of variability in cloud providers and integrating it in an approach to attain the overall goals. Our first two contributions provide extended mechanisms for variability management, while the third contribution is an approach that leverages these results to support the setup and adaptation of multi-cloud environments.

Feature Models and Relative Cardinalities

The first contribution of this thesis is an approach for modeling feature cardinalities that allows the scope of cardinalities to be specified more powerfully. This work is motivated by the limitations identified in the existing feature modeling constructs to manage variability in cloud computing environments. To support modeling variability in the cloud, we introduce the concept of *relative cardinalities* and analyze their impact on cardinality consistency and cross-tree constraints. In addition, we provide a metamodel and a domain-specific language to support modeling feature models extended with relative cardinalities, as well as a method for automating the analysis of such models.

Feature Models and Temporal Constraints

In this second contribution, we combine temporal constraints with variability models to improve the definition of adaptive behavior in Dynamic Software Product Lines. This work is motivated by the complex reconfiguration constraints that arise when adapting cloud environments. We integrate temporal constraints and reconfiguration operations into the definition of a DSPL to support expressing reconfiguration constraints. We

can then reason over the constraints to find valid adaptations when a context change is identified. Additionally, we integrate these concepts into the modeling and reasoning tools developed in the preceding contribution.

Automated Setup and Adaptation of Multi-Cloud Environments

The final contribution is an approach to support an initial setup or an adaptation of a multi-cloud environment. It provides a high-level modeling language to describe multi-cloud environment requirements that supports a service-based application model. Cloud developers can use this language to describe the functionality and properties their applications require from cloud providers. A reasoning mechanism allows for automatically selecting cloud providers that support these requirements and generating a configuration or reconfiguration plan. This mechanism relies on semantic reasoning to handle the heterogeneity across cloud providers and map high-level requirements to provider-specific services and resources. Automated analysis of extended feature models, provided by preceding contributions, is used to ensure the derivation of correct configurations.

In addition to the contributions listed above, we have conducted an experimental evaluation to assess the feasibility and the performance of our approach. The experiments are aimed at understanding the impact of the proposed variability modeling constructs on the performance of variability analysis operations as well as the feasibility of the approach for the setup and adaptation of multi-cloud environments.

1.4 Publications

The results of this work were disseminated through a number of research papers presented in peer-reviewed conferences and published in their respective proceedings.

- G. Sousa, W. Rudametkin, and L. Duchien. Automated setup of multi-cloud environments for microservices applications. In *Proceedings of the 9th International Conference on Cloud Computing*, CLOUD'16, pages 327–334, June 2016. doi: 10.1109/CLOUD.2016.0051. URL http://doi.org/10.1109/CLOUD.2016.0051

- Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. Extending dynamic software product lines with temporal constraints. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '17, pages 129–139, May 2017. doi: 10.1109/SEAMS.2017.6. URL http://doi.org/10.1109/SEAMS.2017.6
- Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. Extending feature models with relative cardinalities. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 79–88, September 2016. doi: 10.1145/2934466.2934475. URL http://doi.acm.org/10.1145/2934466.2934475

1.5 Dissertation Outline

The remainder of this dissertation is composed of 7 chapters that are organized into four parts as follows:

Part I: State of the Art

Chapter 2: Configuration and Management of Multi-Cloud Systems. This chapter introduces the main concepts of Cloud Computing and a survey of the most relevant works on the configuration and management of multi-cloud systems. We classify these approaches according to the level of support they provide for managing heterogeneity and variability in cloud providers. Besides this, we also present an overview on approaches for managing variability in cloud systems.

Chapter 3: Software Product Lines. In this chapter, we introduce the main concepts from Software Product Lines Engineering, Feature Models and Dynamic Software Product Lines. The goal is to provide enough background on the principles and techniques that are employed by the proposed approach and the terminology that will be used later throughout this dissertation.

Part II: Contributions

Chapter 4: Feature Models and Relative Cardinalities. This chapter presents an approach for extending feature models with relative cardinalities to support modeling variability in the configuration of cloud providers. We define the concept of relative cardinalities

and demonstrate how automated analysis methods can be updated to handle them. This chapter is a revised version of the following paper:

Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. Extending feature models with relative cardinalities. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 79–88, September 2016. doi: 10.1145/2934466.2934475. URL http://doi.acm.org/10.1145/2934466.2934475

Chapter 5: Feature Models and Temporal Constraints. In this chapter, we present an approach for defining variability management with temporal constraints in order to capture the dynamic variability that may arise when adapting a running cloud environment. As in the case of relative cardinalities, we define how temporal constraints can be modeled and analyzed. This chapter is a revised version of the following paper:

Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. Extending dynamic software product lines with temporal constraints. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '17, pages 129–139, May 2017. doi: 10.1109/SEAMS.2017.6. URL http://doi.org/10.1109/SEAMS.2017.6

Chapter 6: Automated Setup and Adaptation of Multi-Cloud Environments. This chapter presents how the previous contributions are integrated in an approach to automate the setup and adaptation of multi-cloud environments. It presents the main elements of the proposed approach and how they deal with the issues identified in Section 1.2. This chapter is partly based on the following paper:

G. Sousa, W. Rudametkin, and L. Duchien. Automated setup of multi-cloud environments for microservices applications. In *Proceedings of the 9th International Conference on Cloud Computing*, CLOUD'16, pages 327–334, June 2016. doi: 10.1109/CLOUD.2016.0051. URL http://doi.org/10.1109/CLOUD.2016.0051

Part III: Evaluation

Chapter 7: Evaluation

This chapter describes implementation details of the tools developed for supporting the proposed approach and reports on the results of the experimental evaluation conducted to assess feasibility and performance.

Part IV: Final Remarks

Chapter 8: Conclusions

This chapter summarizes this thesis and discusses its contributions, impact, limitations and perspectives.

Part I

State of the Art

Chapter 2

Configuration and Management of Multi-Cloud Systems

In this chapter we introduce the main concepts of cloud computing and provide a survey of works related to the configuration and management of multi-cloud systems. In addition, we also discuss related works on variability management in cloud computing systems.

In Section 2.1, we introduce the principles of *cloud computing*, including its properties, service and deployment models. Section 2.2 presents the main concepts of *multi-cloud computing* and surveys the existing approaches for configuration or management of multi-cloud systems. In Section 2.3, we survey works that employ variability management techniques to deal with problems related to cloud computing systems. Section 2.4 summarizes the chapter.

2.1 Cloud Computing

The cloud computing paradigm is characterized by a model in which computing resources can be provisioned or release on-demand in a *pay-as-you-go* model. The ability to pay for computing resource based on its usage eliminates the need for upfront investments on computing infrastructure, turning it into an utility service like water, gas, electricity and telecommunication. Because of this, cloud computing can also be regarded as highly flexible computing infrastructure.

Since it is early beginnings, several definitions have been proposed for the cloud computing paradigm. These definitions often highlight different aspects that are considered as the

main properties of cloud computing such as resource sharing, virtualization, the *pay-as-you-go* model, elasticity and the illusion of infinite resources, etc.

For instance, Armbrust et al. refers to the illusion of infinite resources and the *pay-as-you-go* model as the main aspects of the cloud computing paradigm [2]. This conception is closely linked to the notion of computing as an utility service such as water, gas, electricity and communications.

Meanwhile, in [35], Buyya et al. define a cloud as a distributed system of virtualized computers that can be provisioned as resources to consumers, depicting cloud computing as an evolution of distributed computing models such as cluster and grid computing. This definition is clearly more focused on the implementation aspects of cloud computing and its key enabling technologies.

Despite differences on their focus, definitions of cloud computing are consensual when it comes the ability to dynamically provision or release resources on-demand. According to the United States National Institute of Standards and Technology (NIST), "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" [1]. The NIST also defined a set of cloud computing essential characteristics and a taxonomy for cloud service and deployment models. This taxonomy is employed in the following subsections to describe the main aspects of the cloud computing model.

2.1.1 Essential Characteristics

The NIST definition of Cloud Computing [1] identifies five essential characteristics of cloud computing:

On-demand self-service resource provisioning refers to the ability to provision or release computing resources such as computing power, storage and networking without the need of human interaction between the cloud customers and providers. This implies that cloud providers should offer means for managing computing resources through programmable interfaces.

Broad network access implies that cloud computing resources are made available through standard networking protocols and can be accessed through the Internet.

Resource pooling. Resources are provisioned to cloud consumers from a pool of shared resources managed by the cloud provider. The consumer has no control over the exact location of the resources and it is up to the provider to decide in which virtual or physical machine it is better to provision customers resources. Still, consumers can specify higher-level location constraints such as the data center or availability zone in which the resources are to be provisioned.

Elasticity refers to the ability to rapidly provision or release resources to scale the system according to consumer demand.

Measured services. Resource utilization is monitored, controlled and recorded to inform both providers and consumers about the resource utilization. This information can be used by providers and consumers to optimize their resource usage and allocation strategy and to charge users according to their utilization of resources.

2.1.2 Service Models

Computing resources in cloud computing are delivered as services at varying levels of abstraction that define the degree of control that consumers have over computational resources. Service delivery is often categorized into three different models:

Infrastructure as a Service (IaaS). Infrastructure services provide direct access for consumers to low-level computing resources such as storage, networking and computing nodes, often provided as virtual machines. Consumers do not have direct access to the physical machines and may have limited control over networking components. Still, they have complete control over all software stack from the operating system level up to the application level.

Platform as a Service (PaaS). Platform services provide a runtime environment for directly deploying applications. These services are often specialized and conceived to support a given programming language, platform, technology or ecosystem. A platform will often provide a runtime environment such as an application server together with a set of libraries and associated tools. Consumers do not need to manage or have limited control over the underlying infrastructure such as cloud servers, networking and storage. Additional services that are used in the implementation of applications, such as database management systems, are also often included in this category.

Software as a Service (SaaS). Software services provide application functionality to cloud consumers. Applications are made available through a user interface such as a web page

or a thin client; or by means of an application programming interface. Consumers do not manage or control infrastructure resources or development platforms and only have access to application specific configuration options.

Though this taxonomy of service models provides a useful classification for cloud services, there is not a hard line that separates service models, and some cloud services may exhibit properties of more than one service model. For example, some database services do not provide means for managing their resources and consumers are charged in terms of application specific concepts such as the number of rows stored and database queries rather than the underlying resource usage. These properties would qualify these services as a SaaS offering. Still, these services are mostly used by developers to store their application data and could also be regarded as platform services.

Besides this, cloud providers may deliver services at multiple service models and higher-level cloud services may be built on the top of lower-level cloud services. In addition to these three basic service models, we can find throughout the cloud computing landscape a large number of other models such as Database as a Service (DaaS), Storage as a Service (SaaS), Network as a Service (NaaS), Analytics as a Service (AaaS), etc. These service models, which are often grouped under the umbrella term of Everything as a Service (XaaS), can be considered as specializations of the three previously described service models.

2.1.3 Deployment Models

The NIST definition [1] of cloud computing also specifies four deployment models. The deployment models determinate how the cloud infrastructure is operated and which consumers have access to it.

In a *private cloud*, the infrastructure is provisioned to be exclusively used by a single organization. It can be directly managed by the consumer organization or by a third-party.

In a *community cloud*, many organizations may join to provision the infrastructure of a cloud that will be used exclusively by the organizations in the community. As in the case of private clouds it can be managed directly by members of the community or by a third-party provider.

A *public cloud* is one in which the cloud resources are available for the general public over the Internet upon the payment of the service fees.

A *hybrid cloud* integrates two or more clouds in any of the other models in a way that supports data and application portability across the consistuent clouds.

2.2 Configuration and Management of Multi-Cloud Systems

Since the advent of cloud computing, several concerns have been raised about the dependence of cloud consumers on providers and the security of cloud computing environments. These concerns led to a profusion of approaches that propose different means for consuming resources from multiple clouds such as hybrid clouds, federation of clouds, inter-cloud computing, multi-cloud computing, etc. In this section, we present an overview of the aspects involved in the consumption of resources from multiple clouds and a survey of existing approaches for multi-cloud computing.

2.2.1 Consuming Resources from Multiple Clouds

Previous works in the literature [7–9] have been dedicated to the study and classification of approaches for the use of multiple clouds. Here, we give an overview of these classifications about the motivation for using multiple clouds and the approaches for consuming resources from multiple clouds.

Motivation for using multiple clouds

In [8], Petcu presents a list of the top 10 reasons for consuming resources from multiple clouds. In [7], authors summarizes the benefits of using multiple clouds into 3 categories, while Toosi et al. [9] identifies 6 key benefits that capture the essential motivations for cloud interoperability. We classify these identified motivations, listed in Table 2.1, into three broad categories:

Reducing vendor dependence. This includes the uses of multiple clouds to avoid lock-in to a provider specific technology and to implement mechanisms for redundancy and disaster recovery thus improving the system resilience to cloud failures.

Complying with constraints, requirements and service offerings. Multiple clouds may be used to comply with legal issues, regulations and internal policies that require some

services or data to be deployed in data centers located in a given region or within a private cloud. They can also be used when a single cloud does not provides all the services required by the system.

Improving performance, quality of service or resource utilization. As different providers may offer different advantages in using specific features, combining them can be useful to improve quality of services or optimize costs. Distributing services closer to clouds that are closer to their end users may be used to reduce the overall latency. Multiple clouds can also be used to offload processing to other clouds when dealing with request peaks.

Table 2.1 Summary of motivations for consuming resources from multiple clouds.

Category	Motivation
Reducing vendor dependence	Interoperability and Avoiding Vendor Lock-In [9] Availability and Disaster Recovery [9] Avoidance of vendor lock-in [7] Better application resilience [7] Avoid the dependence on only one external provider [8] Ensure backup-ups to deal with disasters or scheduled inactivity [8] Replicate applications/services consuming services from different Clouds to ensure their high availability [8]
Complying with constraints, requirements and service offerings	Legal Issues and Meeting Regulations [9] Diverse geographical locations [7] React to changes of the offers by the providers [8] Follow the constraints, like new locations or laws [8] Consume different services for their particularities not provided elsewhere [8]
Improving performance, quality of service or resource utilization	Geographic Distribution and Low-Latency Access [9] Scalability and Wider Resource Availability [9] Cost Efficiency and Saving Energy [9] Diverse geographical locations [7] Optimize costs or improve quality of services [8] React to changes of the offers by the providers [8] Deal with the peaks in service and resource requests using external ones, on demand basis [8] Act as intermediary [8] Enhance own Cloud resource and service offers, based on agreements with other providers [8]

Approaches for using multiple clouds

Petcu [8] analyzed the most frequently used terms for naming approaches based on the use of multiple clouds and identified three categories of approaches: *cloud federation*, *multi-cloud* and *inter-cloud*. This classification is based on the degree of cooperation among clouds and the way cloud consumers interact with the multiple clouds.

A *cloud federation* is characterized by the establishment of a formal agreement among cloud providers to enable subcontracting resources across different clouds. Federations are driven by provider needs to improve their resource usage and service offerings. The

federation is transparent for the cloud consumer that only needs to interact with a single cloud provider.

Meanwhile, in the *multi-cloud* approach, there is no collaboration between cloud providers and it is up to cloud consumers or a third-party to interact with cloud providers. Multi-cloud is mainly driven by cloud consumer needs to reduce dependence on providers, comply with constraints and improve their costs or quality of service.

Finally, the *inter-cloud* approach is defined by the presence of a cloud marketplace in which cloud providers can negotiate among themselves the subleasing of resources. In this case, collaboration is established dynamically, according to the interests of cloud providers and the current market conditions. Hence, an inter-cloud can also be seen as a dynamic opportunistic federation.

The same categorization is found in [7], but with both multi-cloud and federations being considered as types of inter-clouds. Meanwhile, in [9], approaches are classified into *provider-centric* and *client-centric* according to the party that takes the initiative in the integration of multiple clouds. Cloud federation is classified as a provider-centric approach while multi-cloud as a client-centric approach.

2.2.2 Approaches for Configuration and Management of Multi-Cloud Systems

As we saw above, several approaches have been proposed to support the consumption of resources from multiple clouds. In this work we analyze only the multi-cloud approaches, in which the integration is driven by cloud consumers. Besides this, we focus on approaches that deal with aspects related to the configuration and management of multi-cloud systems. During this analysis, we identify how these approaches deal with the selection of cloud providers and services and particularly how they deal with the heterogeneity and variability of cloud environments.

PaaSage

PaaSage [36–42] is a platform to support the design and deployment of multi-cloud applications based on model-driven engineering principles. In PaaSage, users design provider-independent deployment model that defines application components and their requirements. Components may include location, service level objectives, scalability

and optimization requirements. These requirements are matched against cloud provider offerings to obtain a provider-specific model. The CAMEL [42] modeling language allows for defining concepts such as locations, metrics, resource types and capabilities that can then be used both in providers and requirements definitions. Variability management is based on the Saloon approach [26].

MODAClouds

The goal of the MODAClouds [43, 37, 44, 45, 20] project was to provide methods and tools for the design and automatic deployment of multi-cloud applications. The MODAClouds project was developed alongside PaaSage and share some common contributions and similarities. It also employs a component-based model for applications and a separation between provider-independent and provider-specific models. Provider-independent models define an assignment from application components to cloud services while provider-specific models define the exact cloud services and providers. It also employs a modeling language, called CloudML [45], which supports definition of unifying concepts across provider offerings. MODAClouds distinguishes from PaaSage by providing an approach based on risk analysis to support cloud services selection and the use of mechanisms for performance prediction and optimization.

The project also includes a mechanism for runtime monitoring and adaptation based on Models@Run.time [28, 46, 13, 47, 48]. Variability in providers' configuration is not taken into account during the selection or configuration of cloud services.

mOSAIC

mOSAIC [49–56, 11, 57, 58, 8, 59] is an integrated platform for designing, developing, deploying and executing multi-cloud applications. This platform can be deployed over infrastructure services (IaaS) from multiple cloud providers by means of provider-specific adapters. A cloud brokering mechanism is available to verify if a provider meets the desire service level objecties [50, 56].

The platform implements a component-based event-driven programming model. Applications for the platform are developed according to this model, using the mOSAIC API, which is available for Java, Python, Node.js and Erlang. As components are managed by the mOSAIC platform they can be scaled up and down individually and the platform

provides means for migrating them across providers. As the mOSAIC platform is designed to be deployed over infrastructure services it does not support the consumption of platform or software systems. Services such as databases, message queues, application servers, key-value stores among others are provided as components by the platform, which manages their deployment and execution. A semantic engine based on ontologies is used to query for components and services that support application requirements [59].

Variability in the configuration of cloud providers is not managed. Deployment descriptors are rather low level and the developers must specify the providers where components are to be deployed. [58]

Cloud4SOA

Cloud4SOA [60–62] is an approach for integrating heterogeneous PaaS offerings across cloud providers that share the same technology. The goal of the approach is to support developers in overcoming the heterogeneity across multiple PaaS offerings and find providers that best support their needs. This is achieved by the definition of an unified API based on commonalities across PaaS offerings and a set of specialized adapters for provider-specific services. Ontology-based reasoning is also used for matching requirements to offerings. Cloud4SOA supports cloud service selection, monitoring, management and migration of applications.

Variability and constraints in the configuration of cloud providers are not considered and deployments must define the clouds to which the application parts are to be deployed. Additionally, the analyzed papers present little information on how application requirements can be defined.

soCloud

soCloud [63, 64] is a service-oriented component-based platform for managing portability, elasticity, provisioning and high availability across multiple clouds. Applications are designed using the Service Component Architecture (SCA) model and run on the FraSCAti execution environment, which is a Java implementation for SCA. The FraSCAti environment provides an unified platform for applications, independent of the underlying cloud services. It can be deployed both to infrastructure services such as virtual machines or to any Java platform service.

soCloud application descriptors can include constraints on the necessary computing resources, geographical location of components as well as elasticity and replication rules.

SeaClouds

The SeaClouds [12, 65–68] project was proposed to build a multi-cloud application management system based on cloud standards. It supports the distribution, monitoring and reconfiguration of application modules over heterogeneous cloud providers. To handle heterogeneity across providers, they propose the definition of an unified API and provider-specific adapters; and a software architecture that unifies both IaaS and PaaS offerings under a single interface. The presence of a reconfiguration mechanism is included in the proposal, but little information is given about its functioning. The publications only describe the overall architecture, which is inspired from autonomic computing control loop [69].

As in other approaches there is no reference to variability in provider configuration. Resources required by an application are defined as a topology according to the TOSCA standard [70].

Dohko

Dohko [71, 21] is an autonomic goal-oriented system for inter-cloud environments. The system relies on SPLE principles for implementing a resource discovery strategy. An abstract feature model is used to defined the general concepts available in the configuration of an infrastructure resource, providing an unified view across multiple providers. Meanwhile, concrete feature models define the variability in the configuration of provider-specific infrastructure resources.

Application descriptors, defined by the system users, define the requirements such as cost, memory size and CPU and the desired provider and clouds services. Based on this descriptor, the system generates a deployment descriptor defining a configuration (e.g. virtual machine instance type, data center, virtual machine image, networking, etc) that matches these requirements. Besides this, the Dohko includes a monitoring component on virtual machines that is capable of identifying a virtual machine failure and restart it.

Other initiatives

Liu et al. [72], proposes a multi-cloud management platform that intermediates the interaction between cloud consumers and infrastructure providers. Service catalogs from multiple providers are integrated into a federated catalog that can be queried to find services that meet requirements. A connection and adaptation module is used to encapsulate differences on management interfaces from different providers. Though variability modeling or management is not mentioned, they include a *federation tag tree* that describes simple rules on how cloud services can be configured.

Grozev and Buyya [7, 14] propose an approach for building interactive three-tier multi-cloud applications that allows for adaptive provisioning and workload balancing. The contributions are algorithms for cloud selection, load balancing and autoscaling as well as their evaluation. Only infrastructure resources are considered and neither heterogeneity nor variability is considered.

Roboconf [73, 74] is a cloud orchestrator for deploying multi-cloud systems. It provides an hierarchical DSL that enables developers to specify the relationship between application components and cloud resources (e.g., virtual machines, storage, networking, etc). From an application descriptor it can automate the provisioning of cloud resources and deployment of the application components. In addition it provides some level of support for adaptation to support elasticity and self-recovery from service failures. However, it does not handle heterogeneity nor variability across cloud providers and developers must specify which cloud resources and providers will be used.

SALSA [75] is a framework for automatic configuration of complex cloud services on multi-cloud environments. It enables to specify a topology defining the relationship between application and cloud services such as application servers, virtual machines, execution containers among others. This approach is somewhat similar to Roboconf, in which the developer explicitly defines which kinds of cloud resources and services will be employed. However, topologies in SALSA are described using the TOSCA standard [70]. There is no mention to the variability in cloud provider configurations. However, the approach considers that each cloud service can offer different *configuration capabilities*, operations that are available to modify the configuration of a resource such as increasing the amount of memory or adding more instances of a virtual machine, installing a new software package, etc. These configuration capabilities can be provider-specific and a plugin mechanism is provided for implementing provider-specific configurators.

Besides these, there are several papers that propose or evaluate algorithms for the placement of virtual machines [22, 76–80] or storage [81, 82] across multiple clouds to optimize costs, energy consumption, quality of service or performance. They are mostly algorithmic-centric and focused on optimizing the use of infrastructure resources. Little or no attention is given to the heterogeneity or variability in cloud providers offerings.

2.2.3 Discussion

After an overview of the main approaches for configuration and management of multi-cloud systems we analyze how these approaches handle three important aspects of multi-cloud systems. Firstly, we discuss the *modeling abstractions* employed, which include the notation and concepts used in the description of application requirements or deployment model. Secondly, we analyze the strategies employed by each of these approaches to handle the *heterogeneity* across cloud providers. Finally, we evaluate to which extent these approaches handle the *variability* in the configuration of cloud providers.

Modeling abstraction

In order to support management of multi-cloud systems the approaches are expected to provide means for describing properties of the multi-cloud system using provider-agnostic notation and concepts. Here we analyze the modeling abstractions used by the surveyed approaches.

Notation. All the studied approaches employ provider-independent modeling notations. The PaaSage, MODAClouds and Cloud4SOA provide a set of DSMLs for defining multiple aspects of the cloud system. Meanwhile, the SeaClouds, Dohko and Uni4Cloud employ markup languages such as YAML and XML. Both soCloud and mOSAIC employ a component-based model and also rely on XML files for component configuration.

Modeling concepts. Both PaaSage and MODAClouds employ a component-based model for defining how cloud services and application components interact. In these approaches, users define a deployment model, which is based on *components*, *communications* and *hostings*. Together, these three constructs allow to define communications between application components and which application components are hosted by a given cloud component. This requires cloud customers to define at the design phase the type and location of virtual machines required by their application; and how their application

components will be assigned across these virtual machines. Both approaches include mechanisms for defining scalability rules.

Likewise, in the mOSAIC approach, application descriptors are used to define the relationship between application components and the resource types they require. Service level objectives and scalability rules can be specified at the component level.

soCloud supports the definition of component-based applications according to the Service Component Architecture. The components can be annotated with extra constraints on the location and replication of components as well as their scalability rules.

SeaClouds' abstract deployable profile includes information about cloud services or resources required by application services as well as replication and scalability rules.

Dohko application descriptor defines the clouds that must be used by the system and the requirements for the virtual machines to be provisioned on this cloud (e.g. memory, CPU, cost, operating system, etc). Besides this, it includes a list of applications to be executed.

Application model. Most of the approaches employ an component or service-based application model and do not target any specific application class. mOSAIC and soCloud impose the use of specific programming model and APIs, which implies that application services must be developed to run on these platforms. The component model employed Dohko [71] is targeted for high-performance applications and provides a model in which application tasks are submitted to job queue to be run over multiple clouds.

Service model. Cloud4SOA is the only platform focused on building multi-cloud systems based exclusively on PaaS offerings. SeaClouds supports both infrastructure and platform services. In the MODAClouds approach, infrastructure services can be defined as hostings and platform services can be modeled as external components that can host or communicate with other application components. soCloud provides a platform based on Java technology and thus can be deployed to any Java platform service or directly to virtual machines. In the other approaches, only IaaS offerings are taken into account.

Heterogeneity management

One of the challenges for building multi-cloud systems is to overcome the heterogeneity that exists across cloud offerings from different providers.

The SeaClouds and Cloud4SOA projects propose the definition of "harmonized APIs" based on the common concepts across different providers offerings and a set of adapters that map general concepts and operations to provider-specific ones.

MODAClouds and PaaSage deals with the heterogeneity by separating concerns into cloud provider independent and specific models. The cloud provider independent model allows to define requirements for cloud services and associate them to components in the application descriptor. These requirements are then mapped to actual offerings in a cloud-provider specific model.

soCloud deals with heterogeneity by providing an unified platform that shields applications from the underlying cloud resources. Nevertheless, it does not handle the heterogeneity that may exists across Java platform offerings or infrastructure services.

Similar to soCloud, mOSAIC also provides a platform that can be deployed over infrastructure services. In addition, it also employs semantic reasoning for managing heterogeneity in the service level agreements across infrastructure services. Likewise, Cloud4SOA employs Semantic Web technologies to overcome differences across cloud services and match application requirements to PaaS offerings.

Variability management

PaaSage and Dohko are the only projects that provides some level of variability management. In PaaSage, variability is modeled as a feature model extended with attributes that define the configuration options for infrastructure services such as a virtual machine. These attributes can then be used in a cloud provider specific model to establish a mapping between requirements and provider attributes. Dohko employs cardinality-based feature models to capture variability in infrastructure resources. Both approaches do not give much information about the feature modeling semantics that is adopted, just mentioning the use of constraint programming to evaluate feature models and either select or configure cloud services.

Table 2.2 summarizes the analysis of the discussed approaches, classifying them according to the aspects discussed above.

Table 2.2 Summary of approaches for multi-cloud configuration and management.

Approach	Modeling abstraction	Heterogeneity Management	Variability Management
PaaSage [36–42]	component-based	Model-Driven Architecture	Feature Model with attributes
MODAClouds [20]	component-based	Model-Driven Architecture	-
mOSAIC [49–56, 11, 57, 58, 8, 59]	component-based	unified API and adapters with semantic brokering	-
Cloud4SOA [60–62]	-	unified API and adapters with semantic brokering	-
soCloud [63, 64]	component-based	unified API	-
SeaClouds [12, 65–68]	service-based	unified API and adapters	-
Dohko [71, 21]	resource-based	Abstract Feature Model	Cardinality-Based Feature Model

2.3 Variability Management in Cloud Computing

After analyzing existing work on the configuration and management of multi-cloud systems, we focus on the variability management aspect. In this section, we survey existing work on managing variability in cloud systems. This analysis is not limited to multi-cloud systems and includes different usages of variability in cloud systems. We organize existing approaches into three categories as follows.

2.3.1 Variability in Cloud Applications

A category of works for variability management in the cloud is focused on the problem of building product lines of cloud applications. In a cloud application product line, besides the variability across applications it is also important to consider the variability on the cloud computing services that are going to be used by applications. This means that product line customers can also choose which cloud providers or cloud services will be used by their applications. These choices can impact the performance, costs or even the available features for the application.

Cavalcante et al. [83] propose an approach for developing a product line of cloud applications that can be configurable by selecting which cloud services should be used. A feature model is used to define the possible cloud service choices and based on cost estimations of cloud services the approach can estimate the cost of generated products.

In [84], the authors extended their approach to support the dynamic reconfiguration of cloud applications.

This category also includes approaches that have been proposed to support the development of multi-tenant SaaS applications [85–89], which serve multiple clients from a common code base. SPLE is employed to facilitate the development of customized tenants, tailored to customer needs. These approaches distinguish between the *functional variability* that defines variability in the application functionality and the *deployment variability* that defines the available options for deploying an application. Jamshidi and Pahl [90] propose a similar approach, but includes an extra *accessibility variability* model to model differences across multiple devices such as mobile phones, computers, cars, embedded objects, etc. In [91], authors propose to manage both variabilities to build adaptive cloud applications that can adapt both their functionality and their cloud environment.

The common aspects across these approaches is that their main focus is on managing the variability in a cloud application or across cloud applications in a product line or multi-tenant system. Even though they consider the variability in cloud environments, it is only the variability on the predefined ways in which the application was designed to consume resources.

2.3.2 Variability in Infrastructure Resources Configuration

Another range of approaches have been proposed to manage the variability in the configuration of infrastructure resources or services. For instance, [92, 93] use variability models to define how software packages (e.g. operating system, databases, application servers, frameworks, etc) can be combined to build a Virtual Machine Image (VMI) that can be later used to instantiate a virtual machine. Meanwhile, other works [25, 71, 94, 23] also manage the variability in the configuration of infrastructure services such as virtual machine size, data center location, storage type, etc.

Different from the previous category, in these approaches the goal is not build a product-line of cloud applications, but to manage the variability in the way infrastructure resources can be configured. These works employ variability management to achieve different goals such as optimizing power consumption and performance by finding optimal configurations; or simplifying the provisioning of resources and deployment of applications to the cloud.

2.3.3 Variability in Cloud Services Configuration

In this third category of approaches [24, 26], variability management is employed to select and configure cloud services or providers that support some requirements and generate the corresponding configurations. Feature models are used to capture the variability in the configuration of cloud services or providers, while requirements are defined using provider-independent concepts. Requirements are evaluated against feature models for different cloud services or providers in order to find complying services or providers and generate a configuration. While in [24] the variability is defined on a per-service basis, in [26] variability defines the options available in a cloud provider for deploying an application.

2.3.4 Discussion

Among the analyzed approaches, those in the last category are the most closely related to our work. In these approaches variability management is used to support the selection and configuration of cloud services or providers. Still, besides not supporting multi-cloud application deployments, these approaches manage variability in a limited context. For instance, in [24] feature models only capture the variability in the configuration of a single cloud service. Meanwhile, in [26] feature models are conceived in a way that only supports the configuration of cloud environments for tiered applications. As these approaches do not handle variability in a wider scope, they fail to capture any constraints that may arise once we are interested in deploying a service-based application, in which multiple cloud services must be setup in order to support the deployment and execution of application services.

2.4 Summary

In this chapter, we presented an overview of cloud computing concepts and a survey of state-of-the-art approaches for the configuration and management of multi-cloud systems. In particular, we analyzed how these approaches handle heterogeneity and variability across multiple cloud providers. We identified that few approaches provide semantic management of heterogeneity across service offerings and no approach provides appropriate management of variability regarding the configuration of cloud providers.

We also analyzed approaches to manage variability in cloud systems. We found that approaches for managing variability in cloud providers' configurations only handle it in a limited context. By managing variability in the context of a single service they neglect constraints that may exists between them.

In the next chapter, we introduce the main concepts from Software Product Line Engineering, Feature Models and Dynamic Software Product Lines, which constitute the foundation for our proposed approach for managing variability in cloud environments.

Chapter 3

Software Product Lines

In this chapter, we present a brief introduction to the main concepts of Software Product Line Engineering, Feature Modeling and Dynamic Software Product Lines. The goal is to provide the reader with enough background on the research fields directly related to this work.

This chapter is structured as follows. In Section 3.1, we present a brief introduction to Software Product Lines Engineering. In Section 3.2, we present principles and notations for feature models as well as an overview of approaches for extended feature modeling. In Section 3.3, we describe the main concepts of Dynamic Software Product Lines and their use on adaptive systems. Section 3.4 presents a summary of the chapter.

3.1 Software Product Lines Engineering

Software Product Lines Engineering (SPLE) is a systematic approach for developing a family of related software systems at lower costs, in shorter time and with higher quality [27]. The objective is to exploit similarities among related software products and manage their differences to build customized software products in an efficient manner [95]. SPLE has the potential to substantially improve developers' productivity and software quality while reducing development costs and time to market [95].

Product lines have long been used in industry to achieve *mass customization*, which is the large-scale production of goods with enough variety and customization [96]. SPLE brings the notion of mass customization to software, providing means for efficiently producing

multiple similar, but customized software systems. In a software product line, software products are assembled from reusable assets instead of being developed from scratch.

SPLE is effective only when the upfront investment to develop reusable assets is outweighed by the costs of building and maintaining multiple software products. Unlike other opportunistic reuse approaches, development of software systems as a product line relies on proactively planning for reuse. This requires developing a set of reusable software assets, clearly defining how these assets can be assembled together to build a software product, and systematically reusing these assets to build software products.

The key properties of SPLE, which distinguish it from traditional single software system development and other software reuse approaches, are the explicit management of *variability* and the organization of the software development process in two phases: *domain engineering* and *application engineering*.

3.1.1 Variability

The variability of a software product line is expressed through the different ways in which the set of reusable assets can be assembled to build a software product. In other words, the variability defines what can vary across different software systems that are part of a product line. Variability in software product lines is usually defined in terms of *variation points* and *variants*. While a *variation point* defines a property that can vary across different products a *variant* defines one of the possible acceptable values for a given *variation point*.

In SPLE, variability has to be explicitly defined and managed so that it can be leveraged during the assembly of software products. Variability is defined during the domain engineering phase in which the variation points and variants are specified; and exploited during application engineering when variants are selected to assemble a software product. Variability management is a process that encompasses definition and formal documentation of variability, management of reusable assets and application of variability during program assembly.

Variability can be introduced at different levels of system abstraction such as requirements, architecture, services, components, code, tests, etc. Explicit definition and management of variability is used to restrict customization of software products to only those variation points that make sense in a given product line. This is the main aspect which distinguishes SPLE from other software reuse approaches [27].

3.1.2 Software Product Line Engineering Process

Software development in SPLE is organized around two complementary phases: *domain engineering* and *application engineering* [27, 95]. In the domain engineering phase, variability across software products is defined and common reusable assets are developed. In the application engineering phase, previously developed assets are assembled to build a customized product.

Domain engineering

The goal of this phase is to define a common platform and reusable *core assets* that will be later used in application engineering to assemble of software products. During domain engineering, software engineers should consider not only the current requirements of a single software product, but the requirements of many potential future products in the same domain. This requires the clear definition of the *scope* of the product line, which delimitates the set of potential software products for which it is planned. Once the scope is defined, similarities and differences across software products in a product line are identified and documented as variability models. Then, common reusable assets that accomplish the variability are defined.

Domain engineering is also called as development *for reuse* as it is focused on engineering reusable parts that will be later reused during application engineering. As in other software engineering processes, it is usually organized around many activities such as analysis, design, coding and testing.

Application engineering

The goal of this phase is to develop a customized product by exploiting the assets and variability defined during domain engineering. Customer requirements are identified and matched against the variability model in the product line to obtain a product configuration that binds variants to variation points. Then, this configuration can be used to derive a concrete software product by reusing core assets. The level of automation in product derivation depends directly on the employed supporting tools and the degree of formalization of the product line. Extra development may be required when core assets do not support all the desired requirements.

Application engineering is also called as development *with reuse* as products are derived from reusable assets in domain engineering phase.

SPLE is an industry proven approach that promotes systematic reuse to improve software quality and reduce both development costs and time to market. It is particularly effective when developing multiple similar software systems in a given domain. By combining variability management and reuse of common assets SPLE aims to support mass customization in software products.

3.2 Feature Models

Feature modeling is the most widely used technique for modeling variability in software product lines, being considered as the *de facto* standard for representing variability [97, 98]. Feature models were initially introduced by Kang et al. as part of the Feature Oriented Domain Analysis (FODA) method [99] to capture commonalities and variabilities on the requirements for systems in a given domain. Since then its usage has been extended to model variability at multiple levels of granularity and throughout all activities in the software development process such as architecture, design, implementation and tests [100, 98, 27, 101, 97, 102]. Feature models have also been employed beyond software product lines to model variability in highly-configurable or adaptive systems [103–107].

Feature models describe variability in terms of features, but the understanding of what is a feature can vary significantly across different feature modeling approaches and many definitions have been proposed. Features were initially defined by Kang et al. as "user-visible aspects or characteristics of a domain" [99]. Later, Czarnecki et al. defined a feature as "a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept" [108]. Meanwhile, for Batory, a feature can be regarded as "an increment of program functionality" [109]. While some authors emphasize on the visibility and relevance to an end-user or stakeholder, others have expressed a greater focus on the capacity of feature models for representing variability, regardless of the abstraction level.

From an strict variability perspective, a feature defines a variation point that can be either activated or deactivated. A feature model describes the relationship between features and how active features can be combined into a valid configuration. From a conceptual point of a view, a feature model defines a set of valid feature combinations, in

which each combination represents a configuration for the system or product line under study.

3.2.1 Feature Diagrams

A feature model defines variability in terms of features and their relationship. Feature models are often depicted as a *feature diagram*, in which features are organized in a tree-like structure. Throughout this dissertation, we use the terms feature models and feature diagrams interchangeably. However, the diagram is just a visual representation of a feature model that can be also described by other means. Figure 3.1 shows an example feature diagram for a cloud environment.

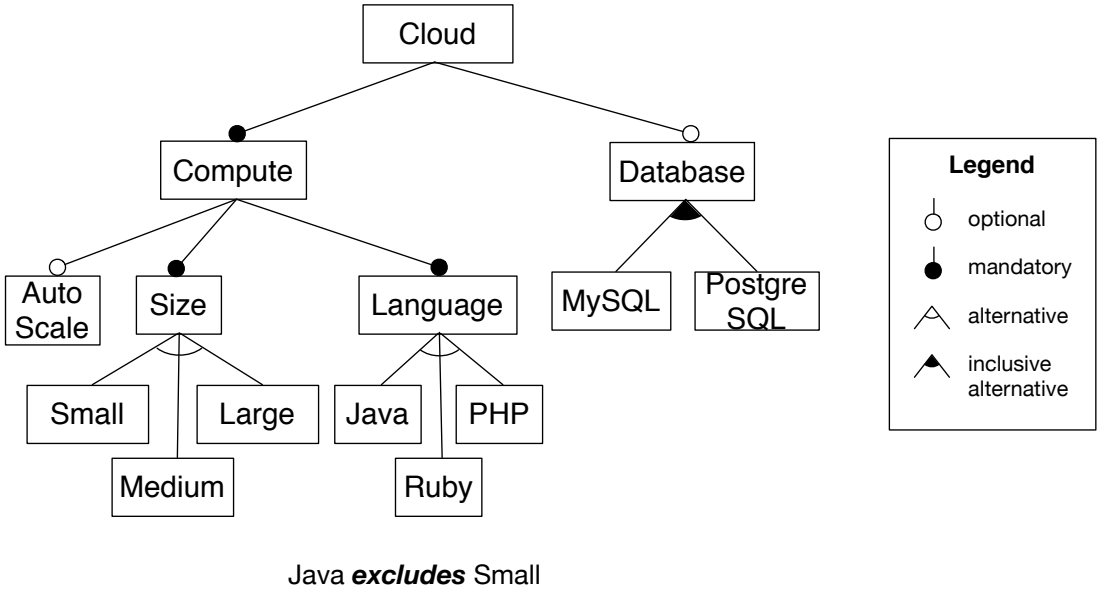


Figure 3.1 An example feature diagram.

Features are graphically represented as boxes organized in a hierarchical structure. While the hierarchy defines how features are decomposed into sub-features, extra graphical notation (see Figure 3.1 legend) characterizes the nature of the relationship as *optional*, *mandatory*, *alternative* or *inclusive alternative*. Additional dependencies or conflicts between features can be expressed using textual *requires* and *excludes* constraints.

The root feature (`Cloud` in our example), represents the system under study and should always be included (i.e. active) in any configuration for the feature model. A feature marked as mandatory (filled circle) should be included in a configuration whenever its parent feature is also included. In our example, feature `Compute` is mandatory and should

be included in every configuration since it is a sub-feature of the root feature `Cloud` which is always included as well. Optional features (empty circle) can be optionally included if their parent is also included in the configuration. In our example, feature `Database` feature is marked as optional. As a general rule, a feature can only be included in a configuration if its parent feature is also included.

The *alternative* and *inclusive alternative* relationship is defined across multiple sub-features and are also called as *feature groups*. For each *alternative* group, exactly one feature in the group must be included in the configuration. Meanwhile, *alternative inclusive* groups require only that at least one feature in the group is included. In our example, for the *alternative* feature group under the feature `Language` exactly one of the available languages (i.e. `Java`, `PHP` and `Ruby`) need to be selected. On the other hand, for the *alternative inclusive* group under feature `Database` it is possible to include any of the databases features available or both.

Finally the textual constraint *`Java` excludes `Small`* is used to define that feature `Java` requires either a `Medium` or `Big` compute node. These constraints are also called as *cross-tree* or *additional* constraints and are used to define extra relationships across features, which do not fit in the feature diagram hierarchy.

A feature model defines a set of valid configurations, each one with a different combination of features. A *feature model configuration* is defined by a selection of features to be included in the corresponding software product or system. A configuration is valid if it follows the composition rules defined by the feature model. Formal definitions of the syntax and semantics of feature models can be found in the literature [110–113].

3.2.2 Analysis Operations

As other modeling notations, feature models are used to express information in a way that is convenable for its analysis. Analysis of feature models is often defined in terms of *analysis operations* that are targeted at extracting some specific information about a feature model. A survey conducted by Benavides et al. identified 30 different operations proposed in the literature [114] that we organize in 4 broad categories:

- *Anomaly detection* includes operations whose goal is to find inconsistencies in the definition of the feature model or its corresponding product line. Examples of such operations include verifying if a *feature model is void* and leads to no valid

configuration, if there is a *dead feature* that cannot be included in any valid product, *false optional features* that are included in all configurations, etc.

- *Property extraction* operations are targeted at obtaining information about a feature model. This includes obtaining the set of *core features* that are part of every products as well as *variant features*. Finding *atomic sets*, *variability factor*, *homogeneity* and *commonality* of features are also examples of property extraction operations.
- *Feature model comparison* operations are used to identify the relationship between different feature models. Verifying if a feature model is a *refactoring*, a *generalization* or an *specialization* of another feature model are examples operations in this category. While the operations in previous categories take only one feature model as input and obtain an output, feature model comparison operations work on multiple feature models.
- We classify as *feature model configuration* operations those operations that take not only a feature model as input but also a feature model configuration. Example of such operations include the validation of *complete* or *partial configurations* as well as the generation of a full configuration from a partial one, also called as *auto completion*. These operations also distinguish from the previous ones in which they are most often applied during the application engineering phase.

Feature model analysis operations are usually implemented by mapping it to a representation in a well-defined formalism such as propositional logic, constraint programming and description logics, and relying on available off-the-shelf solvers. Most of the reasoning operations are computationally hard NP-complete problems [113]. Still, there is evidence that for most operations in realistic feature models the problem are tractable [115, 116].

3.2.3 Cardinality-Based Feature Models

Several extensions to feature modeling have been proposed since its inception, usually motivated from pragmatic needs in product line engineering. The concept of cardinalities has been initially introduced to feature models as an application of UML multiplicities to feature groups from the original FODA notation [117]. Feature cardinalities were introduced later to deal with scenarios where a feature can be selected multiple times for a given product, each time with a possibly different set of sub-features [118]. Czarnecki et al.

integrated feature model extensions such as attributes, group and feature cardinalities into the concept of cardinality-based feature models [119, 120].

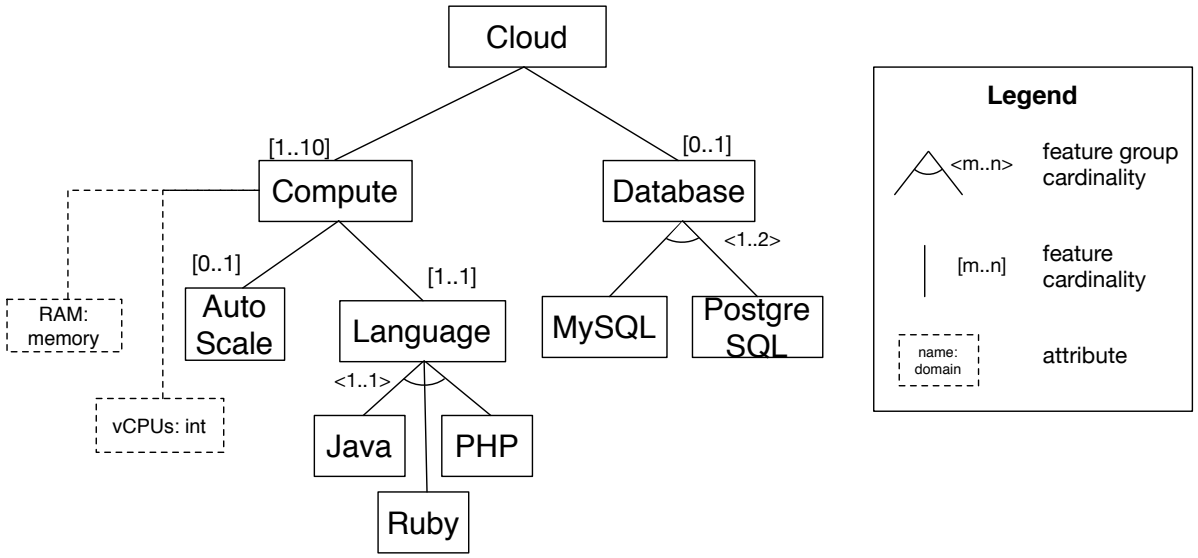


Figure 3.2 An example feature model with cardinalities and attributes.

Figure 3.2 depicts an example cardinality-based feature model. The main elements employed in this notation are *feature group cardinalities*, *feature cardinalities* and *attributes*.

Feature group cardinalities are used to define the minimum and maximum number of features that can be selected among the alternatives in a feature group. It can be seen as a generalization of the *alternative* feature groups found in boolean feature models. In our example, the group cardinality <1..1> under feature `Language` is equivalent to an alternative group as exactly one feature should be chosen among the three available options. On the other hand, group cardinality <1..2> under feature `Database` acts as an alternative inclusive group.

Feature cardinalities determine the number of times a feature can be included in a configuration. Feature cardinalities introduce instantiation to feature modeling as a feature can be included multiple times in a configuration, each time with a different decomposition of its sub-features. In our example, between 1 and 10 instances of feature `Compute` can be included in a configuration, each one with a different configuration for features `AutoScale`, `Language`, `Java`, `PHP` and `Ruby`. Feature cardinalities are also a generalization of *mandatory* and *optional* relationship, which can be represented by cardinalities [1..1] and [0..1] respectively as in the case of features `Database` and `Language`. In most approaches, feature cardinalities apply only to *solitary features*, i.e. those that are not part of a group.

Attributes are used to specify non boolean variability in a more concise representation. Instead of defining a feature group with a large number of alternatives an attribute can be used instead. Attributes often define an attribute *name* and a *domain* from which values can be chosen. In most approaches the domain is finite or constrained to a limited range or enumeration to simplify automated reasoning.

Table 3.1 Overview of approaches for modeling and reasoning on feature models with feature cardinalities and clones.

approach	cardinality range	clon.	level	cross-tree constraints	attrs.	operations	reasoning
Czarnecki et al. [119, 121, 120]	noncontiguous infinite	y	local	XPath	typed	staged config	CFG
Benavides et al. [122]	contiguous finite	n	global	on features	-	find config. and no. products.	CP
Ramírez et al. [123, 124]	contiguous finite	y	local	existential & universal quantifiers	-	check and generate minimal config.	Meta-model, OCL, CP
Gómez and Ramos [125]	contiguous finite	y	local	OCL-based	typed	check and generate minimal config.	Meta-model, OCL
Mazo et al. [126]	contiguous finite	y	local	on features	finite domain	11 ops: consistency check., no. products, check and generate config., etc	CP
Quinton et al. [127]	contiguous finite	n	local	on cardinalities	finite domain	check and generate config.	CP
Karataş et al. [128, 129]	contiguous finite	y	local	on instances	finite domain	12 ops: consistency check., no. products, check and generate config., core feat., commonalities, etc.	CP
Dhungana et al. [130]	contiguous infinite	y	local	-	-	check config., auto completion	GCSP
Schnabel et al. [131, 132]	contiguous infinite	y	local	-	-	card. consistency, check and generate config.	ILP, SMT, Alloy

XPath - XML Path Language
CFG - Context-Free Grammars
CP - Constraint Programming
OCL - Object Constraint Language
GCSP - Generative Constraint Satisfaction Problem
ILP - Integer Linear Programming
SMT - Satisfiability Modulo Theories

There is not an unified view on the syntax and semantics of cardinality-based feature models across the literature. While some works highlight that feature cardinalities define the number of allowed instances of a feature in relation to its parent feature, others just mention about the number of times a feature can be included in a configuration. In some approaches cardinality ranges may be unbounded (e.g. [2..*]) or non-contiguous (e.g. [1..1][3..4]) while others are limited to contiguous finite cardinality ranges. Table 3.1 presents an overview of existing approaches for modeling and reasoning over cardinality-based feature models.

Semantics of cross-tree constraints and the relationship between group and feature cardinalities are often not clearly defined and may also differ across different approaches. Michel et al. worked on the formalization of syntax and semantics of cardinality-based feature models and identified many of these ambiguities [133]. They proposed a classification of possible interpretations for group and feature cardinalities and identified the need for cross-tree constraints over feature instances and their cardinalities.

Quinton et al. proposed a feature modeling approach to support cross-tree constraints over feature cardinalities and attributes [127]. Meanwhile, Karataş et al. introduced support for complex constraints over attributes and feature instances [128].

Feature instances are also called as *clones* in the literature. Because of this, cardinality-based feature models are also named as *clone-enabled feature models*. Throughout this work, we use this terms interchangeably.

Feature models are widely used both in the industry and academia to manage variability in product lines and variability-intensive systems. Several extensions have been proposed over the years to improve their expressive power giving rise to a large number of notations. Likewise, several approaches for reasoning over feature models have been proposed to automatically extract different kinds of information from feature models with variable support for operations. The availability of methods for modeling and analysis of feature models is one of the main driving forces behind their success.

3.3 Dynamic Software Product Lines

Dynamic Software Product Lines (DSPLs) is a systematic approach for building reconfigurable or adaptive systems based on principles from SPLE [134]. The idea behind the DSPLs approach is to leverage proven engineering foundations from SPLE to support the development of such systems. However, while in a classical SPL, the goal is to build a family of related systems in a given domain, a DSPL is a system that can be dynamically adapted or reconfigured to many different, but related configurations. Table 3.2 compare the main properties of SPLs and DSPLs.

As in classical SPLs, variability management is the central principle of the DSPLs approach. However, variability in a DSPL defines the possible ways of configuring an adaptive system. This kind variability is often called as *runtime variability* or *dynamic*

Table 3.2 Relationship between SPLs and DSPLs [134].

Classic software product lines	Dynamic software product lines
Variability management describes different possible systems.	Variability management describes different adaptations of the system.
Reference architecture provides a common framework for a set of individual product architectures.	DSPL architecture is a single system architecture, which provides a basis for all possible adaptations of the system.
Business scoping identifies the common market for the set of products.	Adaptability scoping identifies the range of adaptation the DSPL supports.
Two-life-cycle approach describes two engineering life cycles, one for domain engineering and one for application engineering.	Two life cycles: the DSPL engineering life cycle aims at systematic development of the adaptive system, and the usage life cycle exploits adaptability in use.

variability [135] and refers to the possibilities for dynamically reconfiguring a system at runtime. As in the case of classical SPLs, automated analysis of variability models is used to identify anomalies during the design of the adaptive system and to ensure that reconfigurations are consistent and safe at runtime [136] .

In traditional SPLs, development is organized around domain and application engineering phases. During the domain engineering phase, the scope and the variability of the product line is defined while in the application engineering phase the variability is applied to generate products in the predefined scope. The DSPL approach also employs a similar organization. During the design of the adaptive system, the adaptation scope and variability are explicitly defined. At runtime, as the context changes, the system exploits the variability to perform safe adaptations. Because of this aspect, DSPLs are often used to build systems with *bounded adaptivity* [137], in which dynamic variations are foreseen at design time. Nevertheless, there are works in the literature that support *open adaptivity* [137] by dynamically evolving the variability at runtime [138, 139] .

3.3.1 Contextual Variability

Though DSPLs are not necessarily self-adaptive or context-aware systems [134] they are often used to build these kinds of systems. For such systems, there is a need to monitor and analyze the system's environment in order to identify changes that may require a system adaptation. Many DSPL approaches propose to manage the variability in contextual properties in order to reason about context changes and generate potential adaptation plans [140, 141, 106, 142].

The *contextual variability* [106], also called as *environmental variability* [143], of a system defines how the *environment* of the system changes at runtime. This term is often used to distinguish from the *structural variability* that defines how the *system* can adapt at

runtime. Both are considered as dimensions of the *dynamic variability* of an adaptive system [143].

During the system execution, both the environment and the system can change according to their respective variability. The relationship between contextual and structural variability determines the set of possible adaptations for the system given a context change and the current system state.

3.3.2 Variability Modeling in DSPLs

Most DSPL approaches employ variability modeling notations from SPLE to capture dynamic variability. Feature models and their derivations are the most used notation [136, 144–148, 140, 141, 149, 150, 91, 151], but orthogonal variability modeling [152] and constraint programming [153] have been employed as well.

Though structural variability is always explicitly modeled, contextual variability is sometimes handled using adhoc mechanisms. When contextual variability is explicitly modeled, it is often by the means of a context feature model, which is linked through constraints or mappings to the structural variability model [154, 144, 140, 141, 151]. Some approaches also add *context features* to feature models in order to capture in a single variability model both structural and contextual variability [106, 150, 149, 142].

Another concept that is often explored when managing variability in DSPLs is the *binding time* of variation points. The binding time defines at which moment in the system's lifecycle a given variation point can be bound or rebound to a variant. Binding times commonly used in existing approaches include design, implementation, compilation, build, assembly, configuration, deployment, start-up and runtime [155]. Adding binding time information is a way to delay decisions to later stages and define which parts of a system can be dynamically reconfigured at runtime. Multiple binding times can be used to build product lines of adaptive systems. In this case, some variation points are bound before the system startup, while other variation points can be dynamically rebound at runtime.

3.3.3 Dynamic Variability Mechanisms

The dynamic or runtime variability of a system is defined by the variation points that can be dynamically rebound during the system execution and the variants that can be assigned to these variation points. A system built as a DSPL, can be adapted by

rebinding its variation points to new variants. The dynamic variability of the system can be regarded as a higher-level abstraction on how the system can be adapted or reconfigured at runtime. However, there is still the matter on how to effectively modify the executing system when variation points are bound to a new variant.

Different approaches have been proposed for implementing dynamic binding of variation points [156, 137, 157]. The most simple way is to implement variation points directly at the application level. In this case, variation points can be rebound by changing parameters or configuration files. Despite its apparent simplicity, this approach limits the maintainability of the system as application and adaptation logic are interwoven and handled at the same abstraction level. It also limits the potential for adaptability as only explicitly previewed and coded variations are possible.

Other approaches rely on the adaptation mechanisms available in the underlying system platform to implement variation points. This is the case for component-based or service-based systems in which the architecture of the system can be modified by changing the interactions between system elements, or by integrating new elements into the system. In these cases, the range of possible adaptations and their granularity is defined by the adaptation mechanisms and reconfiguration capabilities of the underlying platform.

3.3.4 Adaptation in DSPLs

As for other software engineering approaches, there is not an unified view on how to build adaptive systems as DSPLs. Different approaches will often focus on different aspects of adaptive systems and employ different methods to deal with their issues. Though there is a consensus on the core element of the DSPLs approach, which is their heritage from SPLE and the management of runtime variability, these elements are not enough to build an adaptive system.

Still, most approaches for building self-adaptive systems through DSPLs organize system adaptation as a feedback loop [158] that includes at least three main functions that deal with concerns of *context management*, *adaptation planning* and *execution* [157].

Context management deals with issues related to monitoring the system context and identifying changes that may require a system adaptation. Not all approaches explicitly deal context management issues, but some works propose to explicitly model and manage the context variability [106, 150, 149, 142]. In these approaches, context features or variables are usually associated to monitored context events. In general, analysis of

context changes leads to *reconfiguration request* defining features or variants that should be activated or deactivated in the system.

The *adaptation planning* activity is responsible for generating a new configuration or *adaptation plan* that meets the requirements of the new system context. Most works on DSPLs are focused on this activity [137, 157]. It is in this phase that variability models are exploited and analyzed to generate an adaptation plan that ensures a safe reconfiguration or optimizes system goals. The resulting plan is based on a configuration for the variability model, in which the variation points were bound to variants.

Finally, in the *execution phase* the *adaptation plan* is mapped to a set of instructions to be executed in the underlying platform. For a component-based system this would include instructions for replacing components or changing connections, while for a parametrizable system this could be represented as parameter changes.

As in other approaches for building adaptive systems, the line between the activities in the feedback loop is not rigid, and according to the approach, some planning activities may be included as part of context management or execution and vice-versa.

Dynamic Software Product Lines is an approach based on SPLE principles for the development of reconfigurable systems. The core element of the DSPLs approach is management of dynamic variability, which determines how a system can be reconfigured at runtime. Due to their SPLE heritage, DSPLs usually employ the same variability modeling and analysis methods as their static counterparts. Adaptation in DSPLs is usually organized around a feedback loop, in which variability management plays the central role. However, adaptation capabilities of DSPLs are limited to the adaptation mechanisms provided by the underlying platform or application.

3.4 Summary

In this chapter, we presented a brief overview of the main concepts and terminology of software product lines and variability management that will be used throughout this dissertation.

First, we introduced the key principles of SPLE such as explicit variability modeling and the domain engineering process. Then, we presented an overview of feature modeling approaches, which are largely used to manage variability in product lines. Finally,

we introduced Dynamic Software Product Lines as an approach that relies on SPLE principles to support the development of adaptive systems.

In the next part of this dissertation, we describe the contributions of this work. These contributions are based on the principles of software product-line engineering and variability management presented in this chapter and are aimed to overcome the limitations identified in the state-of-the-art. Notably, they are focused on managing variability in the configuration of cloud environments with the goal of automating the setup and adaptation of multi-cloud environments.

In Chapters 4 and 5 we propose approaches for improving variability modeling and management in order to support configuration constraints that are commonly found in cloud environments and that are not supported by existing feature modeling constraints. First, in Chapter 4, we propose relative cardinalities, which are motivated by the need to manage cloud environments' variability in a wider scope. Then, in Chapter 5, we present an approach for integrating feature models with temporal constraints in order to capture the dynamic variability that may arise when adapting a running cloud environment. We define the syntax and semantics of these constructs and analyze how they integrate with commonly used feature modeling constructs. Besides this, we demonstrate how to extend the automated analysis of feature models to properly handle these constructs.

Finally, in Chapter 6, we describe an approach to automate the setup and adaptation of multi-cloud environments. This approach leverages our previous contributions on extended feature modeling to support the generation of correct configurations and reconfiguration plans for cloud environments. Besides this, it employs concepts from model-driven engineering and semantic reasoning to overcome heterogeneity across cloud providers.

Part II

Contributions

Chapter 4

Feature Models and Relative Cardinalities

Feature cardinalities were first introduced in [118] to deal with scenarios where a feature can be selected multiple times for a given product, each time with a possibly different decomposition of subfeatures. Unlike *group cardinalities* (see Section 3.2.3) that define the number of available options to be chosen in a feature group, *feature cardinalities* specify the number of times a feature can be included in a product configuration. When feature cardinalities are used, a product configuration is not only defined by a selection of features but also by feature instances.

Feature cardinalities are particularly useful when modeling variability in cloud computing environments as these are often composed of multiple instantiable services and resources. Besides this, cloud providers often exhibit limits on the number of resources and complex constraints between the selected services and the number of available resources. Due to these aspects, feature models with cardinalities have already been employed in related work to model variability in the cloud [26].

The semantics of cardinalities in feature models and their effects on cross-tree constraints have been thoroughly studied and formalized in different ways [120, 133, 127, 159, 160]. Feature cardinalities are usually considered to apply either locally (in relation to its parent feature) or globally (concerning the whole product configuration). However, through our investigation in managing variability in the configuration of cloud computing providers, we found that in many cases feature cardinalities may also be related to an ascendant feature that is not the direct parent feature. To deal with this limitation we introduce the concept of *relative cardinality*, which is a generalization of the existing interpretations

of feature cardinalities. A relative cardinality is the cardinality of a feature in relation to one of its ascendant features. In this chapter we redefine cardinality-based feature models to take relative cardinalities into account. We then analyze the effects on feature model semantics, including cardinalities consistency and cross-tree constraints. The contributions of this chapter are aimed at answering RQ_1 introduced in Section 1.2.

This chapter is structured as follows. Section 4.1 discusses the limitations found in feature cardinalities when modeling variability in cloud computing environments and the motivation for introducing relative cardinalities. Section 4.2 presents the concept of relative cardinality and discusses how it affects cardinality consistency and cross-tree constraints. Section 4.3 describes the implementation of a tool for automatic analysis of feature models with relative cardinalities. Section 4.5 summarizes this chapter.

4.1 Motivation

In the cloud computing paradigm, resources are provided as cloud services at varying levels of abstraction that can be selected by developers to set up an environment to deploy and execute their applications. Due to the wide range of available cloud providers and their large service offer, developers are often faced with difficult choices. In order to select a cloud provider and set up a cloud environment, developers need to be aware of the available services as well as their configuration options and constraints in order to create a suitable configuration. Managing this complexity with no tool support is a error-prone task that requires a lot of effort from developers [23, 26].

To support this activity, commonalities and variabilities in the configuration of a cloud provider can be captured as a feature model, thus making it amenable for automated analysis using a software product lines approach. This approach has been employed in recent work to support the automatic selection and configuration of a cloud provider for deploying an application [24, 26, 23, 21]. However, in previous work, variability is only considered in the context of a resource (e.g. virtual machine, storage, network) or for the deployment of a single monolithic application.

When variability is considered in a limited context, constraints at the provider level are ignored, which may lead to invalid configurations. However, once we try to express the variability in the configuration of cloud providers' we identified that existing feature modeling constructs were insufficient. In the remainder of this section we present an example that illustrates these limitations and the challenges to overcome them.

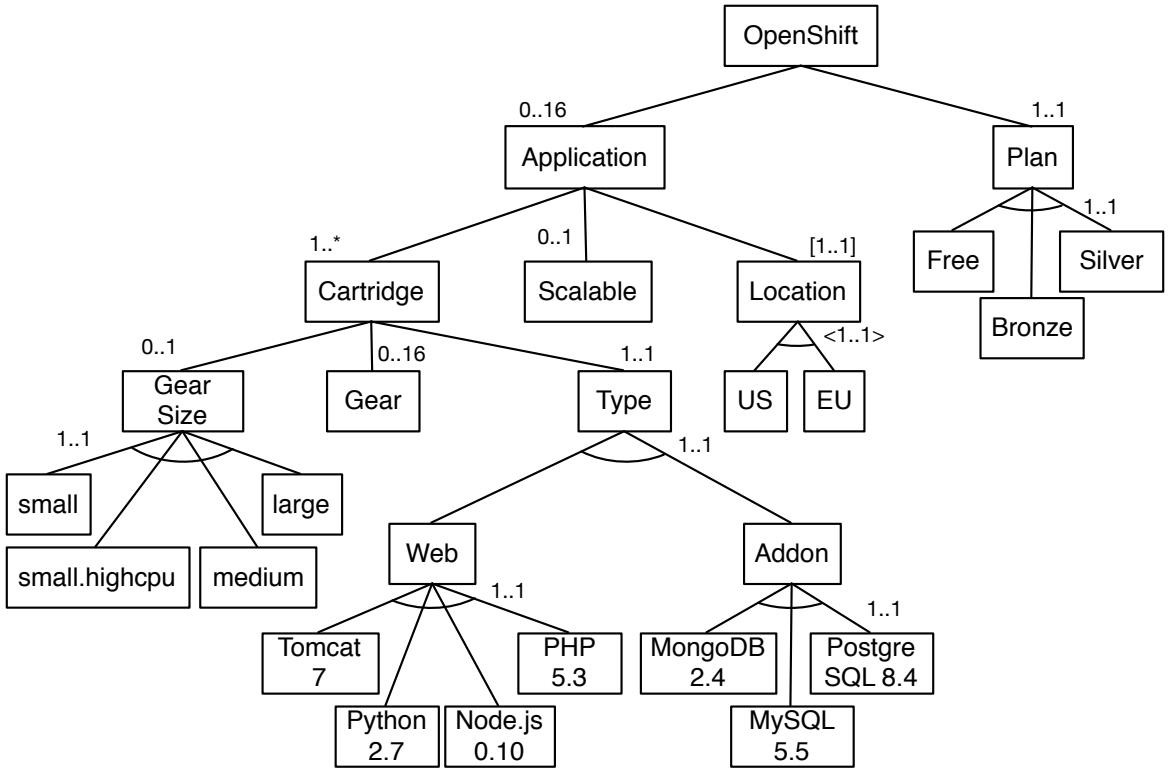


Figure 4.1 Excerpt from OpenShift cloud feature model.

4.1.1 Motivating Example

Figure 4.1 depicts an extract of a feature model we designed to capture variability in the OpenShift PaaS [161] provider. In OpenShift, an `Application` is composed of a set of `Cartridges`, which represent cloud services, resources or software features such as application servers, databases, caching services, management tools, etc. `Cartridges` are run by processing nodes called `Gears`. For each `Cartridge` the user can choose a number of `Gears`, as well as the `Gear Size`, which defines the memory and processing capabilities.

OpenShift imposes no limit on the number of `Applications` or `Cartridges` allowed. However, the number of `Gears` is limited by the user's plan. Considering that the limit for a plan is 16 `Gears`, we can have any combination ranging from 1 `Application` with 16 `Gears`, to 16 `Applications` with 1 `Gear` each. To enable users to describe both of these valid configurations, the feature model was designed in a way that the feature cardinalities of `Application` and `Gear` allow for up to 16 instances. Though this modeling allows for expressing all possible valid configurations, it does not prevent users from describing invalid ones. For instance, we could define a configuration with 16 `Applications`, each one with 16 `Gears`, thus exceeding the provider's limits.

A similar problem can be found in `Web` cartridges configuration. An `Application` can have multiple cartridges, but the provider requires that every `Application` should have exactly one of type `Web`. However, the feature model does not provide any information about this restriction, allowing for `Application`s with any number of `Web` cartridges.

This problem occurs because the features `Gear` and `Web` are directly associated to the `Cartridge` feature, but the number of allowed instances are respectively associated to the whole product configuration and to `Application` instances. This scenario occurs more often when we have features that consume computing resources but for which the maximum number of resources is associated to another feature, usually in a higher position in the feature diagram hierarchy. In these cases, the number of allowed instances of a given feature is not only related to its direct parent, but also to ascendant features in the hierarchy.

Across the literature on feature modeling, the work that most closely deals with similar issues was proposed by Quinton et al. in [127, 26]. In this work, authors propose extended cross-tree cardinality constraints to improve variability modeling in cloud environments. The extended notation supports constraints such as `Application` $\rightarrow +1$ `Web` to express that for each instance of `Application` included in the configuration a corresponding instance of `Web` should also be included. However, the notation and semantics of such constraint does not ensure that the `Web` instance should be in the subtree of the corresponding `Application` instance. The proposed cross-tree cardinality constraints lack scoping support and only guarantee that the final number of instances included in the configuration is correct, but provide no means to constrain where in the configuration hierarchy these feature instances should be included.

4.1.2 Challenges

Providing a way to specify feature cardinalities at different levels in the feature model hierarchy would make it possible to capture the variability found in cloud computing environments and to increase the expressive power of feature models. However, introducing new constructs to feature modeling may also affect its semantics and analysis methods.

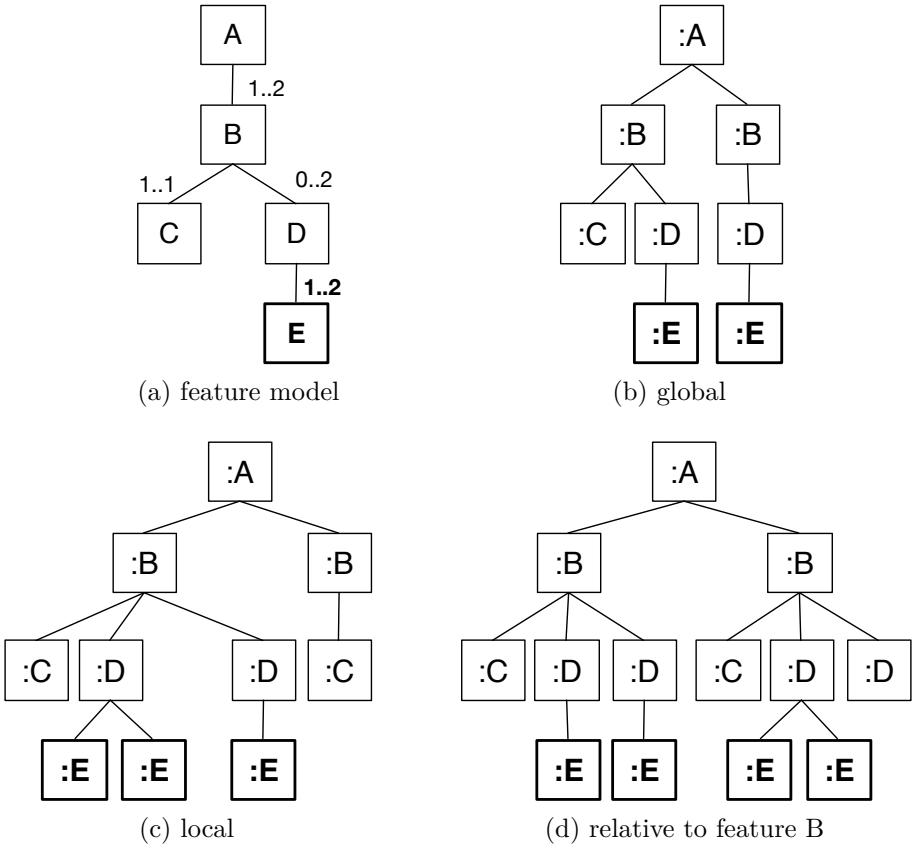
The goal of this work is to investigate how feature models can be extended to deal with multiple interpretations of the cardinality scope and their feasibility for modeling variability in cloud environments. This includes tackling the following challenges.

- ***Capture relative cardinalities in feature models***. The challenge is to extend cardinality-based feature models to take into account cardinalities that can be associated to features at different levels of hierarchy. Tackling this challenge requires a clear definition of syntax and semantics of such cardinalities and how they relate to other feature modeling constructs.
- ***Ensure cardinality consistency***. Introduction of relative cardinalities to feature modeling brings up new issues on the consistency of multiple feature cardinalities, which can be defined for different scopes. In this case, what are the criteria for relative cardinalities to be consistent and how do we ensure consistency in such a feature model?
- ***Handle additional constraints over relative cardinalities***. How does the introduction of relative cardinalities affect additional constraints and how can additional constraints syntax and semantics be updated to take relative cardinalities into account?
- ***Reason over feature models with relative cardinalities***. Relative cardinalities impact directly the set of valid configurations in a feature model. How do we update methods for automated analysis of feature models to take relative cardinalities into account?

4.2 Relative Feature Cardinalities

Semantics of feature cardinalities have been previously studied and formalized in many previous works [133, 120, 127, 159, 160]. A feature cardinality is usually defined as a numeric interval that defines the minimum and maximum number of instances of a feature that can be included in a product configuration. In existing feature modeling approaches, feature cardinalities are considered to apply either in a global or local scope. In the first case, all instances of a feature (throughout an entire configuration) are counted, while in the second case, the number of instances is counted for each instance of the parent feature.

Figure 4.2 shows a feature model and three sample configurations based on different interpretations of cardinality 1..2 of feature `E`. Figure 4.2a shows the feature model. In Figure 4.2b cardinalities are interpreted globally, thus no more instances of `E` can be added to the configuration (i.e., the limit is 2 in the configuration). In Figure 4.2c



(a) feature model (b) global

(c) local (d) relative to feature B

Figure 4.2 Feature model and configurations with different interpretations for cardinalities

cardinalities are interpreted locally, meaning each instance of `D` must have 1 or 2 children instances of `E`.

However, as illustrated in the motivating example from Section 4.1.1, there are cases in which the cardinality scope is neither global nor local, but relative to another feature. The semantics is exemplified in Figure 4.2d where the cardinality 1..2 of `E` is interpreted as relative to feature `B`. In this case, each instance of `B` must have 1 or 2 instances of `E` in its subtree, regardless of the number of instances of `D`.

4.2.1 Formalization

We introduce and formalize the concept of *relative cardinality*.

Definition 1. (Relative cardinality) The relative cardinality between two features x and y, such that x is descendant of y in the feature diagram, is the interval that defines the minimum and maximum number of x instances for each y instance in a configuration.

The concept of relative cardinality can be seen as a generalization of the possible interpretations for cardinalities in feature models. In this sense, the cardinality of a feature in relation to the feature model's root is equivalent to its global cardinality. Similarly, the cardinality of a feature in relation to its parent corresponds to its local cardinality.

Most cardinality-based feature modeling approaches consider that cardinalities apply locally [120, 121, 124, 126, 133, 162]. When feature cardinalities are evaluated locally, the feature diagram hierarchy defines an *implicit relative cardinality* between any feature and its ancestors. For example, in the feature model from Figure 4.2a, if we consider cardinalities apply locally we can infer that the implicit relative cardinality between `E` and `B` is 1..4. Given that each instance of `B` can have from 1 to 2 instances of `D` and for each instance of `D` 1 or 2 instances of `E` are allowed, the number of `E` instances for each `B` instance would be between 1 and 4.

Although feature cardinalities can be interpreted locally in many cases, there are scenarios where a local interpretation of cardinalities is insufficient. In these cases there is a mismatch between the implicit and the actual relative cardinality. By introducing relative cardinalities, we give first-class status to cardinalities that were only implicit, making it possible to override them in order to better capture the variability in the system.

Based on the work done by Michel et al. [133], we redefine a feature model as follows:

Definition 2. (Feature model) A feature model is a 7-tuple $\mathcal{M} = (\mathcal{F}, G, r, E, \omega, \lambda, \phi)$ such that:

- $\mathcal{F}$ is a non-empty set of features;
- $r \in \mathcal{F}$ is the root feature;
- $E : \mathcal{F} \setminus \{r\} \to \mathcal{F}$ is a function that represents the *parent-of* relation between features such that its *transitive closure* on $\mathcal{F}$, denoted by E^+, is *irreflexive* and *asymmetric*. These conditions guarantee the tree structure of the feature diagram;
- $\mathcal{C} = \{(x, y) \in \mathbb{N}^2 : x \leq y\}$ is the set of cardinality intervals;

- $\omega : E^+ \rightarrow \mathcal{C}$ is a function that represents the relative cardinality between two features that are part of the *ancestors-of* relation, denoted by E^+, the *transitive closure* of E on $\mathcal{F}$;
- $G \subset \mathcal{F}$ is a possibly empty subset of feature groups;
- $\lambda : G \rightarrow \mathcal{C}$ is a function that represents the group cardinalities of feature groups;
- ϕ is a set of cross-tree constraints (see Section 4.2.3).

The proposed definition updates the cardinality function ω to consider not only one but two features that are related in the feature model hierarchy. This update enables to specify multiple cardinalities for each feature, relative to any of the ascendants in the hierarchy. This allows for fine grained control on the number of allowed instances of a features at different contexts. This concept is more general and allows for expressing cardinalities at different levels, including global and local cardinalities.

From the definition of relative cardinality we can define the global and local cardinality of a feature as follows.

Definition 3. (Global cardinality) The global cardinality of a feature denoted by the function γ, can be defined as:

$$\gamma(f) = \omega(f, r)$$

Definition 4. (Local cardinality) The local cardinality of a feature denoted by the function ρ, can be defined as:

$$\rho(f) = \omega(f, E(f))$$

The semantics of feature models is defined by the set of configurations that comply with the variability defined by the feature model. Below, we define the syntax for cardinality-based feature model configuration and the conditions that must be met to comply with the feature model structure and cardinalities.

Definition 5. (Feature model configuration) A configuration for a feature model $\mathcal{M} = (\mathcal{F}, G, r, E, \omega, \lambda, \phi)$ is defined by a tuple $P_{\mathcal{M}} = (I, r_i, t, p)$ such that:

- I is a finite set of feature instances;
- $r_i \in I$ is the root feature instance;

- $t : I \to \mathcal{F}$ is a function that defines the feature type of an instance and $t(r_i) = r$;
- $p : I \setminus \{r_i\} \to I$ is a function that defines the parent-of relationship between feature instances such that $\forall i \in I \setminus \{r_i\}\ .\ t(p(i)) = E(t(i))$.

In addition, given the functions:

- $D(i,f) = \{\ j \in I \mid i \in p^{+}(j)\ \wedge\ t(j) = f\ \}$ that returns the descendant instances of i that are of feature type f;
- $CF(i) = \{\ t(j) \mid (\exists j \in I)[p(j) = i]\ \}$ that returns the selected children features of feature instance i;

a configuration is said to conform to feature model cardinalities iff:

- $(\forall i \in I, \forall f \in \mathcal{F})[(f, t(i)) \in E^{+} \implies (((x,y) = \omega(f,t(i)))\ \wedge\ (x \leq |D(i,f)| \leq y))]$;
- $(\forall i \in I, \forall g \in G))[t(i) = g \implies (((x,y) = \lambda(g))\ \wedge\ (x \leq |CF(i)| \leq y))]$.

These constraints ensure that both relative cardinalities and group cardinalities match those described in the feature model.

According to this definition, a feature model configuration is defined as set of feature instances organized in a tree in a way such that each feature instance has an assigned feature and parent, except for the root feature. The semantics of group cardinalities is defined on the scope of *features*, as defined in [133]. This implies, that the group cardinality indicates the number of features that can be select in a feature group, independent of the number of instances of the selected features. Meanwhile, the number of allowed instances is defined by the defined relative cardinalities. The last two constraints in the definition, enforce the compliance of valid configurations to both group and relative cardinalities.

4.2.2 Cardinalities Consistency

When relative cardinalities are considered, a feature may have multiple cardinalities, where each cardinality is relative to a different ancestor feature. The presence of features with multiple cardinalities brings up new issues concerning their consistency.

Cardinality consistency is linked to the notion of *range consistency* [163]. A cardinality is range consistent if each value in its range is used in at least one valid product configuration. Figure 4.3 depicts example feature models with inconsistencies (in red) in

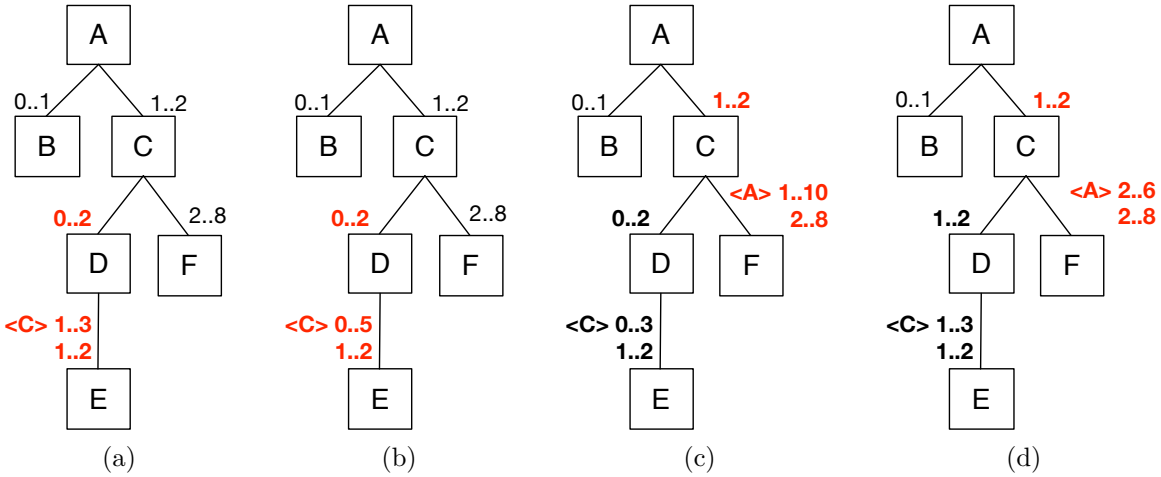


Figure 4.3 Relative cardinalities consistency

relative cardinalities. In the given examples, relative cardinalities are described above the feature node using the notation **<X> m..n**, where $m..n$ is its cardinality relative to feature `X`. When no specifier is given, the cardinality is considered to apply locally, in relation to the feature's direct parent. For example, the cardinality <C> 1..3 over feature `E` in Figure 4.2a represents the relative cardinality between `E` and `C` and 1..2 is the relative cardinality between `E` and `D`. This visual notation will be used throughout this dissertation. In these examples, it is equivalent to the expressions $\omega(E,C) = (1,3)$ and $\omega(E,D) = (1,2)$ in our feature model definition (see Definition 2).

In the feature model in Figure 4.3a, feature `E` has, in addition to its local cardinality 1..2, a cardinality of 1..3 relative to its ancestor `C`. However, this cardinality is inconsistent with the local cardinality 0..2 of feature `D`. Should an instance of `C` have no instances of `D`, there would not be any instance of `E` in its subtree and the cardinality 1..3 between `E` and `C` would not hold. Thus, there are no valid products with 0 instances of D, even though the cardinality of `D` is 0..2. From this analysis we can say that these cardinalities are inconsistent among themselves. Figures 4.3c and 4.3d show sample fixes (in bold black) for the inconsistency identified in Figure 4.3a where either the relative cardinality between E and C, or local cardinality of D, are updated.

In Figure 4.3b, the cardinality of `E` relative to `C` allows for up to 5 instances. However, each instance of `C` allows for a maximum of 2 instances of `D`, and for each of them a maximum of 2 instances of `E`, allowing for a total of 4 instances of `E` for each instance of `C`. Thus, the local cardinalities of `D` and `E` are not consistent with the 0..5 relative cardinality between `E` and `C`. In this case, there is no possible way to build a valid product

in which an instance of `C` has 5 instances of `E` in its subtree, even though their relative cardinality is 0..5.

Figure 4.3c shows another example of inconsistency where the relative cardinality between `F` and `A` is 1..10 but from the local cardinalities of `C` and `F` we can infer that we should have at least 2 instances of `F` in any product. Also, in Figure 4.3d we have an example where the relative cardinality between `F` and `A` has a maximum of 6 instances, which conflicts with the 2..8 local cardinality of feature `F`, since no valid products can exist with 7 or 8 instances of `F`.

When cardinalities are inconsistent they do not capture correctly the actual number of allowed feature instances and are subject to ambiguous interpretations. Based on the notion of *range consistency* and analysis of the semantic relation between multiple relative cardinalities, we define the criteria for cardinality consistency as follows.

Definition 6. (Cardinality consistency) Given the functions min and max that return the minimum and maximum values for a cardinality range, a feature model $\mathcal{M} = (\mathcal{F}, G, r, E, \omega, \lambda, \phi)$ is consistent concerning its relative cardinalities if $\forall x, y, z \in \mathcal{F} \mid (z, y) \in E^+ \ \wedge \ (y, x) \in E^+$ the following conditions hold:

$$min(\omega(z,x)) \geq min(\omega(z,y)) \ \cdot \ min(\omega(y,x)) \tag{4.1}$$

If each x instance has at least $min(\omega(y,x))$ instances of y and each y instance have at least $min(\omega(z,y))$ instances of z, then the minimum number of z instances for each x instance is at least $min(\omega(z,y)) \ \cdot \ min(\omega(y,x))$.

$$max(\omega(z,x)) \leq max(\omega(z,y)) \ \cdot \ max(\omega(y,x)) \tag{4.2}$$

If each x instance has at most $max(\omega(y,x))$ instances of y and each y has at most $max(\omega(z,y))$ instances of z, then each x instance can have at most $max(\omega(z,y)) \cdot max(\omega(y,x))$ instances of z.

$$min(\omega(z,x)) \leq max(\omega(z,y)) \ \cdot \ min(\omega(y,x)) \tag{4.3}$$

There should be at least one valid product in which the number of y instances for an x instance is the minimum $min(\omega(y,x))$. In this case, the maximum number of instances of z for each x is $min(\omega(y,x)) \ \cdot \ max(\omega(z,y))$. Thus, if $min(\omega(z,x)) > max(\omega(z,y)) \ \cdot \ min(\omega(y,x))$ there would be no valid products using the lower bound of the cardinality $\omega(y,x)$ and the cardinalities would not be consistent.

$$max(\omega(z,x)) \geq max(\omega(y,x)) \cdot min(\omega(z,y)) \tag{4.4}$$

Similarly, in at least one valid product, the number of y instances for an x instance should be $max(\omega(y,x))$. In this case, the minimum number of instances of z for each x instance would be $max(\omega(y,x)) \cdot min(\omega(z,y))$. Therefore, if $max(\omega(z,x)) < max(\omega(y,x)) \cdot min(\omega(z,y))$ there would be no product with the maximum cardinality $max(\omega(y,x))$.

$$min(\omega(z,x)) \leq min(\omega(z,y)) + max(\omega(z,y)) \cdot (max(\omega(y,x)) - 1) \tag{4.5}$$

At least one instance of y should have the minimum number of z instances $min(\omega(z,y))$. In this case, if one y instance has this minimum number of z instances, the maximum number of z instances for the x ascendant instance would be reached if all other y instances under the same x had the maximum number of z instances. That said, the maximum number of z instances for this x instance would be $min(\omega(z,y)) + max(\omega(z,y)) \cdot (max(\omega(y,x)) - 1)$. Therefore, if the minimum relative cardinality between z and x was greater than this value, there would be no cases where the lower bound of the cardinality $\omega(z,y)$ would be valid.

$$max(\omega(z,y)) \geq max(\omega(z,y)) + min(\omega(z,y)) \cdot (min(\omega(y,x)) - 1) \tag{4.6}$$

At least one instance of y should have the maximum number of z instances $max(\omega(z,y))$. Also, if at least one y has this maximum number, then the ascendant x instance would have at least $max(\omega(z,y)) + (min(\omega(y,x)) - 1)$. Therefore, if the maximum bounds of $\omega(z,x)$ is less than this value there would be no valid configuration that uses the maximum cardinality of $max(\omega(z,y))$.

This definition establishes that any three features that are linked in an ancestors-descendant relationship should meet the identified constraints. Together, they guarantee that for any value in the concerning cardinality ranges, at least one valid product can be defined in which this value is employed. Since these conditions should apply for all possible relative cardinality relationships, it guarantees that the feature model is consistent regarding relative cardinalities.

4.2.3 Relative Cardinalities and Additional Constraints

Previous works in the literature [133, 127] have pointed out the semantic ambiguities that arise when cross-tree constraints are included in cardinality-based feature models and the need for quantifiable cross-tree constraints, which also consider the number of feature instances and not only their presence or absence.

These limitations motivated Quinton et al. [127] to extend cross-tree constraints with cardinalities, allowing to express constraints over the global or local number of instances of a feature. Similarly, in [128], authors proposed an approach that supports to express cross-tree constraints directly to individual instances of features. These previous developments allow for defining constraints that can be interpreted in global or local scope, but do not consider relative cardinalities neither constraints involving features at different hierarchy levels.

In feature models with multiple relatives cardinalities, additional constraints need to be defined not only over feature cardinalities but also over their relative ones. To enable the definition of additional constraints over relative cardinalities, we update existing constraint notations and define their semantics. We first establish the notion of a *constraining expression*, and based on it, we define the concept of *relative cardinality constraints*.

Definition 7. (Constraining expression) A constraining expression is described by a tuple $\varepsilon = (r, c)$ where

- $r \in \mathcal{C}_*$ is a possibly unbounded cardinality range;
- $c \in E^+$ is a pair of features part of the child-ancestor relation in the associated feature model.
- $\mathcal{C}_* = \{(m,n) \in \mathbb{N} \times \mathbb{N} \cup \{*\} : m \leq n \ \vee \ n = *\}$ is the set of possibly unbounded cardinalities.

We use the notation **[m..n] (x, y)** to describe a constraining expression where $r = (m,n)$ and $c = (x,y)$. Alternatively, we also use the notation **(x, y) <= n** as a reduced syntax for **[0..n] (x, y)** and **(x, y) <= m** for **[m..*] (x, y)**. Such expressions represent a predicate over the number of instances of feature x that are descendant of an instance of y. Therefore constraining expressions represent a condition over a relative cardinality in a feature model.

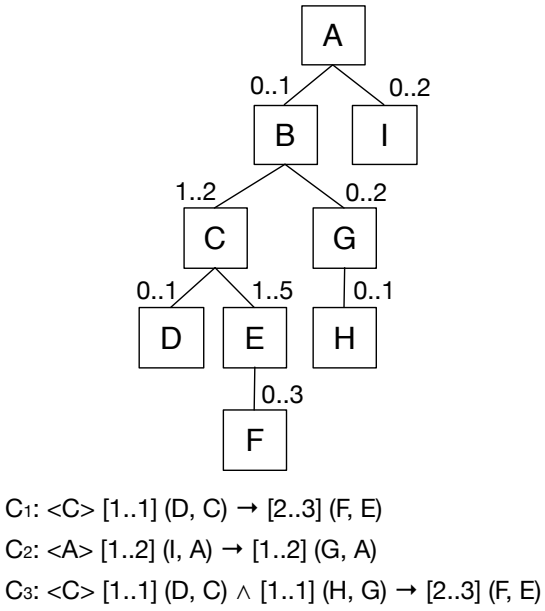


Figure 4.4 Constraints with relative cardinalities

Based on the concept of constraining expression, we define a cross-tree relative cardinality constraint as follows.

Definition 8. (Cross-tree relative cardinality constraint) A constraint is defined as an implication $<C> \varepsilon_1\ \delta\ ...\ \delta\ \varepsilon_n \rightarrow \varepsilon_{cons}$ where:

- $C \in \mathcal{F}$ is a feature that defines the context where the constraint will be evaluated;
- $\varepsilon_1 ... \varepsilon_n$ and ε_{cons} are constraining expressions;
- $\delta \in \{\wedge, \vee\}$ is a logical operation.

A relative cardinality constraint defines that if the composed left hand constraining expressions $\varepsilon_1\ \delta\ ...\ \delta\ \varepsilon_n$, the right hand ε_{cons} expression should also hold.

As shown in the above definition, constraining expressions are the base elements for describing a constraint. However, though they express a condition over a relative cardinality, they do not express in which context it should be evaluated. For example, considering the feature model in Figure 4.4, we can define a constraining expression such as [5..10] (`F`,`C`). Does this expression evaluate to true if one instance of `C` has 5 to 10 instances of `F` or if all instances of `C` have 5 to 10 instances of `F`?

The role of the context feature in the constraint is to remove this ambiguity and specify in which context expressions should be evaluated. Thus, in constraint C_1 (see Figure 4.4), the <C> context indicates that its constraining expressions should be evaluated individually for each instance of `C`. For instance, the C_1 constraint expresses that *`C` instances that have exactly one `D` instance as a child should have 2 or 3 instances of `F` for each `E` instance.* Actually, this constraint redefines the local cardinality of `F` (relative to its parent `E`) for some set of `C` instances, those that have exactly one `D` instance as a child.

Likewise, in C_2 the <A> context indicates that the constraint has to be evaluated globally, for the singleton instance of the root feature `A`. In this case, C_2 expresses a global implication that if at least one `I` instance is part of the product configuration then at least one `G` instance should also be included.

In C_1 and C_2, all constraining expressions deal with features that are under the subtree of the context features <C> and <A> respectively. The semantics of constraint evaluation are simpler as it suffices to consider each instance of `C` or `A` individually (and its subtree) and verify if the constraining expressions hold. However, how can we evaluate a constraint such as C_3, which combines expressions at different levels of hierarchy, including outside the context feature?

In this case, expressions whose features are not in the subtree of the context feature are evaluated in the context of the lowest common feature. This is the lowest common ancestor, in the feature diagram tree, of all features involved in the constraint; which in the case of C_3 is feature `B`.

With the described semantics, C_3 expresses that for each c instance of `C` and b instance of `B` such that b is an ancestor of c in a configuration tree, if c has exactly one child instance of `D`, and if all instances of `G` that are children of b have exactly one instance of `H`, then all instances of `F` that are children of this same c should have 2 or 3 instances of `E`.

The proposed constructs enable describing both simple and complex constraints involving relative cardinalities. The introduction of context features along with the semantics described above clarify how constraints can be evaluated even when they involve cardinalities from different levels in the feature diagram tree.

To enable multiple interpretations of cardinalities, we extended the definition of cardinality-based feature models, replacing feature cardinalities by relative cardinalities. We then identified the set of constraint conditions that ensures consistency between multiple

relative cardinalities. Finally, we introduced constraining expressions and redefined additional cross-tree constraints to take relative cardinalities into account.

4.3 Automated Analysis of Relative Cardinalities

The goal of the automated analysis is to extract information from feature models in an automated way [164]. Many analysis operations have been proposed in the literature [114], most of them targeted at finding inconsistencies such as dead features, void feature models, false optionals, wrong cardinalities, redundancies, etc. In this work, we developed methods for performing three operations: (i) inference and consistency verification of relative cardinalities, (ii) verification of a complete configuration against a feature model and (iii) generation of a complete configuration from a partial one. This section describes the tooling developed to support feature modeling with relative cardinalities, including the language support and analysis operations.

4.3.1 Feature Modeling with Relative Cardinalities

In order to automate the analysis of feature models with relative cardinalities we need means for formally describing them in a computer-processable way. In our approach we employ model-driven engineering tools to implement a domain-specific language that supports feature modeling with relative cardinalities and constraints, according on the definitions given in Section 4.2. The feature modeling language implementation is based on the xText framework [165]. The abstract syntax was designed using the EMF Ecore metamodel [166] and concrete syntax as an xText grammar.

Figure 4.5 depicts the abstract syntax of the designed language as a metamodel. The classes in gray represent the concepts commonly found in cardinality-based feature models. The root element of the metamodel is the `FeatureModel` class, which is composed by a `root Feature` and a set of `Constraints`. Both `Features` and `FeatureGroups` are a subclasses of `AbstractFeature`. The feature model hierarchy is defined by the decomposition of `FeatureGroups` and `Features` into `variants` and `subFeatures`.

The classes in green represent the concepts introduced to support modeling relative cardinalities. The main class is `RelativeCardinality` that associates a `Cardinality` range to a pair of `AbstractFeatures` (`from` and `to`). As defined in Section 4.2, a feature can have multiple relative cardinalities, one to each different ascendant in the hierarchy.

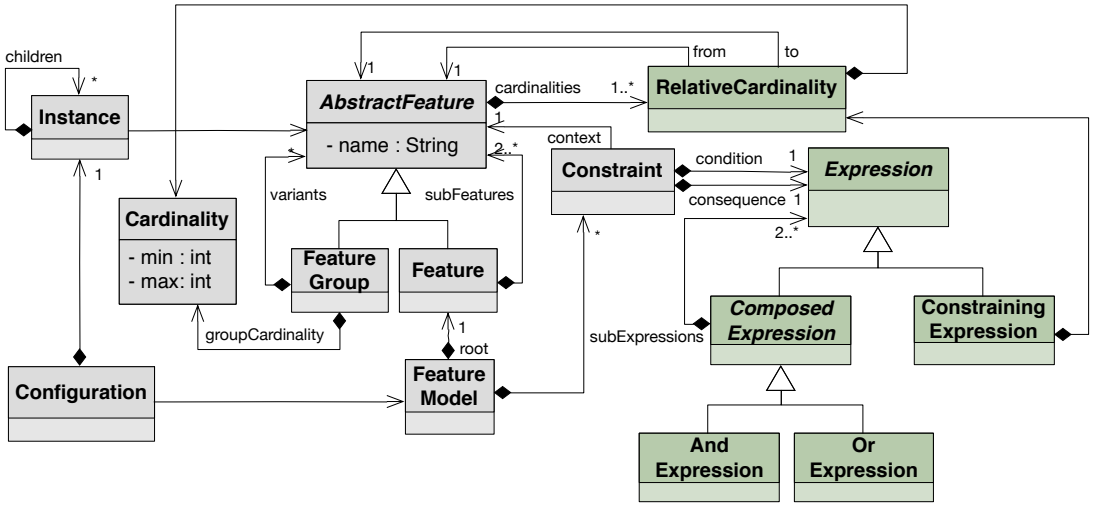


Figure 4.5 Metamodel for Feature Modeling with Relative Cardinalities

The metamodel also includes the classes `Constraint` and `ConstrainingExpression` for describing relative cardinality constraints as described in Section 4.2.3. Besides this, the metamodel also includes extra OCL constraints [167] that define static semantics that guarantee the well-formedness of feature models.

Figure 4.6 illustrates a feature model defined using the designed language. A feature model is described as a tree structure where braces ({}) define the decomposition hierarchy and brackets ([]) feature groups. Feature cardinalities are defined between brackets ([]) and groups cardinalities between less and greater than symbols (<>). Additional relative cardinalities are described using the <`X`> $[m..n]$ notation where $m..n$ is the cardinality relative to feature `X`. In line 8 of Figure 4.6 we can see that relative cardinality between `Gear` and `Application` is 1..16. Once a feature model is defined in the feature model DSL, we can check for the consistency of cardinalities and generate configurations for it.

4.3.2 Cardinalities Consistency

As we can see from the abstract syntax, the language allows the specification of multiple relative cardinalities. However, in most cases, relative cardinalities match exactly with the implicit relative cardinality derived from local cardinalities. In these cases, the product line designer may not want to describe all the relative cardinalities but rather those that are different from the implicit ones, letting the system infer the remaining cardinalities.

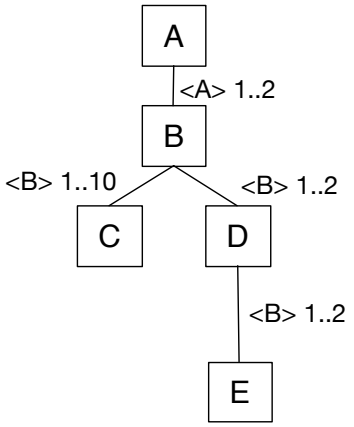
```
featureModel
OpenShift {
  Plan <1..1> [ Free | Bronze ]
  Application [0..16] {
    Scalable [0..1]
    Location <1..1> [ US | EU ]
    Cartridge [1..100] {
      Gear <OpenShift> [0..16] <Application> [1..16]
      GearSize [0..1] <1..1> [
        small | smallhighcpu | medium | large
      ]
      Type <1..1> [
        Web <Application> [1..1] <1..1> [
          Tomcat7 | Python | JBoss | NodeJS
        ]
        Addon <1..1> [ MongoDB | MySQL | PostgreSQL ]
      ]
    }
  }
}
<OpenShift>  Free => [0..3] (Gear, OpenShift)
<Application> Free => US & [1..1] (small, GearSize)
<Cartridge> [0..0] (Scalable, Application) & [1..1] (Web, Cartridge)
                          => [1..1] (Gear, Cartridge)
<Cartridge> !Scalable & Addon => Gear = 0
```

Figure 4.6 OpenShift feature model textual representation

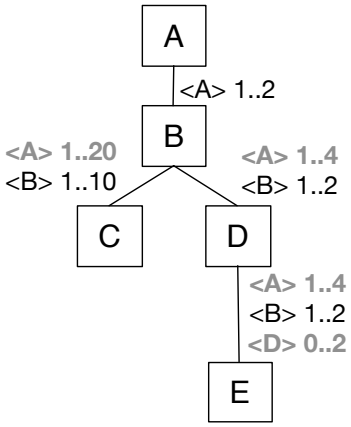
For example, in Figure 4.6, we can see that relative cardinalities were only defined for features `Gear` and `Web`. In the other cases, the expected behavior is expected to be the same of local cardinalities. So, even though relative cardinality between `Scalable` and `OpenShift` is not defined, we can infer from local cardinalities of `Scalable` and `Application` that it should be 0..16 Also, in the case of `Gear` feature, the cardinality relative to its direct parent `Cartridge` is not defined and should be inferred from the other cardinalities. In this case, we can infer that local cardinality of `Gear` is 1..16.

To support this requirement the designed language requires to specify only one cardinality for each feature (local or relative) and can automatically infer the remaining cardinalities. The inference is achieved by translating cardinality consistency conditions described in Section 4.2.2 into a constraint program and using an constraint programming (CP) solver [168] to find a solution that maximizes the inferred cardinality ranges.

The mapping from consistency conditions to a constraint program is straightforward. First, for any possible relative cardinality we create two variables that represent the lower and upper bound of the relative cardinality range. Thus, for any two features



(a) feature model



(b) feature model with inferenced cardinalities

LB(C, B) = 1	UB(C, B) = 10
LB(C, A) = [1..20]	UB(C, A) = [1..20]
LB(B, A) = 1	UB(B, A) = 2
LB(E, D) = [1..2]	UB(E, D) = [1..2]
LB(E, B) = 1	UB(E, D) = 2
LB(E, A) = [1..8]	UB(E, A) = [1..8]
LB(D, B) = 1	UB(D, B) = 2
LB(D, A) = [1..4]	UB(D, A) = [1..4]

(c) CP variables

for all $x,y,z \in \mathcal{F}$ **such that** $(z,y) \in E^+$ **and** $(y,x) \in E^+$
LB(z, x) >= LB(z, y) * LB(y, x)
UB(z, x) <= UB(z, y) * UB(y, x)
LB(z, x) <= UB(z, y) * LB(y, x)
UB(z, x) >= UB(y, x) * LB(z, y)
LB(z, x) <= LB(z, y) + UB(z, y) * (UB(y, x) - 1)
UB(z, y) >= UB(z, y) + LB(z, y) * (LB(y, x) - 1)

(d) CP constraints

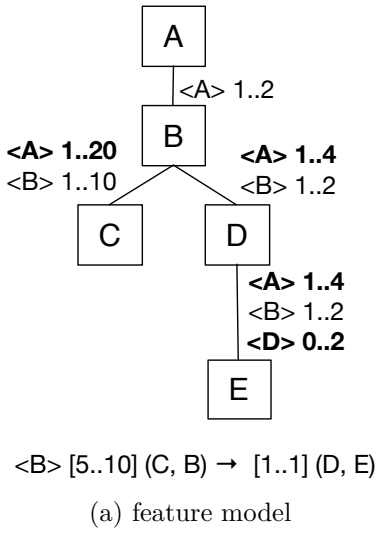
$$max \sum_{(a,b) \in E^+} UB(a,b) - LB(a,b)$$

(e) CP optimization goal

Figure 4.7 Consistency check and inference of relative cardinalities

$(a,b) \in E^+$, we create two CP integer variables $LB(a,b)$ and $UB(a,b)$. If the relative cardinality between these features was specified in the feature model, we assign the specified values to these variables. Then, for any three features x,y,z linked through an ancestors-descendant hierarchy we add constraints that correspond directly to the consistency conditions of Definition 6.

Figures 4.7a and 4.7b illustrate a feature model where not all cardinalities are specified and the resulting feature model after inference. Figures 4.7c and 4.7d show the CP variables created for representing cardinalities in this feature model and how the constraints are generated from consistency conditions. The expression in Figure 4.7e represents the optimization objective, which is to maximize the cardinality ranges. As the inference process relies on the cardinality consistency constraints, the inferred cardinalities are always consistent. In addition, if the specified cardinalities are not consistent, the



(a) feature model

$A = [0..1]$
$A_1 = [0..1]$
$A_1B = [0..2]$
$A_1B_1 = [0..1]$
$A_1B_1C = [0..10]$
$A_1B_1D = [0..2]$
$A_1B_1E = [0..2]$
$A_1B_1D_1 = [0..1]$
$A_1B_1D_1E = [0..2]$
$A_1B_1D_2 = [0..1]$
$A_1B_1D_2E = [0..2]$
$A_1B_2 = [0..1]$
$A_1B_2C = [0..10]$
$A_1B_2D = [0..2]$
$A_1B_2E = [0..2]$
$A_1B_2D_1 = [0..1]$
$A_1B_2D_1E = [0..2]$
$A_1B_2D_2 = [0..1]$
$A_1B_2D_2E = [0..2]$

(b) CP variables

$A_1B_1E = A_1B_1D_1E + A_1B_1D_2E$
$A_1B_2E = A_1B_2D_1E + A_1B_2D_2E$

if A_1B_1C **in** [5..10] **then**
  A_1B_1E **in** [2..2]

if A_1B_2C **in** [5..10] **then**
  A_1B_2E **in** [2..2]

(c) cross-tree constraints to CP

Figure 4.8 Feature model translation to CP

corresponding constraint problem will not lead to a valid solution. Therefore, the cardinality consistency check and inference are executed as a single process.

4.3.3 Configuration Checking

To automatically verify if a configuration conforms to a feature model, we also translate it into a constraint program and rely on a solver to verify the validity of the configuration against the feature model. We base our approach on the translation method described by Mazo et al. in [126], extending it to deal with relative cardinalities.

The referenced method considers cardinalities to apply locally, therefore enabling configurations that consider not only the number of feature instances, but also how they are hierarchically organized. For each possible feature instance, a boolean variable is created to represent if the instance is part of a configuration; and a set of integer variables are created to represent the number of instances of each of its subfeature types. For example, when translating the feature model in Figure 4.8a, for each instance of feature B, it creates a boolean variable B_i and two integer variables B_iC and B_iD with domains [0..10] and [0..2] that represent the number of instances of C and D for the i-th instance of B.

Additional constraints are added to enforce the minimum number of instances for mandatory features, hierarchical dependencies and group cardinalities.

For the semantics of relative cardinalities, besides creating variables representing the number of instances for each direct subfeature, we also create variables that represent the number of instances of indirect subfeatures. Still in the example of Figure 4.8, it means that in addition to the variables B_iC and B_iD, we create a variable B_iE representing the number of instances of E for this given instance of B. Constraints are added to guarantee that B_iE matches with the number of instances of E for each of its D instances. In the given example, we add the constraint $B_iE = B_iD_1E + B_iD_2E$. The variables and constraints generated by this process for the feature model in our example are shown in Figures 4.8b and 4.8c. The variable names are prepended by A_1 to express that they represent instances that are under the singleton instance of the root feature A.

To improve the translation and solving of the constraint problem, variables and constraints for relative cardinalities are only added when they are explicitly declared and do not match the implicit cardinality derived from the individual local cardinalities. After generating the variables, they can be used for implementing the additional cross-tree constraints using simple logic implication constraints. For example, the cross-tree constraint $<B>$ [5..10] $(C, B) \rightarrow$ [1..1] (D, E) would be translated into the following CP constraint for each i-th instance of B:

$$\text{IF } (B_iC \text{ IN } [5..10]) \text{ THEN } (B_iD_1E \text{ IN } [1..1] \text{ AND } B_iD_2E \text{ IN } [1..1]).$$

As discussed earlier, a configuration for a clone-enabled feature models is a tree of feature instances. Once a feature model is translated into a constraint program, verifying the compliance of a configuration is very simple. It suffices to assign 1 to the boolean variables that correspond to feature instances included in the configuration, and 0 to all other variables representing feature instances. If the constraint program is satisfiable, the configuration conforms to the feature model, otherwise not.

4.3.4 Partial Configuration Checking

In many cases, specifying a complete configuration of a system may be unfeasible or require a lot of effort due to the large number of features and ways that feature instances can be combined to form a configuration. Besides this, we may be interested in a configuration that concerns only a small subset of features, regardless of the presence of other features. Still, we need to verify if this partial config is valid and how to obtain a complete configuration from it. In the literature on feature models these operations are called respectively as *partial configuration validation* and *auto-completion of partial configurations* [121, 114]. These operation take a feature model and a partial configuration

as input and verify if there are no contradictions in the partial configuration. In addition, autocompletion also returns a new configuration that is obtained by propagating the feature model constraints.

Operations on partial configurations are usually mentioned in the literature in the context of boolean feature models. In this case, a partial configuration is defined by sets of selected and removed features. However, in clone-enabled feature models a configuration is not only defined by a set of selected features but a tree of feature instances. Few works deal with partial configurations in clone-enabled feature models [126, 131] and there is little discussion in the literature on the representation of partial configurations. In [131], authors provide a language in which both complete and partial configurations are described as a tree of feature instances. In [126], complete and partial configurations are defined by a set of features together with an integer that specified the number of instances of each feature in the configuration.

Both approaches, suffer from limitations. In the first case, the complete hierarchy of the feature model needs to be followed even for specifying partial configurations. In the second case, the hierarchical decomposition of feature instances is ignored. Finally, none of the partial configuration notations support the exclusion of features.

In our proposed language, configurations are also defined in a hierarchy that matches that of the feature model. However, in a partial configuration, not all required feature instances need to be included and intermediary features may be omitted. To avoid repeating identical instances, feature model designers can also add the number of desired instances of a feature.

In Figure 4.9a, lines 3-10 include two `Application` instances with the same decomposition of features. Line 14 states that the third `Application` should have a `Cartridge` of size greater than `small` with 10 instances of `Gear`. The `0 small` in line 18 is used to exclude instances of feature `small` from the subtree of this `Cartridge` instance. Overall, this example describes a configuration for the OpenShift provider to deploy two Python application services in Europe as well as a Java application service and a Mongo database in the United States.

In Figure 4.9b we have a partial configuration that includes 2 instances of feature `Cartridge` in which each of this instances has features `Web` and `Java` included at least once in their subtree. This example describes a configuration to deploy two Java web application services. This partial configuration omits the `Application` feature to highlight that it is not important how the obtained `Cartridge`s are distributed across different

```
config OpenShift

2 Application {
  EU
  Cartridge {
     Gear
     medium
     Python
   }
}
Application {
  US
  Cartridge {
    10 Gear
    0 small
    Tomcat7
  }
  Cartridge {
    MongoDB
  }
}
```

(a)

```
config OpenShift

2 Cartridge {
  Web
  Java
}
```

(b)

Figure 4.9 Partial configuration examples

`Applications`. Actually, the OpenShift feature model defines that each `Application` has exactly one `Cartridge` of the `Web` kind. Therefore, complete configurations for this example will require two instances of `Application`.

To implement this operation, we encode a feature model as a constraint program as in the case for verifying a complete configuration. However, instead of assigning values to all variables that represent feature instances, we rely on CP variables representing relative cardinalities to define the number of expected instances. Figure 4.10 illustrate how these partial configurations would be translated into CP constraints.

This language gives great flexibility in the specification of partial configurations as designers can choose the level of detail that is more convenient according to the requirements at hand. It is particularly useful in the case of cloud computing as we are often interested in features that provide a given service rather than how they are structured.

The developed tools allow to define feature models with relative cardinalities and their configurations using a domain specific modeling language. The cardinality inference and

$$\texttt{OpenShift}_1\texttt{Application} = 3$$

$$\texttt{OpenShift}_1\texttt{Application}_1\texttt{EU} > 0$$
$$\texttt{OpenShift}_1\texttt{Application}_1\texttt{Cartridge}_1\texttt{Gear} > 0$$
$$\texttt{OpenShift}_1\texttt{Application}_1\texttt{Cartridge}_1\texttt{medium} > 0$$
$$\texttt{OpenShift}_1\texttt{Application}_1\texttt{Cartridge}_1\texttt{Python} > 0$$

$$\texttt{OpenShift}_1\texttt{Application}_2\texttt{EU} > 0$$
$$\texttt{OpenShift}_1\texttt{Application}_2\texttt{Cartridge}_1\texttt{Gear} > 0$$
$$\texttt{OpenShift}_1\texttt{Application}_2\texttt{Cartridge}_1\texttt{medium} > 0$$
$$\texttt{OpenShift}_1\texttt{Application}_2\texttt{Cartridge}_1\texttt{Python} > 0$$

$$\texttt{OpenShift}_1\texttt{Application}_3\texttt{US} > 0$$
$$\texttt{OpenShift}_1\texttt{Application}_3\texttt{Cartridge}_1\texttt{Gear} = 10$$
$$\texttt{OpenShift}_1\texttt{Application}_2\texttt{Cartridge}_1\texttt{smal} = 0$$
$$\texttt{OpenShift}_1\texttt{Application}_2\texttt{Cartridge}_1\texttt{Tomcat7} > 0$$
$$\texttt{OpenShift}_1\texttt{Application}_2\texttt{Cartridge}_2\texttt{MongoDB} > 0$$

Figure 4.10 CP constraints for partial configuration

check mechanism prevents feature model designers from describing inconsistent cardinalities and automatically infer unspecified cardinalities. The translations provided from feature modeling to constraint programming enable to verify the validity of configurations or generate complete configurations from partial ones.

4.4 Discussion

Relative cardinalities provide a generalization of feature cardinality scope, and are particularly useful when modeling variability in resource constrained environments. In cloud computing environments, relative cardinalities enable to express the cardinality scope of cloud resources and services using a simplified notation. As our investigation was focused on the cloud domain, it is not possible to conclude about its usefulness on other domains.

The introduced constructs are a super set of those commonly found in cardinality-based feature modeling approaches. Hence, the proposed modeling language and analysis tools will behave as regular cardinality-based feature models in the absence of relative cardinalities. The introduced notation and semantics for additional cross-tree constraints over relative cardinalities is also more general than previous existing notation and supports expressing both global and local constraints. In addition, the introduction of a context feature, together with a clear definition of semantics allows for defining constraints involving features from diverse places in the feature model hierarchy.

Our approach handles feature model variability at the feature instance level, which implies that each feature instance may have a different decomposition of its subfeatures. Likewise, additional cross-tree constraints are evaluated individually for each instance of the specified context feature. Previous works proposing additional constraints in cardinality-based feature models [127, 122] do not handle feature instances and only reason over the number of times a feature is included, regardless of their decomposition. Meanwhile, works that employ an instance-based reasoning either provide limited supports for additional constraints between features [126] or require constraints to be defined individually for each feature instance [128].

Like most works on cardinality-based feature models [122, 126–128], this approach assumes that feature cardinalities are bounded. In [131, 132], authors employed ILP and SMT solvers to automate consistency checking and anomaly detection in feature models with unbounded cardinalities. Meanwhile, in [130], authors propose to use Generative CSP [169] to reason on feature models with unbounded cardinalities.

Another approach that has been proposed to handle complex variability is to directly use constraint programing instead of feature models [170]. Authors argue that translating feature modeling constructs into constraint programs limits the potential offered by constraint solvers.

4.5 Summary

In this chapter we introduced the concept of relative cardinalities to feature modeling. We presented a definition of syntax and semantics of feature models with relative cardinalities and we discussed how they relate to existing interpretations of feature cardinalities. In addition, we analyzed the impact of introducing relative cardinalities on the consistency of feature models and additional cross-tree constraints. Finally, we demonstrated how automated analysis of feature models with relative cardinalities can be achieved by translating it into a constraint program.

Together, these constructs enable to use feature models for managing the variability that arises in the configuration of cloud environments. The work presented in this chapter provides an answer to RQ_1 outlined in Section 1.2. Indeed, with the use of relative cardinalities, it is possible to handle the configuration variability of cloud environments

In addition to configuration variability, cloud environments may be subject to complex constraints governing their reconfiguration. In the following chapter, we present an

approach for handling dynamic variability that arises when adapting running cloud environments.

Chapter 5

Feature Models and Temporal Constraints

One of the main properties of the cloud computing model is the support for rapidly provisioning and releasing resources with minimal management effort or interaction with the service provider [1]. This property makes it possible to dynamically change the resources used in a cloud system with no need for human interaction. The support for dynamic provisioning of resources in cloud computing opens the way for building adaptive cloud environments, capable of reconfiguring themselves to cope with context changes and new application requirements.

The high-variability found in the configuration of cloud providers, together with their support for dynamic provisioning, make Dynamic Software Product Lines (DSPLs) a promising approach for building adaptive cloud environments. As discussed in Section 3.3, DSPLs are a systematic approach for building adaptive systems based on concepts, techniques and tools from product line engineering. DSPLs employ a variability model to represent the system configuration space and ensure that adaptations lead to correctly working configurations. Once a configuration is selected and validated against its variability model, it relies on the adaptation mechanisms available in the underlying platform to enact it.

However, in cloud computing providers, the adaptation mechanisms available for reconfiguring a running environment are subject to complex rules and constraints. Cloud providers may offer multiple alternative ways of updating a cloud environment to a target configuration, each one with a different impact on performance, costs and downtime. Choices made in the initial configuration or in previous reconfigurations may affect the set

of available future configurations, thus changing the system's variability over its lifecycle. Besides this, each provider may have their own rules concerning the reconfiguration of cloud environments, hence the same context or requirements change will lead to different reconfigurations in each provider.

In order to correctly reconfigure a cloud environment, we must ensure that all provider reconfiguration constraints are followed. Initial works on DSPLs [136, 171] highlight the importance of guaranteeing a safe transition from an initial to a target configuration during system adaptation. However, in most DSPL approaches, verification is still limited to checking if a target configuration complies with a variability model. In existing approaches for building DSPLs, there is little regard about how the system transitions between configurations, if there are any constraints over these transitions, or if multiple alternatives are available.

To overcome this limitation and support the complex reconfiguration rules found in the cloud computing domain, we introduce temporal constraints and reconfiguration operations to variability modeling of DSPLs. Together, these constructs support a better definition of the adaptive behavior of a DSPL and reasoning over multiple alternative reconfiguration paths. In this chapter, we describe how these concepts are integrated into feature modeling and demonstrate how principles from model checking can be used to reason over them in order to validate reconfiguration requests and find complying reconfigurations. This contribution answers RQ_2 introduced in Section 1.2 and shows that temporal properties can be used to handle reconfiguration constraints in feature models.

In Section 5.1, we discuss the motivation for using temporal constraints in DSPLs for cloud computing environments. In Section 5.2, we define the semantics of temporal logic constraints and reconfiguration operations on feature models. In Section 5.3, we demonstrate how we can use symbolic representation to support reasoning over feature models with temporal constraints and reconfiguration operations. In Section 5.4 we show how these concepts can be integrated into feature models with relative cardinalities. Finally, Section 5.5 discusses the implications of this work and Section 5.6 summarizes this chapter.

5.1 Motivation

When a cloud environment is initially setup for an application, it is conceived to support initial application requirements. However, as application requirements evolve or the system load changes, its cloud environment must be adapted to cope with new requirements. Examples of adaptations include the provisioning of extra resources such as virtual machines or application containers when more requests arrive, upgrading to a larger database plan as the system usage increases, changing the application framework to cope with application evolution, etc.

Systems that exhibit runtime variability, such as cloud environments, are candidates for employing a DSPL architecture. DSPLs leverage concepts and engineering foundations from SPLE to support the construction of adaptive systems [172]. The core element of both approaches is a variability model that specifies variation points in which members of a product line can differ. In DSPLs an adaptation is usually triggered by monitored context features or another system component requesting the inclusion (exclusion) of a set of features to (from) the current configuration. Techniques for automated analysis are then used to verify if the requested configuration complies to the variability model structure and constraints. Once a target configuration is validated, adaptation mechanisms from the underlying platform (e.g., component models, reflection, aspects, models@runtime) are used to enact the it.

However in cloud computing, adaptation mechanisms may vary according to the provider, and even for the same provider different services may have different support for adaptation. While some services may allow their settings to be updated at runtime, others may require a service restart or still a complete redeployment of the service.

5.1.1 Motivating Example

To illustrate the problems faced while managing variability and adaptation in cloud environments we present an example from the Heroku PaaS provider. The feature model in Figure 5.1 describes part of the variability in the configuration of Heroku PaaS provider. In Heroku, users can define multiple application projects that include support for development in a given programming language such as `Java`, `Ruby` or `PHP` and a set of additional cloud services such as databases, caching, monitoring tools, etc. They can also choose a geographical region (`US` or `EU`) in which the application project should be

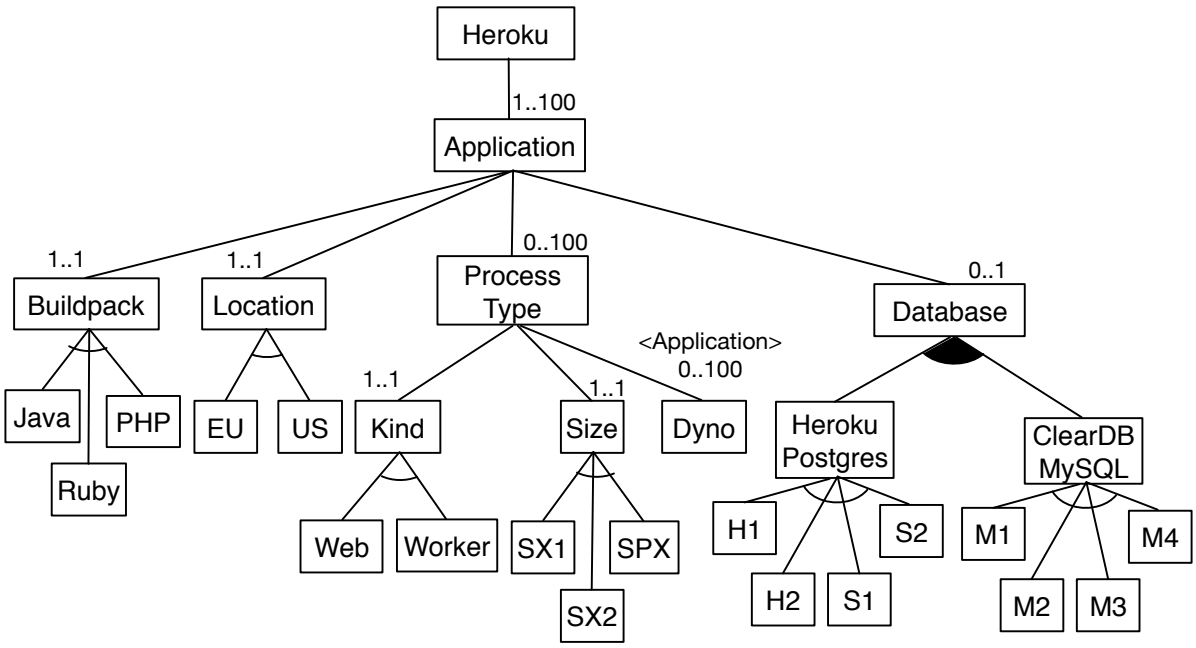


Figure 5.1 Excerpt from Heroku cloud feature model

deployed and additional cloud services may be available at different plans (`M1`, `M2`, `H1`, `H2`, `S1`, `S2`, etc), with different capabilities and costs.

While initially setting up an application project, customers have access to all cloud services, and can select any configuration that complies with the feature model. However, after the initial setup, reconfigurations may be limited by the initial choices or require extra operations. Below we present some of the reconfiguration constraints identified on the Heroku PaaS provider.

- *MySQL database plan.* Four different plans are available for the ClearDB MySQL service, each one with different storage and pricing. Heroku provides support for upgrading directly from a smaller to a larger plan [173], however it does not support downgrading to a smaller plan. Thus, once a choice is made, future reconfigurations should not consider smaller plans, even if the feature model describes it as a possible alternative.

- *PostgreSQL database plan.* Heroku Postgres provides a wide range of service plans classified into three categories: Hobby, Standard and Premium (Figure 5.1 depicts only Hobby `H1`, `H2` and Standard `S1`, `S2` service plans, but Heroku provides a total of 17 different service plans). Heroku proposes two ways to upgrade or downgrade between different plans [174]. The first way relies on the use of `pgcopy` tool, in which data is backed up and restored to a newly created database. According to

Heroku documentation using `pgcopy` requires a downtime of about 3 minutes per GB. The second method creates a follower database using the new desired service plan and synchronize it with the original database in the background. This process can take several hours, in which additional costs for maintaining two databases should be considered, but downtime is minimal or unnoticeable. Reconfigurations involving any Hobby plans are also only possible with the `pgcopy` method.

- *Framework type.* Changing the application development framework [175], also called as *buildpack* in Heroku terminology, can be achieved by updating a configuration variable and redeploying the application code. This change requires the application server instances to be restarted.
- *Application container size.* Application container instances (called *dynos* in Heroku) can be easily resized at runtime [176] by setting the size attribute. Heroku platform will take care of alternately replacing instances of previous size by new instances of the requested size.
- *Application location.* Changing the location of an application project requires migrating it to another data center [177] and therefore implies a series of complex operations, which may include the migration of application code and data. Similar to the examples above, this can be achieved in different ways by combining different operations, with different effects on costs, downtime and performance of the application during migration.

Even from this simple example, we can notice that cloud environments may be subject to complex constraints governing when and how a new configuration can be reached. However, most DSPL approaches provide little or no support for modeling and reasoning over reconfigurations constraints and analysis is mostly restricted to the verification of the static variability described through a variability model.

5.1.2 Challenges

Building adaptive cloud environments requires managing the constraints that govern how a running cloud environment can be dynamically reconfigured. This requires means for modeling and reasoning over these constraints together with the variability that defines the system configuration space.

The goal of this work is to investigate the use of temporal constraints to support modeling and reasoning over cloud environments adaptation. More specifically, we deal with the following challenges.

- ***Model temporal constraints on DSPLs***. The first challenge is to model the complex constraints that govern the reconfiguration of cloud environments. These constraints represent temporal dependencies among features, which define how the system variability changes over the execution of system in response to previous configuration choices.
- ***Model reconfiguration operations***. The challenge is to model the reconfiguration operations required to reconfigure a cloud environment to a chosen target configuration. This includes dealing with multiple alternative reconfiguration paths available in cloud providers as well as the relationship between features and reconfiguration operations.
- ***Finding reconfigurations***. Given a change request the challenge is to reason over a feature model and reconfiguration constraints in order to efficiently identify valid reconfigurations from the current system state.

5.2 Variability Modeling with Temporal Constraints

Previous works [136, 171] on DSPL highlight the need for adaptive systems to perform safe transitions between configurations. According to Morin et al. [136], systems should evolve through a safe migration path between configurations, while Hubaux and Heymans [171] point out that transitions should not only consider static constraints of a variability model but also the "dynamic constraints determining the allowed transitions between configurations". In spite of it, existing works on DSPLs provide little support for reasoning over transitions between configurations.

In some approaches, the variability model is augmented with context [105, 144, 140, 141, 106, 142, 178, 107] or binding time [179, 180] information. Context information is usually captured as *context features* that, together with additional constraints, define the interaction between context and system features. These approaches are based on the idea of modeling the *context variability* [105, 106] that defines the commonalities and variability of the environment in which the system executes. Context changes are thus represented by the (de)activation of context features, while additional constraints

propagate these changes to system features. Hence, context features are used to establish a link between context and system changes.

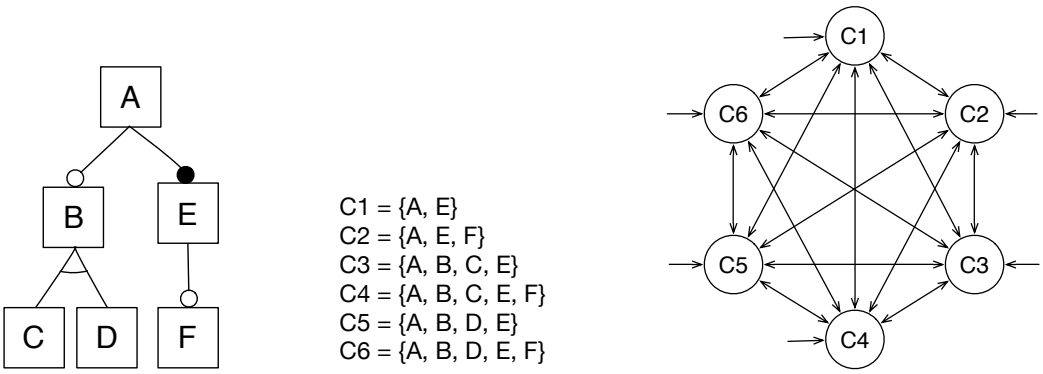
Binding time is a property of a variation point that defines at which stage in the application lifecycle it should be bound. Binding times commonly used in existing approaches include design, implementation, compilation, build, assembly, configuration, deployment, start-up and runtime [155]. Adding binding time information can be seen as a way to delay decisions to later stages and define which parts of a system can be dynamically reconfigured at runtime. In [181], authors use constraints over binding times, features and attributes to constraining the reconfiguration process in DSPLs. A binding time constraints can be used to define a dependency between the (de)activation of a feature and a feature binding time, thus providing support for defining some temporal dependencies among features.

While these extensions improve the expressiveness of variability models for DSPLs they do not provide enough means for modeling or reasoning on how a DSPL transitions between configurations. After all, verification is still restricted to monitoring context changes, checking what features need to be (de)selected and verifying if these features can be rebound at runtime.

According to [182, 136, 140] a DSPL can be abstracted as a state machine in which states represent valid system configurations and state transitions define system adaptations. The variability model is thus a compact notation for describing an underlying reactive transition system [183] that defines the adaptive behavior of the system. By using a variability model we eliminate the need to enumerate all possible system configurations and transitions and define only the rules on how valid configurations can be built.

Figure 5.2 illustrates this vision. Here we have a boolean feature model (Figure 5.2a) from which we can obtain a set of possible system configurations (Figure 5.2b). A DSPL for this feature model can be regarded as the transition system depicted in Figure 5.2c, in which the system can start in any configuration and transition to any other configuration as long as these configurations conform to the feature model.

However, as a variability model abstracts away from its underlying transition system, it fails to capture any restriction that may exist on the transitions between configurations. By introducing temporal constraints and reconfiguration operations to the definition of a DSPL, we give first class status to these transitions while looking for a compromise between full specification of a transition system and the more abstract view based on variability models.



(a) feature model (b) configurations (c) transition system

Figure 5.2 Feature model as a compact definition of a transition system

In the remainder of this section we present how a DSPL can be represented as a transition system and how temporal constraints and reconfiguration operations can be used to improve their definition. In the first moment, we employ boolean feature models as a representation for variability model. Later, in Section 5.4 we demonstrate how these concepts can be applied to cardinality-based feature models such as those employed in cloud computing environments.

5.2.1 Preliminary Definitions

In the following paragraphs, we go over definitions for feature models, transition systems and DSPLs that will be used throughout this chapter.

Definition 9. A feature model can be defined as a tuple $M = (\mathcal{F}, C)$ where $\mathcal{F}$ is the set of features and $C \subseteq \mathcal{P}(\mathcal{F})$ is the set of valid products that can be derived from the feature model. We also use the $[\![M]\!]_{FM}$ notation to denote the set of products of a feature model M.

From this definition, we can notice that a feature model defines a finite configuration space where each valid configuration is defined by a set of active features.

Definition 10. A reactive transition system [183] can be described by a Kripke structure $T = (S, I, R, AP, L)$ where S is a finite set of system states, $I \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a left-total relation representing state transitions, AP is a set of atomic propositions and $L : S \to 2^{AP}$ is a labeling function.

A reactive transition system has no final states and is not expected to terminate and produce a result as output, but rather maintain a continuous interaction with its environment [184]. As described above, the transition relation is left-total, which implies that all states have at least an outgoing transition. The atomic propositions in AP are used to define known facts about a system state through the labeling function $L :\rightarrow 2^{AP}$. Thus, $x \in L(s)$ means that a given fact $x \in AP$ holds at state s.

Definition 11. An execution path of a transition system $T = (S, I, R, AP, L)$ is a non-empty infinite sequence of states $\pi = \langle s_0, s_1, s_2, s_3, ... \rangle$ such that $s_0 \in I$ and $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. Formally, an execution path is a function $\pi : \mathbb{N} \rightarrow S$ such that for each $i \in \mathbb{N}$ we have that $(\pi(i), \pi(i+1)) \in R$.

Definition 12. The behavior of a transition system T is defined by the set all possible execution paths, denoted by $[\![T]\!]_{TS} \subseteq S^{\omega}$, in which S^{ω} is the set of all infinite sequences of states in T.

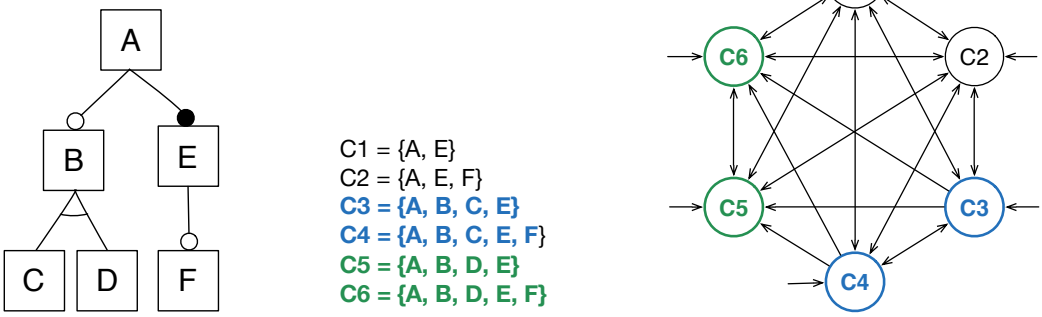
Definition 13. Given a feature model $M = (\mathcal{F}, C)$, the corresponding DSPL can be represented as a transition system $DSPL(M) = (S, I, R, AP, L)$ where $I = S = C$, $R = S \times S$, $AP = \mathcal{F}$ and $L(s) = s$.

From this definition, a DSPL can be seen as a transition system in which each state represents a possible configuration according to the feature model. The system can start at any configuration and transition to any other possible configuration, as illustrated in Figure 5.2. Atomic propositions are represented by features and the state labels correspond to the features selected in the corresponding configuration.

5.2.2 Temporal Constraints

A temporal property [185] defines a condition over the execution paths of a transition system. A transition system is said to satisfy a given property if all of its possible execution paths satisfy the property. Thus, a temporal property specifies the admissible or desired behavior of a system over time.

Definition 14. Let $T = (S, I, R, AP, L)$ be a transition system, a temporal property $P \subseteq S^{\omega}$ over T is a set of infinite sequences of states in T. The transition system T satisfies the property P, written $T \models P$ iff $[\![T]\!]_{TS} \subseteq P$.



(a) feature model (b) configurations (c) transition system

$$P = \{\langle s_0, s_1, s_2, s_3, ...\rangle \in S^\omega \mid (\forall i \in \mathbb{N})[D \in L(s_i) \rightarrow C \notin L(s_{i+1})]\}$$

(d) temporal property

Figure 5.3 Combining feature models and temporal properties

Temporal properties are commonly used in model checking [183] to verify if a system exhibits a certain behavior, but can also be used to specify how a system should behave. Adding temporal properties to DSPL specification enables to constrain the executions of its underlying transition system to only those that satisfy the properties. Thus, by combining feature models with temporal properties, we can define with more precision the adaptive behavior of a DSPL.

As illustrated in Figure 5.2, from a feature model we can obtain a set of valid configurations and a corresponding fully-connected transition system. By combining this transition system with a temporal property we can obtain a transition system that is constrained by the property. Figure 5.3, illustrates the same feature model (Figure 5.3a) together with its corresponding configurations (Figure 5.3b) and transition system (Figure 5.3c) constrained by a temporal property (Figure 5.3d). The temporal property in Figure 5.3d includes all execution paths in which *a configuration that includes feature* `D` *is never followed by a configuration including feature* `C`. Figure 5.3c illustrates the obtained transition system, in which the transitions from configurations C5 and C6 (which include feature `D`) to configurations C3 and C4 (which include feature `C`) were removed by the constraint.

As discussed above, the semantics of transition systems and temporal properties are both defined by a set of infinite sequences of states ($[\![T]\!]_{TS} \subseteq S^\omega$ and $P \subseteq S^\omega$). Therefore, given a transition system T and a temporal property P over T, the intersection $[\![T]\!]_{TS} \cap P$ represents the set of execution paths in T that satisfy P. Hence, from a feature model

M and a set of temporal properties $\Phi \subseteq \mathcal{P}(S^\omega)$, we can build a transition system T' such that

$$[\![T']\!]_{TS} = \bigcap_{P \in \Phi} [\![DSPL(M)]\!]_{TS} \cap P.$$

That is a transition system in which all execution paths are infinite sequences of configurations for feature model M, and which conforms to all temporal properties in Φ.

Linear Temporal Logic

Linear temporal logic (LTL) [183] is one of the most commonly used formalisms for expressing temporal properties. LTL formulas are composed by combining atomic propositions of a transition system, boolean connectors such as $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ and temporal logical operators $\mathcal{U}$ (until), $\bigcirc$ (next time), $\Box$ (always) and $\diamond$ (eventually). Given $p \in AP$, the syntax of an LTL formula can be defined by the following grammar:

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid$$
$$\phi \,\mathcal{U} \phi \mid \bigcirc \phi \mid \Box\phi \mid \diamond\phi$$

Besides the standard boolean logic operators, LTL temporal operators can be derived from the $\bigcirc$ (next) and $\mathcal{U}$ (until) operators. The intuitive semantics of for $\bigcirc$ (next) operator is that $\bigcirc\phi$ holds at a given state in an execution path iff ϕ holds in the subsequent state in this same execution path. The formula $\phi \wedge \bigcirc\psi$ defines the ϕ should hold in the current state and ψ should hold in the next state. The until operator defines that a a proposition should hold until another becomes true. Therefore $\phi \,\mathcal{U}\, \psi$ holds at a given state if ϕ holds at this same state and at all the following states until a state in which ψ becomes true.

An LTL formula describes a temporal property. Hence, a transition system satisfies an LTL formula if all its execution paths comply with it. The semantics of LTL can be described by the satisfaction relation between an execution path π and an LTL formula. Let π be an execution path, we call $\pi^i = \langle s_i, s_{i+1}, s_{i+2}, ... \rangle$ a suffix of π starting at position i. The satisfaction relation, written as $\models$, between a path π and an LTL formula can

then be defined as follows:

$$
\begin{aligned}
&\pi \models \textbf{true} \\
&\pi \models p \qquad \text{iff} \quad p \in L(\pi(0)) \\
&\pi \models \neg\phi \qquad \text{iff} \quad \pi \not\models \phi) \\
&\pi \models \phi \vee \psi \qquad \text{iff} \quad \pi \models \phi \vee \pi \models \psi \\
&\pi \models \phi \wedge \psi \qquad \text{iff} \quad \pi \models \phi \wedge \pi \models \psi \\
&\pi \models \bigcirc\phi \qquad \text{iff} \quad \pi^1 \models \phi \\
&\pi \models \phi \mathcal{U} \psi \qquad \text{iff} \quad \exists i \geq 0 \text{ such that } \pi^i \models \psi \text{ and for all } 0 \leq k < i \text{ we have } \pi^k \models \phi
\end{aligned}
$$

Additional logical and temporal operators can then be derived as follows:

$$
\begin{aligned}
&\phi \rightarrow \psi && = \neg\phi \vee \psi \\
&\phi \leftrightarrow \psi && = (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi) \\
&\textbf{false} && = \neg\textbf{true} \\
&\diamond\phi && = \textbf{true}\, \mathcal{U} \phi \\
&\Box\phi && = \neg \diamond \neg\phi
\end{aligned}
$$

In this work, we employ LTL as a formalism for defining temporal constraints. When defining temporal constraints in a DSPL the set of atomic propositions is defined by the set of features in the feature model. Thus, an atomic proposition holds at a given state if the corresponding feature is selected in the corresponding configuration.

Using LTL we can express the temporal property presented in Figure 5.3d that *a configuration including feature* `D` *cannot be followed by a configuration including feature* `C` as follows:

$$\Box(\mathtt{D} \rightarrow \bigcirc\neg\mathtt{C}).$$

Actually, this constraint defines that at any point in time ($\Box$), if the system goes through a state in which `D` is selected, then `C` should not be selected in the subsequent ($\bigcirc$) state.

DSPL with Temporal Constraints

LTL can represent a subset of the class of ω-regular temporal properties. These properties correspond to the class of ω-regular languages, an extension of regular languages to infinite words. Therefore, for any LTL formula ϕ there is a language $\mathcal{L}(\phi)$ of infinite words, in which infinite words correspond to execution paths that satisfy the property.

As their regular counterparts, ω-regular languages are closed under union and intersection and can be accepted by corresponding automata for infinite words [183]. Hence, given two LTL formulas ϕ and ψ we have that $\mathcal{L}(\phi) \cap \mathcal{L}(\psi) = \mathcal{L}(\phi \wedge \psi)$. If we have a set of temporal constraints Ψ, we can obtain a single constraint $\phi = \bigwedge_{\psi \in \Psi} \psi$ such that $\mathcal{L}(\phi) = \bigcap_{\psi \in \Psi} \mathcal{L}(\psi)$.

From any LTL formula ϕ, we can also build a corresponding Büchi automaton $\mathcal{A}_\phi$ that accepts $\mathcal{L}(\phi)$ [183]. Büchi automata are an extension of finite automata for ω-regular languages and can be composed with a transition system to obtain another transition system that represents the intersection of both languages. Thus, given a feature model and a temporal constraint, we can define a DSPL transition system whose execution paths are sequences of feature model configurations that satisfy the temporal constraint.

Definition 15. Given a feature model M and a LTL formula ϕ, we can build DSPL with temporal constraints as $DSPL(M, \phi) = DSPL(M) \otimes \mathcal{A}_\phi$. That is, the synchronized product between the transition system for the DSPL defined by M, and the Büchi automata for ϕ [183, p. 200].

As discussed earlier, a feature model M defines a corresponding transition system $DSPL(M)$ in which there are transitions between any two possible feature model configurations. The goal of including temporal constraints is to limit these transitions and express the actual variability of the system over time.

By combining a feature model with temporal constraints we can better characterize the adaptive behavior of a DSPL in a compact way. Temporal constraints can be defined using well-known LTL logical formalism, which can be transformed into a Büchi automaton and combined with the feature model into a transition system.

5.2.3 Reconfiguration Operations

In the previous section we introduced temporal constraints to feature models in order to constrain a DSPL's adaptive behavior. Analysis of temporal properties and transition systems was strictly state-based and we did not consider any actions or labeling of system transitions.

However, as seen in the motivating example in Section 5.1, reconfiguring a cloud environment may require executing a set of operations such as migrating databases, restarting services, redeploying an application project, etc. In these cases, there is a relationship between an adaptation in a DSPL and a set of operations. Thus, when reconfiguring a cloud environment, we must take into account not only the possible transitions from the current system configuration but also the operations required to safely migrate the system from the current to a target configuration.

For modeling a DSPL with reconfiguration operations we rely on the definition of Doubly Labeled Transition Systems (L^2TS) [186], an extension of Kripke structures in which both states and actions can be labeled with propositions. In the literature, other similar notations such as labeled Kripke structures [187] and mixed transition systems [188] have also been used for state/action-based analysis.

Definition 16. A L^2TS is a tuple $T = (S, I, Act, R, AP, L)$ where S is a finite set of system states, I is the set of initial states, Act is a finite set of transition actions, $R \subseteq S \times 2^{Act} \times S$ is the transition relation, AP is a set of atomic propositions and $L : S \to 2^{AP}$ is a state-labeling function.

Definition 17. An execution path of T is an infinite sequence of states and sets of actions $\rho = \langle s_0, \alpha_0, s_1, \alpha_1, s_2, \alpha_2, s_3, ... \rangle$ such that $s_0 \in I$ and $(s_i, \alpha_i s_{i+1}) \in R$ for all $i \geq 0$.

This definition extends Kripke structures to support labeling over the transitions of a system. Thus a transition in a L^2TS is identified not only by a source and target states but also includes a set of actions. In the context of a DSPL these actions represent reconfiguration operations required to adapt the system from a source to a target configuration.

State/Event Linear Temporal Logic (SE-LTL) [187] is an extension of LTL to support expressing temporal properties over states and actions. SE-LTL is very similar to LTL, but supports temporal logic formulas over state and actions propositions. Given $p \in AP$

and $a \in Act$, the syntax of an SE-LTL formula is defined by the following grammar:

$$\phi ::= p \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid$$
$$\phi \, \mathcal{U} \phi \mid \bigcirc \phi \mid \Box\phi \mid \diamond\phi$$

Semantics differ slightly as an SE-LTL property may also include constraints over actions. In these cases, an action holds at a given state in an execution path if it is part of the transition immediately after it. Thus, given an execution path $\rho = \langle s_0, \alpha_0, s_1, \alpha_1, s_2, ... \rangle$, action $a \in Act$ holds at state i iff $a \in \alpha_i$. Note that the transition relation $R \subseteq S \times 2^{Act} \times S$ is defined over the power set of actions and each transition can have multiple actions. The semantics of logical and temporal operators is the same as in LTL.

Using the SE-LTL we can express constraints from the motivating example given in Section 5.1.1 that involve operations and features. For instance, we could express the rule that changing the geographical region requires migrating the application as follows:

$$\Box(((\texttt{US} \wedge \bigcirc\texttt{EU}) \vee (\texttt{EU} \wedge \bigcirc\texttt{US})) \leftrightarrow \texttt{MigrateApp})$$

This constraint defines that whenever the location feature is changed from `US` to `EU` or vice-versa, the `MigrateApp` operation should be executed. Therefore, by employing SE-LTL we can define temporal and propositional constraints over features and reconfiguration operations.

Considering temporal constraints and reconfiguration operations we can redefine the syntax and semantics of DSPLs based on a feature model.

Definition 18. Given a feature model $M = (\mathcal{F}, C)$ and a set of reconfiguration operations OP, we can define the corresponding DSPL as a doubly labeled transition system such that $DSPL(M, OP) = (S, I, Act, R, AP, L)$, where $I = S = C$, $Act = OP$, $R = S \times 2^{OP} \times S$, $AP = \mathcal{F}$ and $L(s) = s$.

This definition extends DSPLs based on a feature model with a set of reconfiguration operations. The obtained transition system allows transitioning between any feature model configuration by executing any combination of reconfiguration operations. As in the case of LTL, an SE-LTL formula can also be used to build a Büchi automaton [187] that can be combined with the transition system obtained from the feature model and reconfiguration operations to limit the allowed transitions.

Definition 19. Given a feature model M, a set of reconfiguration operations OP and a SE-LTL formula ϕ, we build a transition system $DSPL(M, OP, \phi) = DSPL(M, OP) \otimes \mathcal{A}_\phi$.

That is the synchronized product, as defined in [187], of the transition system obtained from the feature model and reconfiguration operations with the Büchi automaton that accepts the property defined by the formula ϕ.

Introducing temporal constraints on DSPLs enables us to express detailed constraints on how variability can be applied to the system over its lifetime. Reconfiguration operations allow describing and analyzing the different ways of effecting a reconfiguration, as well as making explicit the constraints between features and operation. Together these constructs support modeling adaptation constraints in cloud environments that could not be represented by variability models.

5.3 Automated Analysis of Temporal Constraints

The previous section presented how we can represent a DSPL as a transition system, and how temporal constraints can be combined with feature models to improve the representation of the adaptive behavior of a DSPL. In this section we discuss implementing a reasoner for identifying a reconfiguration plan given a reconfiguration request.

5.3.1 Reconfiguration Query

Adaptations in a DSPL are usually triggered by requests to include or remove a set of features from the current configuration, or by changes in monitored context features. In both cases, we have a current system configuration and a change request that specifies which features should be included or excluded. Given such a request, we want to identify the set of possible reconfigurations that will move the system to a target state that conforms to the requested change.

Definition 20. A reconfiguration query $q = (A, E)$ is defined by a set A of features that should be included in the target configuration and a set E of features that should not be included in the target configuration.

Given a doubly-labeled transition system $T = (S, I, Act, R, AP, L)$, its current state s and reconfiguration query $q = (A, E)$, we can define the set of possible reconfigurations as a function

$$N_T(s, q) = \{(\alpha, s') \mid (s, \alpha, s') \in R \ \wedge \ I \subseteq s' \ \wedge \ E \cap s' = \emptyset\}.$$

A reconfiguration includes a target state s' and a set of reconfiguration actions α for moving the system to s'. The reconfigurations returned by function N_T are those for which the target states and reconfiguration actions comply with the reconfiguration query.

In the case a reconfiguration query specifies a full configuration (i.e. $A \cup E = \mathcal{F}$), the function N_T can be used to verify if this configuration is reachable from the current system configuration.

5.3.2 Symbolic Representation

From a system current configuration and reconfiguration query, we are interested in finding one or many possible target configurations. Given a representation of the DSPL as a transition system, we can easily identify the possible reconfigurations by looking at the transitions outgoing the current system state and filtering those that agree with the reconfiguration query. However, building an explicit representation of a transition system for a DSPL is often unfeasible due to the large number of possible states and transitions.

One approach to deal with this problem is to build a symbolic representation of the transition system. This approach is widely used in the model-checking community to deal with the recurrent problem of state explosion. The idea behind symbolic representation is to represent the set of system states and the transition relation as propositional formulas. Set operations can then be implemented as logical operations between propositional formulas and set membership by solving satisfiability. Thus, the set of states in a transition system can be defined by a propositional formula over the set of atomic propositions using substantially less space than the number of possible states.

Given a transition system $T = (S, I, Act, R, AP, L)$, its transition relation $R \subseteq S \times 2^{Act} \times S$ can also be represented by a propositional formula over the propositions in $AP \ \cup \ Act \ \cup \ AP'$. Here we have $AP' = \{p' \mid p \in AP\}$ as a copy of atomic propositions

that represent the labeling in a target state, while AP represent the labeling of a source state and Act the set of transition actions.

For each state in a transition system, the labeling function $L : S \to 2^{AP}$ defines exactly what are the atomic propositions that hold at this given state. Therefore, for any state $s \in S$ in a transition system, we may also define a propositional formula that uniquely identifies this state as

$$\widetilde{L(s)} = (\bigwedge_{p \in L(s)} p) \wedge (\bigwedge_{p \in (AP \setminus L(s))} \neg p).$$

Likewise, every transition (s, α, t) in the transition relation $R \subseteq S \times 2^{Act} \times S$ defines a set of transition actions $a \in \alpha$. For any set of transition actions, we may define a propositional formula that uniquely identifies it as

$$\widetilde{L(\alpha)} = (\bigwedge_{a \in \alpha} a) \wedge (\bigwedge_{a \in (Act \setminus \alpha)} \neg a).$$

Relying on these functions that uniquely identify a set of transition actions and a state, we can define a transition relation by the following propositional formula, in which $\widetilde{L(t)}'$ represents the propositional formula that uniquely identify a state t applied to primed variables that represent the in a target state

$$\widetilde{R_T} = \bigvee_{(s,\alpha,t) \in R} \widetilde{L(s)} \wedge \widetilde{L(\alpha)} \wedge \widetilde{L(t)}'.$$

Once we have propositional formulas that represent the transition system and a given state $s \in S$, the set of outgoing transitions from s can be obtained by the propositional formula

$$\widetilde{N_T(s)} = \widetilde{R_T} \wedge \widetilde{L(s)}.$$

Based on this we can obtain a propositional formula that represents the set of transitions that can be obtained by a reconfiguration query. Given a reconfiguration query $q = (A, E)$ and current state s, we can obtain the propositional formula

$$\widetilde{N_T(s,q)} = \widetilde{N_T(s)} \wedge (\bigwedge_{p \in A} p') \wedge (\bigwedge_{p \in E} \neg p').$$

that defines the set of transitions from state s that agree with the conditions defined by reconfiguration query q. This propositional formula is a symbolic representation of the

function N_T presented in Section 5.3.1. By solving satisfiability over the formula we can find the set of possible reconfigurations complying with the query.

So far, we demonstrated how we can represent a transition system symbolically by a propositional formula and use satisfiability solving to find a set of transitions that support a given query. However, we are not interested in building a symbolic representation from a complete transition system, but directly from a feature model extended with temporal constraints and reconfiguration operations.

A feature model can also be regarded as a symbolic representation of the set of valid configurations for a system as it is a compact notation for a large number of configurations. A feature model is also often represented as a propositional formula [109, 189], for which the satisfiable assignments represent the set of valid configurations.

As mentioned earlier, if we do not consider temporal constraints, a DSPL defined by a feature model can transition between any two valid configurations. So, given a feature model M, we have a corresponding propositional formula $\widetilde{M}$ that represents the set of allowed configurations $[\![M]\!]_{FM}$. Then, we can define the transition relation of its corresponding DSPL transition system by the propositional formula

$$\widetilde{R_{FM}} = \widetilde{M} \wedge \widetilde{M}'.$$

LTL and SE-LTL formulas can also be represented as propositional formulas and there are known methods for translating temporal logic formulas into equivalent symbolic representation [190–192]. By employing the *tableau* method [190] we obtain from a temporal constraint ϕ, two propositional formulas that represent a set of initial states $\widetilde{I_\phi}$ and a transition relation $\widetilde{R_\phi}$. These formulas are also defined over the features at the source and target configuration and reconfiguration operations. Additional propositions may be introduced by the translation to keep extra state information required for constraints that involve the $\mathcal{U}$ (until) operator or its derivatives.

Thus, from a feature model M and a temporal constraint ϕ we can obtain the following propositional formula that represents the set of initial states of the transition system

$$\widetilde{I_{M\phi}} = \widetilde{M} \wedge I_\phi.$$

Likewise, we can obtain the a propositional formula that represents the transition relation for the DSPL

$$\widetilde{R_{M\phi}} = \widetilde{M} \wedge \widetilde{M}' \wedge R_{\phi}.$$

Once we have a representation of the transition relation we can define the propositional formula for the function $N_T(s,q)$ as follows

$$\widetilde{N_T(s,q)} = \widetilde{L(s)} \wedge \widetilde{R_{M\phi}} \wedge (\bigwedge_{p \in A} p') \wedge (\bigwedge_{p \in E} \neg p').$$

Symbolic representation of transition systems enables to express the sets of states and the transition relation as a propositional formula. Given a feature model and LTL formulas for a DSPL, we can build a symbolic representation for its underlying transition system. Using propositional logical operators, we can build a propositional formula for answering a reconfiguration query. We can then identify a set of complying reconfigurations from a current state by solving satisfiability.

5.4 Cardinality-Based Feature Models: the Case of Cloud Computing

Up to now, we discussed about the use of temporal constraints in DSPLs where variability is defined by boolean feature models. In these systems a configuration is identified by a set of active features and adaptations are represented by the inclusion or removal of features in the current system configuration.

When dealing with Cardinality-Based Feature Models (CBFMs) such as the ones used to model variability in cloud providers, a system configuration is defined by a tree of feature instances. In this case, a feature may be included multiple times in the system configuration at different places in the hierarchy and adaptations are represented as modifications in this feature instance tree. Nevertheless, it can still be viewed as a transition system in which each state corresponds to a system configuration and transitions that correspond to adaptations.

Temporal constraints based on LTL formulas are defined over the atomic propositions used in the labeling of states of a transition system. While labeling is straightforward in boolean feature models, it becomes less clear when feature cardinalities come into play. As multiple instances can be included in a configuration, we need means for expressing

temporal dependencies that exist not only at the feature, but also at the instance level. However, defining constraints directly over individual instances of features may not be desirable nor feasible due to the potentially large number of feature instances. This calls for a method that supports expressing temporal constraints at the instance level but using a concise notation. This issue is closely linked to what we find when expressing additional cross-tree constraints in CBFMs and therefore we rely on the same solutions that were previously employed for additional constraints in feature models with relative cardinalities (see Section 4.2.3).

5.4.1 Context Feature and Constraining Expressions

In Figure 5.4, we present a cardinality-based feature model derived from the feature model shown in Figures 5.2a and 5.3a. Here, mandatory and optional features had their feature cardinalities assigned to the cardinality ranges 1..1 and 0..1 respectively. Besides this, the only change in this feature model is in the feature cardinality of `B`, which allows for up to 10 instances.

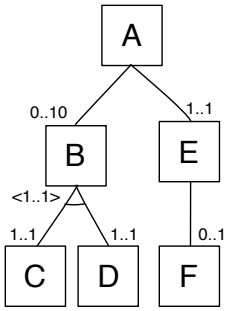


Figure 5.4 Sample feature model with cardinalities

In Section 5.2, we have illustrated how to specify the constraint that *a configuration that includes feature `D` cannot be followed by a configuration including feature `C`* using the constraint $\Box(\mathtt{D} \rightarrow \neg \bigcirc \mathtt{C})$. In clone-enabled feature models the interpretation of temporal constraints based only on features may lead to ambiguities similar to those found in cross-tree additional constraints (see Section 4.2.3). For instance, in the given example, does the proposition `D` evaluates to true when at least one instance of `D` is included in the current system state? When all existing instances of `B` include `D` in its subtree? Or still, should it be evaluated individually for each instance of `B`?

To deal with these differences and support modeling temporal constraints on cardinality-based feature models, we leverage the constructs introduced for modeling additional

cross-tree constraints in Section 4.2.3. A temporal constraint is annotated with a *context feature* that defines the scope in which it should be evaluated while *constraining expressions* are used as atomic propositions.

For the example given in Figure 5.4, using these constructs we can define the constraint that *an instance of feature* `B` *that includes* `D` *cannot be reconfigured to include* `C` as follows

$$<\texttt{B}>\ \Box([1..1](\texttt{D},\texttt{B}) \rightarrow \neg \bigcirc [1..1](\texttt{C},\texttt{B})).$$

Here, the context feature `B` specifies that this constraint should be evaluated individually for each instance of feature `B`. Meanwhile, constraining expressions $[1..1](\texttt{D},\texttt{B})$ and $[1..1](\texttt{C},\texttt{B})$ define a predicate over the number of features instances of `D` and `C` in relation to feature `B`.

Actually, this formula is a shorthand for a set of formulas defined over each individual instance of feature `B`. In this example, a configuration can have at most 10 instances of feature `B`. Thus, this temporal constraint actually describes 10 different formulas, each one applied to a possible instance of feature `B`. The same idea applies to constraining expressions that are part of a temporal constraint. For example, a constraining expression such as $[1..1](\texttt{D},\texttt{B})$ will lead to 10 different atomic propositions in the corresponding transition system. The semantics of constraining expressions are the same as defined in Section 4.2.3.

As we did for relative cardinalities we also use a simplified notation when referring to constraining expressions that only verify the presence or absence of a feature instance. Thus, the above mentioned example cloud be rewritten as

$$<\texttt{B}>\ \Box(\texttt{D} \rightarrow \neg \bigcirc \texttt{C}).$$

A similar example arises in the case of Heroku PaaS platform shown in Figure 5.1. As discussed in the motivating example (see Section 4.1.1), a configuration of a cloud environment in Heroku can be composed of multiple application projects. Each application can include multiple application services based on different programming technologies and using different additional cloud services such as databases, caching, message queues, etc. In this case, temporal constraints apply only in the context of an application project or cloud service. If we take as an example, the case of the MySQL restriction on downgrading a database plan, it applies individually to each instance of a MySQL database. Therefore, if a system configuration includes two instances of `ClearDBMySQL`,

one with setup with the `M4` plan, it does not imply that the other instance using the plan `M2` is forbidden to update to plan `M3`. Based on these concepts, we can redefine the MySQL constraint as follows:

$$<\texttt{MySQL}> \Box(\texttt{M2} \rightarrow \neg \Diamond \texttt{M1})$$

This constraint states that for each instance of `MySQL` feature (i.e. the context feature), once an instance of `M2` is included under the subtree of this same `MySQL` instance, an instance of `M1` cannot be eventually included in the subtree of this `MySQL` instance for any future configurations.

Additionally, we introduce the operators *switch* and *change* that can be used to simplify the definition of constraints involving the (de)activation of a feature instance or a change in the selected features from a feature group. Given a feature $f \in \mathcal{F}$, the *switch* operator S can be defined as follows:

$$S(f) = \neg(f \leftrightarrow \bigcirc f).$$

Given a feature group $g \in \mathcal{G}$, the *change* operator C can be defined as follows:

$$C(g) = \bigvee_{(f,g) \in E} S(f).$$

Using these operators, we can define the constraint that *when feature `F` is activated it is not possible to change the currently selected alternative (`C` or `D`) for any instance of feature `B`* as follows

$$<\texttt{B}> \Box(\texttt{F} \rightarrow \neg C(\texttt{B})).$$

The same ideas employed for features, apply for constraints involving reconfiguration operations. When handling feature models with instances it is important to identify the feature instance in which the operation is to be applied. The context feature is used to indicate the feature instance in which the operation is to be executed. For instance, following constraint, defines that when the location of an application changes, the `MigrateApp` reconfiguration operation needs to be executed in this same `Application` instance.

$$<\texttt{Application}> \Box(\ C(\texttt{Location}) \leftrightarrow \texttt{MigrateApp(Application)}\ )$$

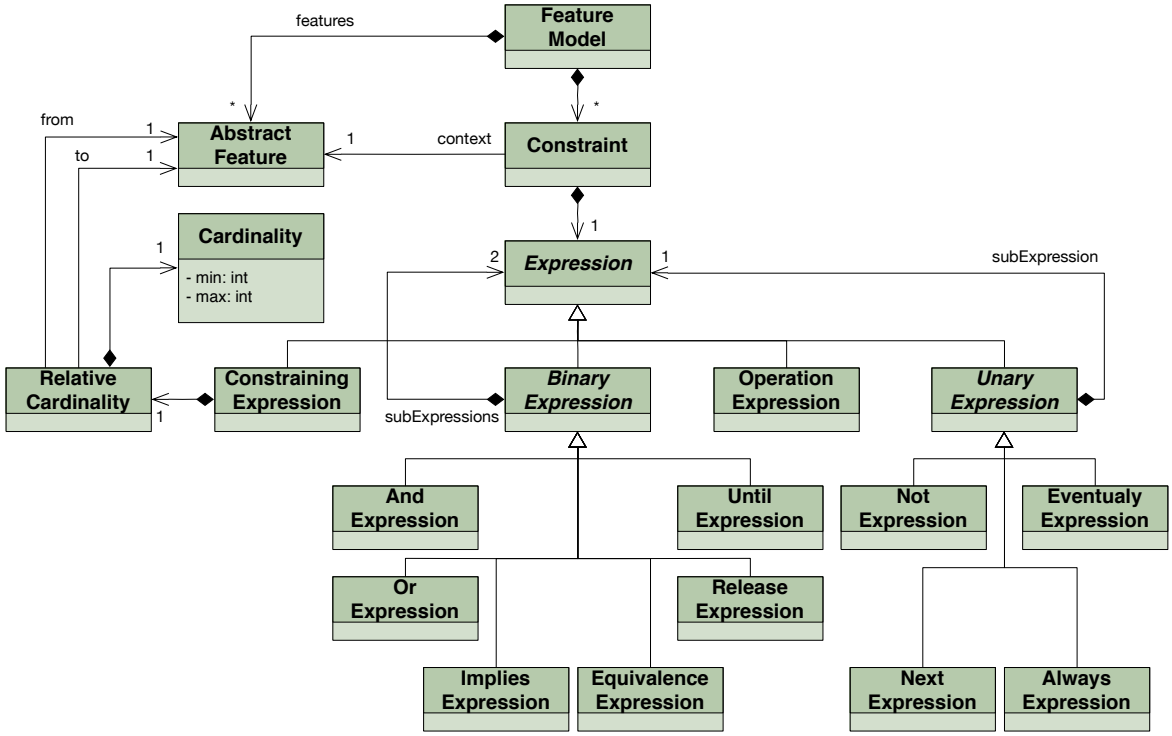


Figure 5.5 Metamodel for Feature Modeling: `Expression` hierarchy

As in the previous case, this constraint is a shorthand for a conjunction of LTL formulas intervening on each possible `Application` instance. Figure 5.5 illustrates the metamodel for defining temporal constraints in feature models with relative cardinalities. This updates the metamodel introduced in Figure 4.5 by updating the `Expression` class hierarchy, allowing to combine both temporal and logical operators in additional feature model constraints.

5.4.2 Analysis of Temporal Constraints

As discussed earlier, adaptation in a DSPL is triggered by a reconfiguration request that defines the expected properties for the target configuration. From this request, we are interested in finding a reconfiguration towards a satisfying configuration. A reconfiguration includes both a target configuration and a set of reconfiguration operations required to adapt the system to the target configuration.

In boolean feature models a reconfiguration request was defined as a query $q = (A, E)$ that specifies features to be present or absent in the target configuration. However, when working with cardinality-based feature models, configurations are not defined

by a selection of features but by a tree feature instances. In this case, we define a reconfiguration request as a partial configuration for the feature model, as described in Section 4.3.4. A partial configuration for cardinality-based feature models is defined by an incomplete tree of features instances. As in the case of boolean feature models, a partial configuration specifies the expected properties of the target configuration. Likewise, our goal is to find reconfigurations that satisfy these properties.

To support reasoning over CBFMs extended with temporal constraints, we apply the same principles of symbolic representation described in Section 5.3. CBFMs are translated into constraint programming (CP) constructs using the method described in Section 4.3.3. As explained in Section 4.3.3, this method creates variables that represent the relative cardinalities in a configuration. As temporal constraints in CBFMs are defined using constraining expressions over relative cardinalities, they can be directly mapped to constraints over existing variables in the corresponding constraint program. Because of this property, we can use the method described in Section 5.3.2 to translate a temporal constraint into a CP constraint that represent the transition relation by mapping the atomic propositions to their respective representation in the constraint program.

Therefore, by combining a CBFM and a temporal constraints we obtain a constraint programming problem that defines the transition relation of the corresponding transition system. When analyzing a reconfiguration request, we can translate it into extra constraints using the method described in Section 4.3.4 and assign constant values to the variables that represent the current system configuration. By solving satisfiability over the constraint program, we can find the possible reconfigurations.

To illustrate how these concepts are applied we present an example based on the Heroku PaaS feature model depicted in Figure 5.1. From the feature model hierarchy we obtain a set of CP variables that represent feature instances and cardinalities. For example, for each possible instance of feature `Application`, we will have a boolean variable $\texttt{Heroku}_1\texttt{Application}_i$ that represents the inclusion of i-th instance in the configuration. Meanwhile, a numeric variable $\texttt{Heroku}_1\texttt{Application}$ represents the total number of applications included in the root instance of feature `Heroku` identified by $\texttt{Heroku}_1$. A relative cardinality is also represented as a numeric variable, thus the variable $\texttt{Heroku}_1\texttt{Application}_i\texttt{Dyno}$ represents the total number of instances of feature `Dyno` in the subtree of $\texttt{Application}_i$.

Once, we have a constraint programming representation of the CBFM we can rewrite our temporal constraints directly based on these CP variables. Let us consider the following

temporal constraint:

$$<\texttt{Application}>\ \Box((\texttt{US} \wedge \bigcirc\texttt{EU}) \leftrightarrow \texttt{MigrateToEU})$$

The *context feature* indicates that this constraint should be added to each `Application` instance. Thus the propositions `US` and `EU` stand respectively for the *constraining expressions* $[1..1](\texttt{US}, \texttt{Application})$ and $[1..1](\texttt{EU}, \texttt{Application})$. For each application instance, we can then obtain the corresponding CP constraints for each constraining expression. For example, the constraining expression $[1..1](\texttt{US}, \texttt{Application})$ can represented as the CP constraint: $1 \leq \texttt{Heroku}_1\texttt{Application}_i\texttt{Location}_1\texttt{US} \leq 1$. The variable $\texttt{Heroku}_1\texttt{Application}_i\texttt{Location}_1\texttt{US}$ represents the number of instances of feature `US` under the i-th instance of feature `Application`. Other constraining expressions can be translated using the same approach. Similarly, reconfiguration operations are mapped to boolean variables that uniquely identify the execution of the operation in a feature instance. Thus, the `MigrateApp` proposition will be translated into a boolean variable $\texttt{MigrateApp}(\texttt{Heroku}_1\texttt{Application}_i)$. Putting everything together, the example constraint will translated into the following expression

$$\begin{aligned}\Box(\ &(1 \leq \texttt{Heroku}_1\texttt{Application}_i\texttt{Location}_1\texttt{US} \leq 1) \\ &\qquad \wedge \bigcirc(1 \leq \texttt{Heroku}_1\texttt{Application}_i\texttt{Location}_1\texttt{EU} \leq 1) \\ &\leftrightarrow \texttt{MigrateToEU}(\texttt{Heroku}_1\texttt{Application}_i)\ )\end{aligned}$$

From this formula, we employ the same methods used in boolean feature models to convert an SE-LTL formula to a symbolic representation of the transition relation. We then obtain a CP constraint over variables that represent the system at the source and target state as well as reconfiguration operations. For example, for each application instance, we will have variables $\texttt{Heroku}_1\texttt{Application}_i\texttt{Location}_1\texttt{US}$ and $\texttt{Heroku}_1\texttt{Application}_i\texttt{Location}_1\texttt{US}'$ that identify if the i-th application is located in US at the current and following state respectively. The same process is applied for all the temporal constraints to obtain a representation of the transition system as a constraint program.

5.5 Discussion

The combination of variability modeling and temporal constraints allow to more precisely define the adaptive behavior of DSPLs. This is specially useful in domains in which

the system cannot freely transition between any valid configuration and adaptations are expected to follow a predefined order.

By using model checking theory and symbolic representation, our approach maps temporal constraints to propositional logic formulas, which are also extensively used in the feature model analysis community and can be easily mapped to analysis approaches based on SAT or CP solvers. This facilitates integrating temporal and variability reasoning, and limits the overhead added by the approach.

Still, temporal constraints employed in this work cannot describe all kinds of temporal properties, but only a subset of regular temporal properties that can be defined as LTL formulas. This approach also only applies to cases in which the adaptive behavior of a DSPL is known before hand, at design time, and it does not handle the problems related to dynamically adding variants or dynamically evolving the variability at runtime [138, 139].

Though we mention that reconfiguration operations may have an impact on performance, downtime and costs of adaptations, this work does not handle nor take this impact into account during the analysis of reconfiguration requests. In [33], we employed Pseudo-Boolean SAT solvers to handle cost-based reconfiguration requests, however this work was limited to boolean feature models.

A similar approach [193] has also been employed in the specification of reconfigurable component-based systems. As in our work, temporal logic is used to declaratively define reconfiguration rules of a component-based systems.

5.6 Summary

In this chapter we introduced temporal constraints to feature models to support the definition of the adaptive behavior of Dynamic Software Product Lines for cloud environments. We integrated temporal constraints and reconfiguration operations into the definition of DSPLs to express complex constraints that arise during the lifecycle of adaptive cloud environments. Using concepts from model checking, we have built a symbolic representation for a DSPL that uses boolean satisfiability to solve reconfiguration queries over DSPLs.

Based on these findings, we have described how this approach can be extended to handle cardinality-based feature models, thus supporting temporal constraints on the lifecycle of feature instances. This approach enables defining many of the constraints that arise in the

reconfiguration of cloud environments that were not supported by previous approaches in DSPL management.

Temporal constructs enable handling many of the reconfiguration constraints that arise during the adaptation of cloud environments which were not supported by previous DSPL approaches. This contribution answers RQ_2, outlined in Section 1.2, that is, we show that we can use temporal properties to handling reconfiguration constraints in feature models.

In the following chapter, we present an approach that integrates the first two contributions on variability management in order to automate the setup and adaptation of multi-cloud environments.

Chapter 6

Automated Setup and Adaptation of Multi-Cloud Environments

A multi-cloud environment is defined by a selection of cloud services provisioned from multiple clouds in order to support the execution of an application. By combining services from multiple providers, cloud customers aim to reduce their dependence on any single cloud provider, to comply with application requirements , and to optimize the performance, availability and costs of running their application. However setting up and managing a multi-cloud environment is a very complex task. Due to the wide range of available cloud providers and heterogeneity among service offerings, it is very difficult for cloud customers to select providers and services that are suitable for their applications. In addition, customers need to be aware of the complex rules that govern how cloud providers can be configured in order to make an informed choice and build a consistent environment.

In Chapters 4 and 5 we introduced extensions to feature modeling to manage the variability found in the configuration of cloud computing environments. This variability is expressed by the many available cloud services offered by a cloud provider and the rules that govern how they can be combined into a cloud environment. Managing the variability in cloud environments allows us to ensure that a configuration complies with the provider's rules.

However, when dealing with multi-cloud environments, managing variability in the configuration of cloud environments is not the only challenge. Cloud providers have very heterogeneous service offerings. Each provider offers a different set of cloud services based on different concepts, and even for comparable services, substantial differences can be

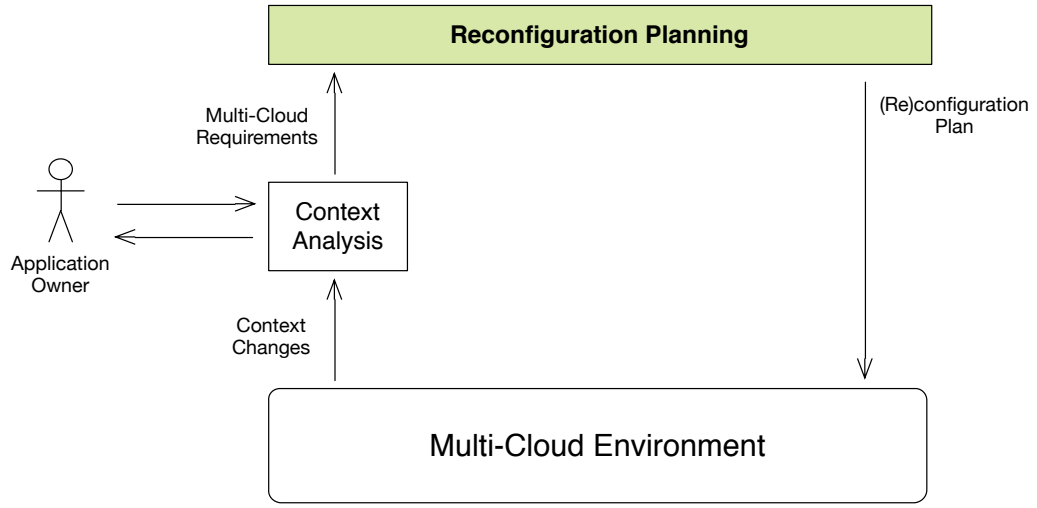


Figure 6.1 Vision of an adaptive multi-cloud environment.

found in their functionality. Though variability management assists in the configuration of cloud environments, it is still very complex for developers to go through different provider offerings and find which features in each provider fit their needs. Moreover, as the application's context or requirements change, the multi-cloud environment may need to be updated accordingly to cope with new application needs.

The lack of support for managing this complexity is a limiting factor for the adoption of multi-cloud computing. To overcome this difficulty, we look towards an approach for automatically building adaptive multi-cloud environments, capable of reconfiguring themselves to match the evolution of the application's needs. This requires dealing with a large set of concerns such as monitoring heterogeneous cloud providers and application services; analyzing context changes and identifying new requirements for the multi-cloud environment; matching these requirements against available cloud services to build a configuration or reconfiguration plan that complies with cloud provider rules; and enacting this plan using the heterogeneous reconfiguration mechanisms and interfaces offered by providers. In this work, we deal only with the problems related to the generation of proper configurations or reconfiguration plans from a description of requirements for a multi-cloud environment.

Figure 6.1 illustrates this vision of an adaptive multi-cloud environment and how this work integrates into it. In this vision, application owners such as developers or IT staff can define the requirements for a multi-cloud environment that supports their application and obtain a configuration that can be used to build a multi-cloud environment. Once the multi-cloud environment is running, context changes are analyzed and mapped to new

requirements for the multi-cloud environment. The planning mechanism, which is the focus of this work, is responsible for creating a proper configuration or reconfiguration plan that describes how to setup or adapt a multi-cloud environment to support the defined requirements. The application owner may intervene in the process by manipulating the multi-cloud environment requirements or choosing when to apply a reconfiguration.

Getting from multi-cloud environment requirements to a proper (re)configuration requires dealing with both the variability and the heterogeneity across service offerings, as well as the complex constraints that govern the reconfiguration of cloud providers. To deal with these problems, we propose an approach that integrates concepts, techniques and tools from model-driven engineering and ontology reasoning together with our previous contributions on managing variability in cloud environments (Chapters 4 and 5) to automate the setup and adaptation of multi-cloud environments. The goal is to get from a high-level description of multi-cloud environment requirements to a proper (re)configuration, which complies with all cloud providers' (re)configuration rules. By doing so, we aim to build adaptive multi-cloud environments that are capable of automatically reconfiguring themselves to support the requirements of an application. The proposed approach provides a domain specific modeling language for describing multi-cloud environment requirements and a method for mapping requirements to provider-specific features. Our previous contributions on the automated analysis of feature models are employed to generate compliant (re)configurations.

In Section 6.1 we give a quick retrospect on the motivations for multi-cloud computing, presenting a motivating example and the challenges to automate the setup and adaptation of multi-cloud environments. In Section 6.2 we present an overview of the approach and its main elements. Section 6.3 gives more details on the definition of multi-cloud environment requirements, while Section 6.4 describes the mapping from requirements to cloud provider features. Sections 6.5 and 6.6 detail the process for obtaining a configuration or reconfiguration plan for a multi-cloud environment from its requirements. Finally, we discuss the implications of this work in Section 6.7 and the conclusions in Section 6.8.

6.1 Motivation

As discussed in Chapter 2, consuming services and resources from multiple clouds has been motivated by multiple reasons that can be classified into three broad categories.

- *Reducing vendor dependence.* This includes protecting applications from cloud outages or failures, but also avoiding vendor lock-in;
- Complying with requirements, constraints or regulations. When a single cloud does not satisfies all of the applications requirements and constraints combining clouds becomes mandatory;
- Optimizing performance and costs. As different cloud providers offer specific advantages, combining them can improve QoS or costs.

This means that multi-cloud computing can be used to achieve different goals, according to application requirements. Even for a single application, each application service may have different requirements concerning their desired level of availability, scalability, provider dependence, among others, which will lead to different uses of multiple clouds.

However, building a multi-cloud system is a very complex task due to the large number of factors that must be considered. When building a multi-clouds system, customers must consider factors such as functional and non-functional requirements of application services as well as cloud provider configuration options, pricing policy, data center location, availability levels, etc. On the one hand, cloud providers employ different concepts for their offerings and are subject to intricate configuration rules, making it difficult to establish a comparison between competing services. On the other hand, each application service that is part of an application may be developed by a different team, written in a different programming language, using different application frameworks or databases, and may have different scalability and availability requirements. Hence, each application service will require different functionality from cloud providers and a cloud provider that supports one of the application's services might not support another.

Besides this, as application requirements change over time, the system must be re-configured accordingly. Mechanisms for reconfiguring a cloud environment are also heterogeneous across providers. In addition to the factors involved in the construction of multi-cloud systems, the adaption of a multi-cloud system must also guarantee that the system can safely transition from the current configuration to a new configuration that supports the new requirements.

While some existing approaches in the literature deal with different aspects of the development and management of multi-cloud systems [12–14, 11, 15, 78, 16–21], they do not take into account the heterogeneity in their service offerings and the variability in their configuration. Nevertheless, works on managing the variability in the configuration of cloud providers [26, 23] do not deal with the issues related to multi-cloud environments.

In the remainder of this section we present an example that illustrates the complexities associated to the setup and adaptation of a multi-cloud environment as well as the challenges that must be tackled to achieve it.

6.1.1 Motivating Example

To illustrate the complexities faced in order to build and manage a multi-cloud environment, we will consider a scenario of an example e-commerce application for which we plan to use multiple clouds in order to reduce provider dependence, improve its reliability and comply with data location policies. Below, we describe in more detail the requirements for this sample application and demonstrate somes examples of how the heterogeneity and variability in cloud providers complicate the configuration of multi-cloud environments.

Multi-cloud environment requirements

Our example e-commerce application is composed of the following services:

- a *product catalog* that keeps a record of product information and inventory;
- a *recommendation service* that records purchases and recommends new products for customers;
- a *user management* service that manages a *user database* and a *credit card database*;
- an *order management* service that tracks orders and their statuses;
- a *payment* service responsible for interacting with banking and credit card partners;
- a *web frontend* and a *mobile API gateway* to provide access to end users' devices.

Each of these services may be developed by different teams, employing different methods and technologies. This implies that each application service may require a different set of functionality from a cloud provider. For instance, if the *web frontend* is designed as web application based on Java web standards, it can be directly deployed to any Java web container such as Tomcat, Jetty or JBoss. Meanwhile, if the *recommandation* service is designed in a way that requires direct access to operating system functions, a virtual machine will be required. In addition, according to its development process, a service may also require extra tools for continuous integration, issue tracking, staging resources, etc. Hence, the methods and technologies employed in the development of an application

service will determinate which cloud providers support it and which cloud services can fulfill its requirements.

Throughout this chapter, we call the functionality required by an application service as *application service requirements*. However, it is important to point out that it does not refer to the requirements for the application service itself, but rather the requirements for a cloud environment in order to support the deployment and execution of the application service.

In addition to application service requirements, this example application should also meet the following conditions:

- data concerning European customers should be stored in a country that is a member of the European Union;
- *web frontend*, *API gateway* and *product catalog* services should run on to at least two different providers to ensure these essential services are always available, even in the case of a provider failure;
- *product catalog* should scale up automatically to cope with the application demand;
- the *credit card database* should be kept in a private cloud controlled by the company.

These are examples of *multi-cloud requirements*. We call multi-cloud requirements, the non-functional requirements associated to how application services can be distributed across different clouds in order to reduce provider dependence, increase reliability or comply with location constraints. These requirements are directly related to the motivation behind the use of multiple clouds as discussed in Section 2.2.1.

Apart from these requirements, there are no further restrictions on application services. Thus, when setting up a multi-cloud environment to deploy this application, we will be looking to optimize costs or improve the quality of service while complying with application requirements and constraints.

Together, application service and multi-cloud requirements define the requirements for a multi-cloud environment that can be used to deploy and run the example application. Multi-cloud environment requirements represent the application needs in terms the functionality required from cloud providers and the constraints that must be followed in order to support the execution of the application. We employ the term *application needs* instead of application requirements to highlight that these are not the requirements for the application itself, but rather the requirements for a multi-cloud environment that

supports the execution of the application. That is what the application needs from cloud providers.

From such a high-level description of application needs we want to find a selection of cloud services from multiple clouds that supports application service requirements while complying with multi-cloud requirements.

Heterogeneity across cloud providers

One of the complexities that arise when building a multi-cloud environment is the heterogeneity across cloud providers offerings. For example, if we are looking for a cloud service that supports the *web frontend* application service in the given example, we may find out that across different providers, support for Java applications is not the same. For instance, in the Heroku platform provider, support for Java application stands for a containerized executing environment with Java Development Toolkit (JDK) and Maven installed. As our *web frontend* was designed to run on a managed Java web container (e.g., Tomcat, Jetty, JBoss, etc), the developer would have to set up a web container before deploying the application service or provide a standalone application with an embedded web container. Meanwhile, the OpenShift platform provider offers a set of different Java Application Containers such as Tomcat, JBoss and Glassfish as the runtime environment for Java applications. This means that Java web applications can be directly deployed to OpenShift. However, it also implies that standalone Java applications that require to run on an unconstrained environment are not supported by the OpenShift Java service.

When looking for candidate cloud services in order to deploy an application service, developers have to consider their heterogeneity and verify if they match the specific needs of the application service at hand. For the given example of the *web frontend* service, if the developer is ready to provide means for setting up a Java web container, both Heroku and OpenShift are valid candidates. Otherwise, only the OpenShift provider will directly support the application service requirements. On the other hand, if they want to deploy a standalone Java application to OpenShift, they must provide means for setting up an execution environment for Java applications from a barebones container.

According to their goals and the availability of automation tools, developers may be interested in providing multiple ways for deploying their application in order to have a wider range of available choices. In the case of the *web frontend* service, by using automated building tools such as Maven and Gradle one can easily provide a standalone

version of the application service with an embedded web container. Nevertheless, in more complex cases, developers may be interested in limiting their choice to only those cloud services that support very well their application services' requirements. When looking for cloud services to deploy their application services, developers are required to deal with these subtleties in service offerings in order to evaluate how they match to their application service requirements and development methodology.

Variability in cloud environments

Besides the heterogeneity across providers, developers also have to handle the variability in the configuration of cloud providers. For instance, let us suppose that we want to deploy both *web frontend* and *mobile API gateway* services to Heroku platform provider. Both services require external access through HTTP requests and should be configured as *web* process types in Heroku (see Chapter 5). However, Heroku allows for only one *web* process type per application project. Hence, two application projects must be created in order to deploy these two application services to Heroku. On the other hand if the *product catalog* and *web frontend* are both Java applications deployed in the same geographical region, only one application project needs to be created because the *product catalog* is an internal service and can be setup as a *worker* process type.

Another example arises in the configuration of databases. In Heroku, an application project can include only one MySQL database. Thus, if more than one independent MySQL database is needed, multiple application projects must be created. However, Heroku supports multiple PostgreSQL databases to be provisioned into a single application project. Similar constraints abound in cloud providers and each provider have their own arbitrary set of rules that govern their configuration. Failing to follow these constraints leads to invalid configurations that will cause provisioning or deployment errors in cloud providers.

Adaptation in multi-cloud environments

Application needs evolve over time as a result of changes in its context or requirements. Here we give an example of possible changes that may arise in our example application:

- as the popularity of the application increases *user database* may require more storage space;

- as the system evolves and becomes more complex the *product catalog* may need to be refactored into separate functions of *product catalog* and *product inventory*;
- a new constraint requires that data for US users be stored within the country;
- a new version of *recommendation* service replaces the embedded database by a database service;

All these changes will lead to new requirements for the existing multi-cloud environment. Whenever requirements for an existing multi-cloud environment change we need not only to find a supporting configuration for a multi-cloud environment but also how to adapt the current configuration to a configuration that supports the new requirements. The reconfiguration may require executing a set of reconfiguration operations in the selected cloud providers (see Chapter 5) or even migrating an application service across providers if the current selection of cloud providers does not support the new requirements.

6.1.2 Challenges

Even from this simple example we can see that taking all relevant factors into account for setting up and adapting a multi-cloud environment can be a very complex task, which calls for automation and supporting tools. The goal of this work is to automate the process of finding of a proper configuration or reconfiguration plan for a multi-cloud environment, thereby freeing cloud customers from the complexities related to the management of the heterogeneity and variability across cloud providers' offerings. To achieve this goal, the following challenges must be tackled:

- ***Model multi-cloud environment requirements***. To automate the configuration of a multi-cloud environment, we need means for expressing its requirements in a formal way so that it is amenable for computer processing. This requires a provider-agnostic modeling language that enables the description of requirements related to the motivation for using multiple clouds (Section 6.3).
- ***Map multi-cloud environment requirements to cloud provider offerings***. Cloud providers' offerings are heterogeneous among themselves and based on provider-specific concepts. Meanwhile, multi-cloud environment requirements are defined based on concepts that are not directly related to any specific provider. Hence, there is a mismatch between application needs (problem space) and cloud providers' offerings (solution space). The challenge is to handle the heterogene-

ity across cloud providers and identify which cloud services support multi-cloud environment requirements (Section 6.4).

- ***Find a proper multi-cloud environment configuration***. Cloud providers are subject to complex rules governing how their cloud services can be combined and configured into a consistent configuration. Failure to comply with these rules may lead to provisioning and deployment errors. Hence, it is mandatory to ensure that generated configurations agree with the configuration variability of cloud environments (Section 6.5).
- ***Find a proper reconfiguration plan for a multi-cloud environment***. When reconfiguring a multi-cloud environment, in addition to configuration rules, it is necessary to take into account the mechanisms that are made available by cloud providers for adapting a running cloud environment, in order to generate a valid reconfiguration plan (Section 6.6).

6.2 Overview

In this section we present an overview of the proposed approach to tackle the identified challenges. This approach is organized around three main ideas:

1. The use of feature models to manage cloud environments' variability.

   Cardinality-based feature model extended with relative cardinalities (see Chapter 4) and temporal constraints (see Chapter 5) are used to model the variability in the configuration of cloud providers. Automated analysis of feature models is then used to generate cloud environment configurations that comply with cloud providers configuration rules.

2. The use of ontology reasoning for handling the heterogeneity across cloud providers.

   A cloud ontology is used as a common vocabulary across multiple cloud providers. Ontology reasoning is used to identify equivalence relationships between apparently unrelated concepts.

3. The use of a DSML for modeling multi-cloud requirements.

   The language supports definition of application services' and multi-cloud requirements. Its abstract syntax is formally defined as a metamodel based on concepts that are familiar to cloud application developers or operators.

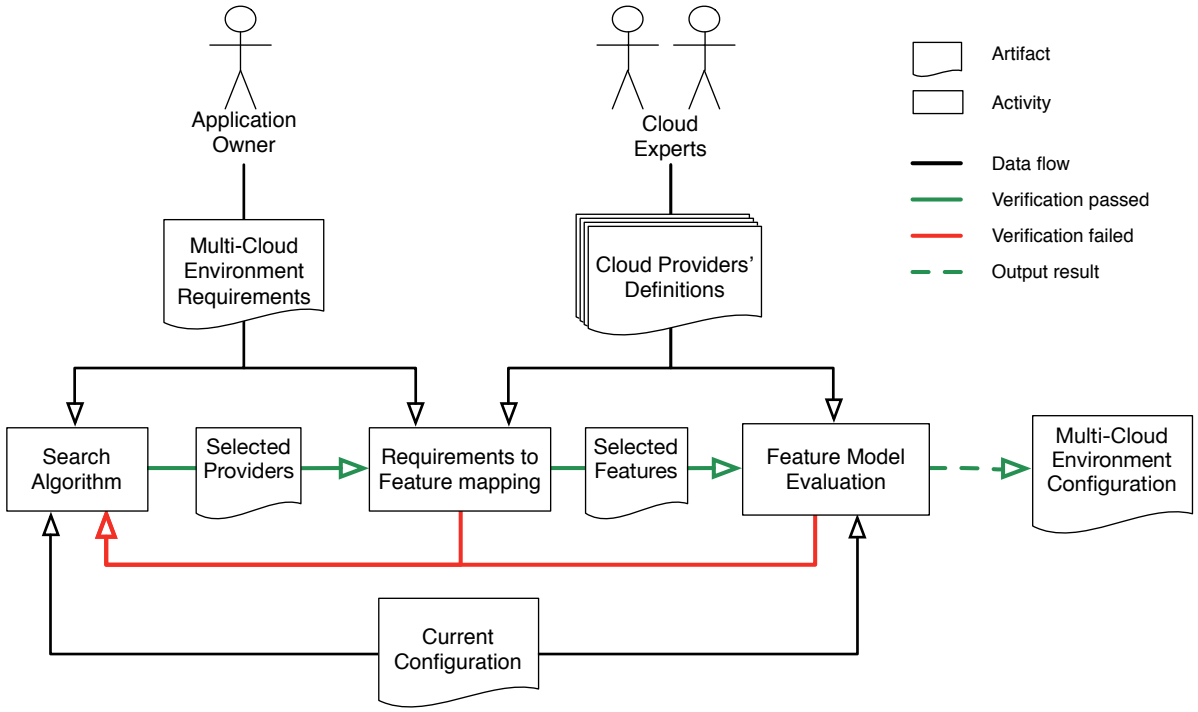


Figure 6.2 Overview of the process for generating a multi-cloud environment (re)configuration from requirements.

As discussed earlier in the introduction of this chapter, building an adaptive multi-cloud environment requires to deal with a wide spectrum of concerns, ranging from low-level issues such as the heterogeneity across management interfaces, to higher-level aspects related to service-level agreements, quality of service and pricing. Nevertheless, this work is focused on handling the heterogeneity and variability across cloud provider offerings and thereby provide means to automatically generate a multi-cloud environment configuration that supports application needs. Or to generate a reconfiguration plan, for an already setup environment, that updates it to changing application needs.

Figure 6.2 illustrates the process employed by our approach to generate a multi-cloud environment (re)configuration.

The input of this process is a description of *Multi-Cloud Environment Requirements*, which is defined by *application owners* such as cloud application developers or IT staff; or by an external system component that analyzes the context and defines new requirements for the multi-cloud environment. These requirements are described using the proposed multi-cloud environment requirements DSML (Section 6.3).

On the other hand, multiple Cloud Providers' Definitions, one for each provider, are defined by *cloud experts*. A *Cloud Provider Definition* specifies the cloud services offered by a cloud provider as well as its configuration variability. It is defined by means of a feature model that models the variability and mappings from features to ontology concepts that describe cloud services' semantics (Section 6.4).

Given these inputs, the goal is to find a multi-cloud environment configuration that meets the specified requirements. This process is guided by a search algorithm that looks for a set of candidate providers that, together, support application needs. For a given set of selected providers, it tries to map application needs to features defined in the selected providers' feature models. These selected features are then evaluated using automated analysis of feature models to generate valid configurations for each selected provider. In case of failure in any of these steps, the search algorithm will learn what caused the failure and look for another selection of candidate cloud providers (Sections 6.5 and 6.6).

The output of this process is a multi-cloud environment, which is defined by a selection of cloud providers and a corresponding feature model configuration for each of the selected providers. When adapting an already setup environment, the current environment is also taken into account and the output may also include a set of reconfiguration operations and migrations that are required to update the current system to the new configuration.

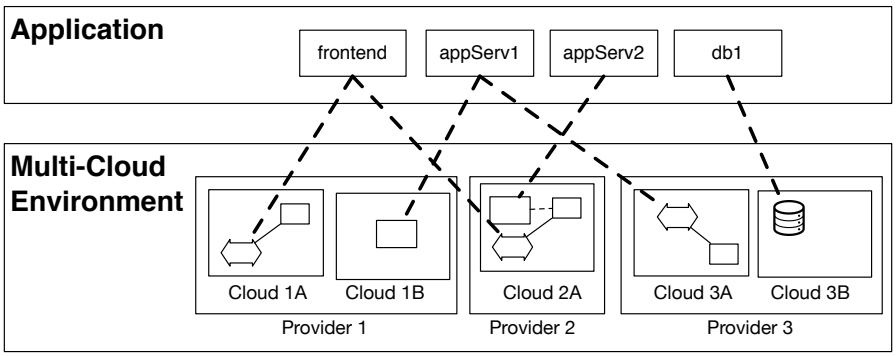


Figure 6.3 Visual representation of an application running on a multi-cloud environment

A multi-cloud environment can be defined by a selection of cloud services from multiple cloud providers that were setup to deploy and run an application. This vision is illustrated in Figure 6.3, which depicts an example application composed of multiple application services that consume cloud services from multiple providers. In the given example, the *frontend* application service is deployed to clouds $1A$ and $2A$ that are maintained by different cloud providers. A multi-cloud environment can also be regarded as a set of coordinated cloud environments, where each environment is defined by the selection

of cloud services from one specific provider. In the given example, we have a multi-cloud environment that is composed by 3 cloud environments, where each environment corresponds to one cloud provider.

An adaptive multi-cloud environment should be capable of reconfiguring itself to cope with context changes. From the viewpoint of an adaptive multi-cloud environment, the application is part of the context to which it must adapt. Hence, changes in the application execution or requirements may lead to adaptations in the multi-cloud environment. As mentioned in the previous section, examples of adaptations include the provisioning of extra resources such as virtual machines or application containers when more requests arrive, upgrading to a larger database plan as the system starts having more users, migrating a service to another provider to cope with evolution in application requirements, etc.

As already pointed out, the aim of this work is provide an approach for automatically generating a configuration for a multi-cloud environment from a description of its requirements. In an adaptive system, this contribution is expected to be integrated with other components capable of monitoring and analyzing context changes to identify if the environment needs to be changed and its new requirements for it. A description of multi-cloud environment requirements is then used as an input to the planning process from which we obtain a new multi-cloud environment configuration.

In the following sections, we describe in detail how this approach tackles the challenges listed in Section 6.1.2

6.3 A Domain-Specific Modeling Language for Multi-Cloud Environment Requirements

As discussed in Chapter 2, multiple clouds are used for different means and to achieve different goals. Thus, a language for defining multi-cloud environment requirements should be capable of representing requirements that capture the different intended usages of multiple clouds. Besides this, multi-cloud environments are tailored to support the needs of a particular application. Hence, this language should also support the definition of the specific needs of each application service that compose the application. Additionally, it should also be able to do this in a provider-independent way, based on concepts that are not specific to any particular cloud provider.

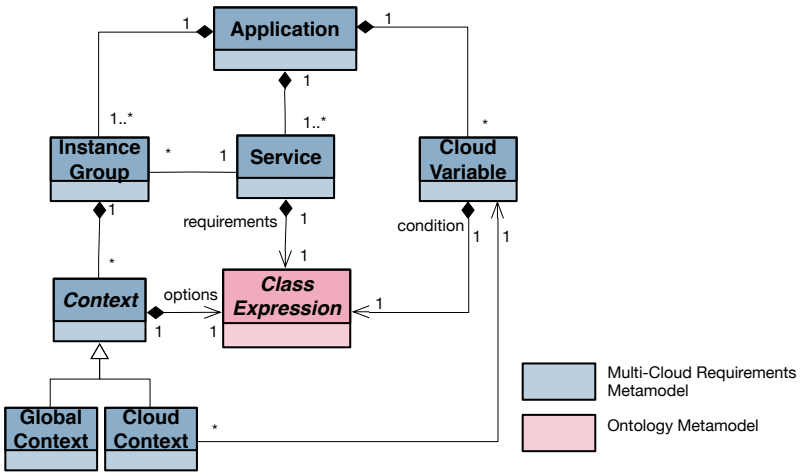


Figure 6.4 Metamodel for multi-cloud environment requirements

To support these needs we designed a DSML that enables to describe the functionality required by each application service as well as constraints on how these services can be deployed across candidate providers. Figure 6.4 shows the metamodel that describes the abstract syntax for the proposed language. The root element of the metamodel is the `Application` metaclass, which represents application needs. Application needs are defined in terms of its `Service`s, `CloudVariable`s and `InstanceGroup`s.

Application services are represented by the `Service` metaclass, while their requirements are defined as a `ClassExpression`. Application service requirements are intended to describe the functionality an application service requires from a cloud. These can be virtual machines, runtime environments, databases, software management tools, or any other cloud service. Figure 6.5 depicts an example of requirements for a multi-cloud environment inspired by the motivating example presented in Section 6.1.1. In this example, the *web frontend* service requires a WebContainer with at least 2 GB of RAM. This container should provide either the Java Development Kit version greater than 7, or a Java Web Container that implements version 2.5 of the Servlet API.

The concepts of WebContainer, memory, software, JDK, JavaWebContainer, among others used in the definition of application service requirements are part of the cloud ontology. In the cloud ontology, a WebContainer represents an executing environment that can be externally accessed using the HTTP protocol. If a required concept is not part of the cloud ontology, it is also possible to define an application specific ontology to include new concepts. For example, one could add a new ontology class called HighPerformanceWebContainer and define it as equivalent to a WebContainer that has

```
/* Service requirements */
service web_frontend {
  requires WebContainer {
    memory 2GB
    software JDK { version >= 7 } or JavaWebContainer {
      providesAPIs Servlet { version >= 2.5 }
    }
  }
}
service recommendation {
  requires VirtualMachine {
    memory 4GB
    operatingSystem Windows { version >= 7 }
  }
}
service user_db {
  requires PostgreSQL  { version [>= 8.2, <9] }
}
/* Cloud variables */
cloud A { self.provider <> B.provider }
cloud B { self.location = WesternEurope }
/* Instance groups */
instanceGroup  web_frontend {
  global { numInstances 1..20 }
  A { numInstances 2..19 options AutoScale }
  B { numInstances 1..5 options AutoScale }
}
instanceGroup recommendation {
  global { singleton }
}
instanceGroup user_db {
  B { role MasterDB }
}
```

Figure 6.5 Example of multi-cloud environment requirements

more than 4 virtual CPUs. Once a new concept is added it can be used in the description of requirements.

Application service requirements are represented as an instance of `ClassExpression` which is part of the Ontology Metamodel (see Figure 6.6) and corresponds to an OWL 2 *class expression*. According to the OWL 2 specification, a class expression represents "a set of individuals by formally specifying the conditions on the individuals' properties; individuals satisfying these conditions are said to be instances of the respective class expressions" [194]. In our case, application service requirements are understood to describe the class of cloud services that support the needs of this given application service.

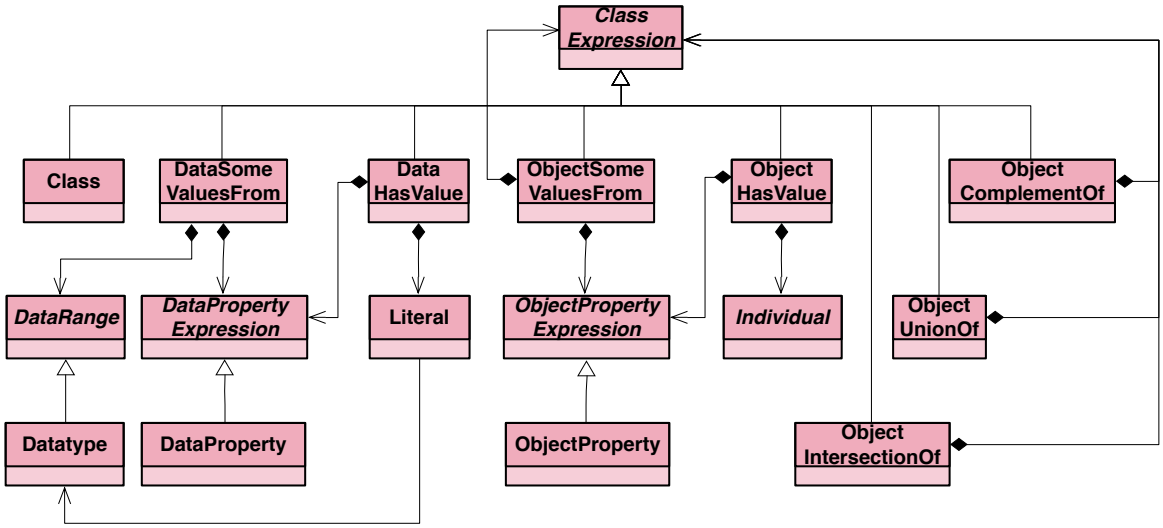


Figure 6.6 Ontology metamodel: `ClassExpression` class hierarchy

The use of ontological class expressions enables to define complex service requirements by combining ontological classes, properties and individuals with logical operators.

While service requirements define the functionality required from the cloud, `CloudVariable`s are used to define conditions on the clouds where the services are to be deployed. This allows to define constraints on characteristics of the cloud itself rather than its cloud services. By using cloud variables and conditions one can describe requirements, such as, that instances of an application service should be placed in clouds maintained by different providers, or, that a given application service should be deployed to a cloud located in a specific region. In the example of Figure 6.5, we have a condition that cloud B should be located in WesternEurope and that cloud A and B are not clouds from the same provider. Cloud variable conditions are also represented as a `ClassExpression`, that defines the class of clouds that match these conditions.

Finally, `InstanceGroup`s describe where to deploy application services and how they should be instantiated. An instance group defines parameters for the deployment of an application service at cloud and global `Context`. For instance, this construct enables to specify the role, number of instances, horizontal scaling of application services. In our example, the *web frontend* service is setup to run on both clouds A and B with different auto scaling parameters. Meanwhile, the *user db* database should be located in cloud B; and the recommendation service should have a singleton instance running in any cloud.

In essence, the proposed language enables developers to define for each application service, the functionality it requires from a cloud in a way that is independent of the actual cloud

services offered by cloud providers. Cloud variables and instance groups can be used to define scalability and redundancy rules across providers or regions, as well as location and colocation rules for application services. These constructs can be used to define multi-cloud environment requirements to match the different intended use of multiple clouds. Finally, the use of ontologies allows for defining complex requirements involving logical operators and composition rules.

6.4 Mapping Requirements to Cloud Services

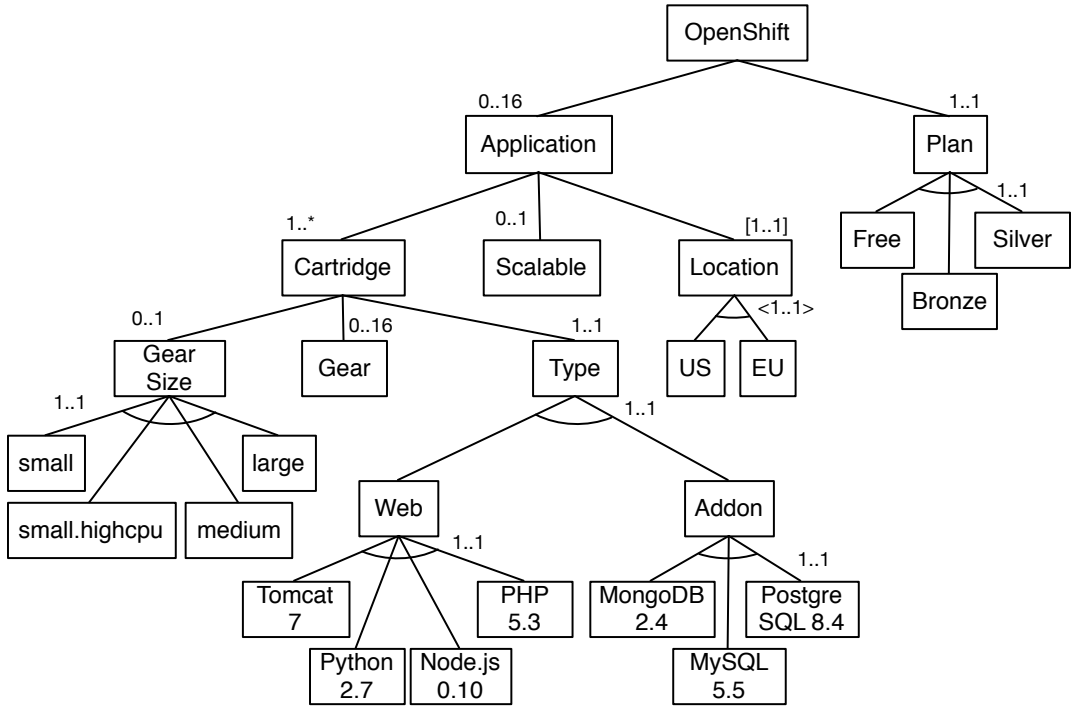
The proposed DSML enables to describe multi-cloud environment requirements by using concepts from cloud computing that are independent from the actual cloud providers' offerings. However, in order to build a multi-cloud environment, at some point we need to find a set of providers and cloud services that support these requirements.

As pointed out in Section 6.1, one of the difficulties is to overcome the heterogeneity across providers' offerings that are often based on different concepts. In this approach we propose to use ontologies to manage this heterogeneity. A cloud ontology, including concepts from the cloud computing domain, is employed as a common vocabulary that is used for the definition of both multi-cloud environment requirements and cloud providers' offerings.

An ontology is usually conceived as a formal description of concepts and how they are related among themselves. They are often used to represent knowledge about a particular domain and are also viewed as a vocabulary or taxonomy of domain concepts [195, 196]. Ontologies are largely employed in the Semantic Web [197], for which many standards and tools have been developed over the years. By using ontologies, we can leverage developments on ontology reasoning to find equivalence and specialization relationships between cloud concepts from different providers and application needs.

6.4.1 Cloud Provider Service Definition

As shown in Chapters 4 and 5, extended feature models can be used to model the variability found in the configuration of cloud providers. The hierarchical decomposition of features in the feature diagram tree, together with cardinalities and cross-tree constraints define how features can be instantiated to obtain a valid cloud environment configuration. However, feature models do not describe the semantics of their features. For instance, a



C_1: <Application> [1..1] (Free, Plan) → [1..1] (US, Location) & [1..1] (small, GearSize)

C_2: <OpenShift> [1..1] (Free, Plan) → [0..3] (Gear, OpenShift)

C_3: <OpenShift> [1..1] (Bronze, Plan) → [0..16] (Gear, OpenShift)

Figure 6.7 Excerpt from OpenShift cloud feature model

feature model can describe a constraint that choosing the `Free` plan implies using only small size `Gears` located in the `US` (see Figure 6.7). However, a feature model cannot define what is a "`small` sized `Gear` located in the `US`".

For describing semantics of features we map feature modeling elements to concepts of the cloud ontology. By combining a feature model that describes the variability in the cloud provider configuration with a mapping to cloud computing concepts we can describe the cloud services offered by a cloud provider and how they can be configured in order to build a cloud environment.

In our approach, mapping a feature model to ontology concepts is done by means of a domain-specific modeling language. Figure 6.8 depicts the metamodel for the mapping language. The `MappingModel` is the root element and is directly associated to a `FeatureModel` that describes the configuration variability of the cloud provider. The `MappingModel` can include multiple instances of `FeatureProvides` metaclass. The `FeatureProvides` element is used to establish an association between an

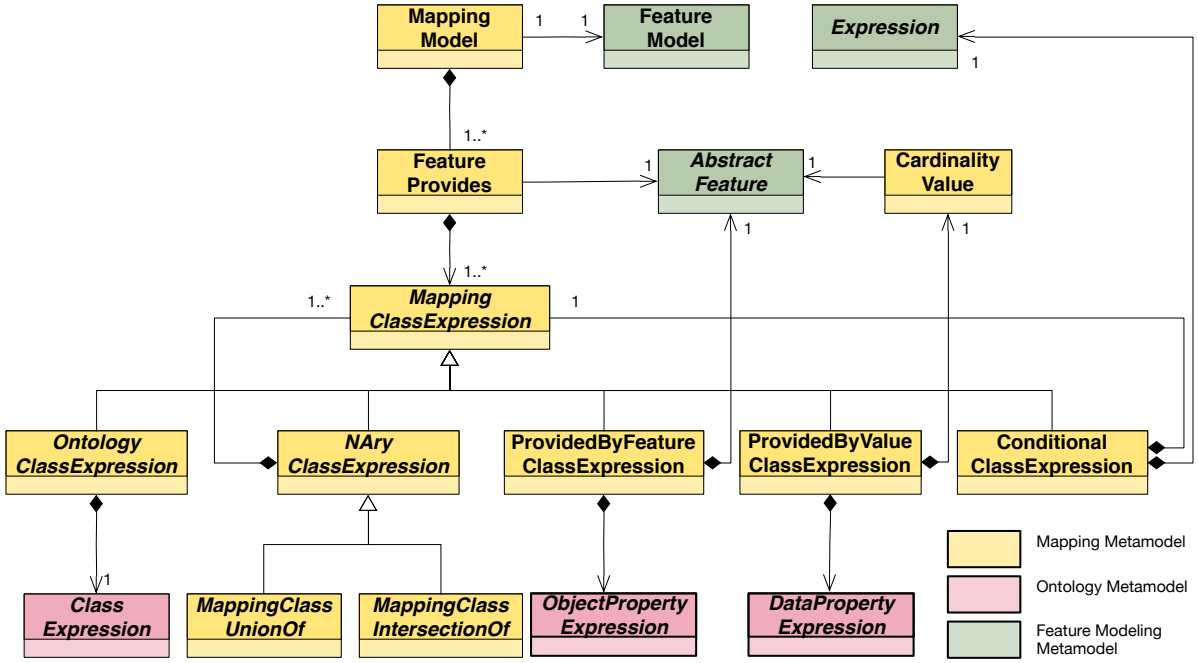


Figure 6.8 Metamodel for feature model to ontology mapping

`AbstractFeature` (i.e. `Feature` or `FeatureGroup`) and a `MappingClassExpression`. This indicates that an instance of the `AbstractFeature` in the feature model represents an ontology individual of the class defined by `MappingClassExpression`.

Figure 6.9 shows an example of mapping model for the OpenShift feature model shown in Figure 6.7. In this example, we can see a mapping from feature `Tomcat` to ontology class JBossEWS. This mapping states that an instance of feature `Tomcat` represents an individual of the ontology class JBossEWS. The concepts referenced in the mapping are also part of a cloud ontology that includes the main concepts of the cloud computing domain.

The mapping of `Tomcat` feature is defined using the `OntologyClassExpression` metaclass, which associates a feature directly to an ontology `ClassExpression`. However, the relation between features and ontology concepts is not always a direct one-to-one mapping. In many cases, related features determinate the properties of the ontology individual. For instance, still in Figure 6.9, the `Cartridge` feature in OpenShift represents an executable container that can be used to run a web application server or a database. In the cloud ontology, executable containers are represented by the class AppContainer, which has a set of properties such as CPU power, available RAM, disk size, the kind of software it can run, in which cloud it is deployed, etc. The properties of a given instance of feature `Cartridge` depend on how other related features are selected. Therefore, an instance of

```
1  feature Tomcat provides JBossEWS {
2    version value 2.0
3  }
4  feature Cartridge provides AppContainer {
5    httpExternalAccess value select Web
6
7    memory value 512MB if select small
8    vCPUs value 1 if select small
9    memory value 512MB if select small.highcpu
10   vCPUs value 2 if select small.highcpu
11   memory value 1GB if select medium
12   vCPUs value 2 if select medium
13   memory value 4GB if select large
14   vCPUs value 4 if select large
15
16   deployedInCloud value AmazonUSEast1 if select US
17   deployedInCloud value AmazonEUWest1 if select EU
18
19   option value AutoScale if select Scalable
20   option value AcrossAZs if [2..*] Gear
21   maxNumInstances value Gear
22
23   software value from Tomcat
24   software value from Python
25 }
26 /* ... */
```

Figure 6.9 Mapping from feature model for OpenShift PaaS to cloud ontology

feature `Cartridge` will have 1GB of RAM if it is of `medium` size, it will provide a JBoss Enterprise Web Server 2.0 as a servlet container if `Tomcat` feature is selected, and it will be deployed to to Amazon US-East-1 cloud if the `US` feature is selected.

To express this kind of mapping, we can use subclasses of `MappingClassExpression` that support definition of a class expression based on the selection of features. These metaclasses are organized according to the interpreter pattern, mimicking the hierarchy of `ClassExpression` metaclass in the Ontology Metamodel (see Figure 6.6). The mapping for feature `Cartridge` in Figure 6.9 illustrates how such a mapping can be defined using the concepts of the metamodel. For instance, the `ConditionalClassExpression` is used through lines 7-14 to define the processing power and available RAM for a `Cartridge` instance according to the selection of features `small`, `small.highcpu`, `medium`, `large`. The `ProvidedByValueClassExpression` metaclass is used in line 21 to map the value of the data property maxNumInstances to the number of instances of feature `Gear`. Meanwhile, `ProvidedByFeatureClassExpression` is used in lines 23-24 to map an ontology property

value to an individual provided by another feature. For example, a `Cartridge` will include JBossEWS as software when feature `Tomcat` is selected.

Like for multi-cloud environment requirements, if there are concepts that are not available in cloud ontology or that are provider specific, it is possible to define a provider-specific ontology. For example, to simplify the definition of mappings, we can define an ontology for OpenShift and include the concept of OpenShiftSmallAppContainer, which will be defined as an AppContainer with 512MB of RAM and 1 virtual CPU. Once this concept is introduced to the provider-specific ontology it an be directly used in the mapping description.

Using the designed mapping language it is possible to define how feature instances in a cardinality-based feature model can be mapped to concepts of the cloud ontology and therefore their semantics. So, given the mapping in Figure 6.9, we can infer that a `small Cartridge` located in `US` is actually an AppContainer, which is an executing environment, with 512MB of RAM deployed to the Amazon US-East-1 cloud, which is located in North Virginia, United States.

6.4.2 Mapping Service Requirements to Feature Selections

As explained in Section 6.3, application service requirements are defined as an ontological class expression that defines the class of cloud services that support it. By matching this class against mappings from feature models to ontology concepts we can obtain a feature selection that specifies the combinations of feature instances that correspond to an instance of the described class expression.

Figure 6.10b shows the class expression, using Manchester Syntax for OWL 2 [198], that captures the application service requirements for the *web frontend* service in Figure 6.10a. By reasoning over ontologies, together with cloud provider mappings we obtain feature selection that provides an instance of this required class. Figure 6.10c shows the feature selection that is obtained by mapping *web frontend* service requirements to features from OpenShift provider.

In this example, the *web frontend* service requires a WebContainer, but feature `Cartridge` in OpenShift provides an `AppContainer` (see Figure 6.9). However, by reasoning over the ontologies we can find out that WebContainer is equivalent to an AppContainer for which the data property httpExternalAccess is set to true. The mapping for feature `Cartridge` defines that this property is set to true iff the feature `Web` is also selected.

```
service web_frontend {
  requires WebContainer {
    memory 2GB
    JavaWebContainer {
      providesAPIs Servlet { version 2.5 }
    }
  }
}
cloud B { self.location = WesternEurope }
instanceGroup web_frontend {
  B { numInstances 1..5 options AutoScale }
}
```

(a)

```
WebContainer that hasMemory some (DataAmount that hasUnit value MB and
    hasValue some integer [>= 2048]) and hasSoftware some (
    JavaWebContainer that providesAPIs some (JavaServlet that hasVersion
    value 2.5)) and deployedInCloud some (Cloud that isLocatedIn some (
    partOf value WesternEurope)) and options value AutoScale and
    maxNumInstances some integer[>= 5]
```

(b)

```
Cartridge { large, Tomcat, EU, Scalable, 5 Gear }
```

(c)

Figure 6.10 Mapping from application service requirements to feature selections

Thus, an instance of feature `Cartridge` provides a WebContainer if its subfeature `Web` is selected. Likewise, we can find that JBossEWS service provided by feature `Tomcat` implements the Servlet API. When reasoning to find equivalences, both provider-specific and requirements-specific ontology are used together with the cloud ontology.

The obtained result, shown in Figure 6.10c, states that an instance of feature `Cartridge` is an individual of the class expression in Figure 6.10b if features `large`, `Tomcat`, `EU`, `Scalable` are selected and there are 5 instances of feature `Gear`. By combining multiple *feature selections* for different application services, we can build a partial configuration for a cardinality-based feature model as shown in Section 4.3.4. These can then be used to automatically generate a complete configuration that provides the functionality required by a set of application services.

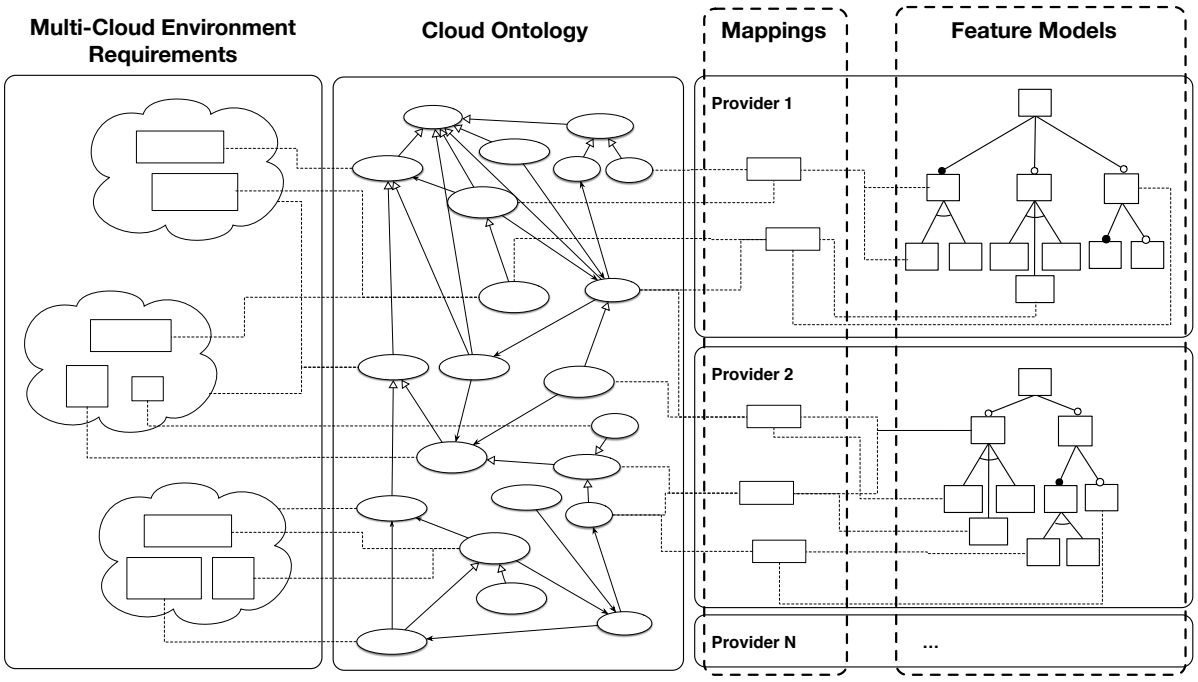


Figure 6.11 Overview of the artifacts employed in the approach

In a summary, ontologies are used to bridge the gap between service requirements and cloud providers. Figure 6.11 illustrates how the different artifacts employed in the approach are integrated by means of ontology concepts. By using ontologies we can describe requirements that are as complex as an ontology class description and reason upon them to find which features from cloud providers may provide instances of these classes. This enables to overcome the heterogeneity across providers, providing means to identify which cloud providers support application services' requirements; and which cloud services, represented by means of a feature selection, provide the required functionality.

6.5 From Requirements to Configuration of a Multi-Cloud Environment

The designed DSML allows to define multi-cloud environment requirements based on concepts from the cloud ontology. Application service requirements can then be matched against a cloud provider definition to obtain a feature selection that describes the set of features from this cloud provider that support the requirements of this given application service. However, to generate a complete configuration for a multi-cloud environment,

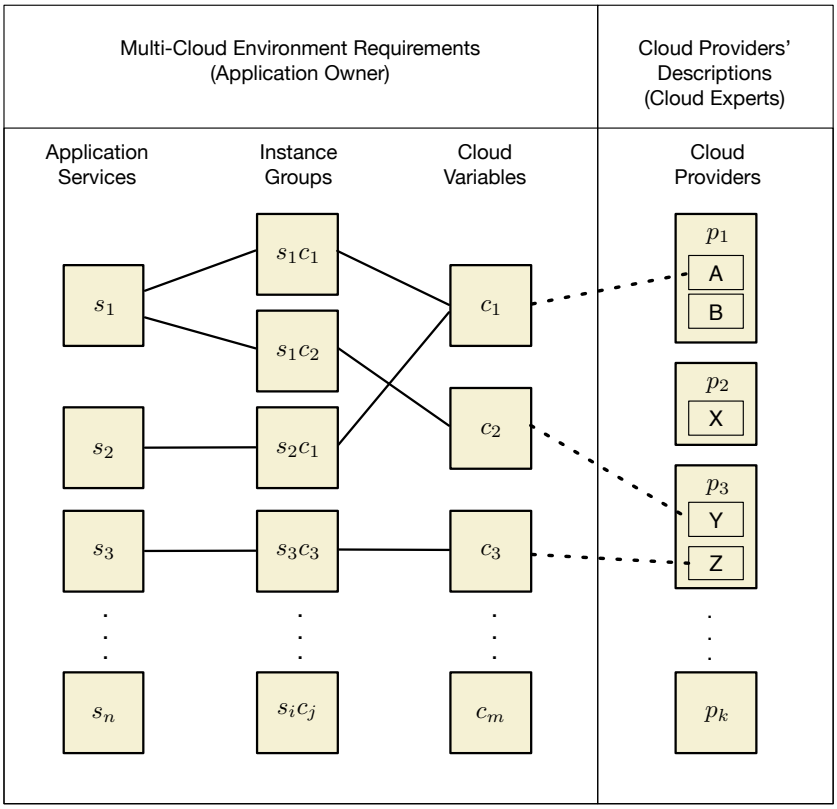


Figure 6.12 Multi-cloud configuration problem

we still have to select a set of cloud providers that support the multi-cloud environments and generate a complete configuration for them.

The problem of generating a multi-cloud configuration that suits application needs can be seen as an assignment problem subject to constraints. As illustrated in Figure 6.12, an instance of this problem is described by multi-cloud environment requirements and a set of available cloud providers. From requirements we have a set of *application services*, which are assigned to *cloud variables* through a set of *instance groups* (see Section 6.3). On the other side, we have a set of *clouds* maintained by different *cloud providers*. A solution to an instance of this problem is an assignment from *cloud variables* to *clouds*. A solution is valid, i.e. complies with the constraints, if the selected clouds, together, support all the specified requirements.

Due to the number of providers, the search space for this problem can be very large, making it difficult to find a solution in a reasonable time. Besides this, verifying the suitability of a candidate solution is a complex and time consuming task that requires reasoning on ontologies to convert application requirements to provider specific constructs

and evaluating variability in feature models. These factors, when combined, may make this problem intractable. To handle this complexity, verification of a candidate solution is conducted in a step-wise fashion, in which less expensive constraints are evaluated first to eliminate invalid solutions in a faster way and thus reduce the search space. We start with a set of constraints obtained from cloud conditions (see Section 6.3) and increase our constraint base as verification of assignments fail. Doing so, we avoid performing expensive validations beforehand and gradually reduce the search space as new constraints are learned.

Verification of a candidate solution is done in three steps: 1) validating that assigned providers comply with cloud conditions; 2) checking that application service requirements can be mapped to a set of features in the assigned provider; 3) verifying that there is a valid configuration where all services assigned to a provider can be deployed and generating a complete configuration. During the search process, these steps are performed for each candidate solution to verify if it supports application needs. These steps are described below.

Validating cloud conditions

Cloud conditions are those described in application requirements as part of the cloud variables description (see Section 6.3). In Figure 6.10a, cloud variable B has as a condition that it should be assigned to a cloud located in WesternEurope. Evaluating these conditions is relatively simple and consists in checking if each assigned cloud variable complies with its condition. However, this may still require some semantic reasoning to identify, for example, that a cloud located in France meets the criteria of being located in WesternEurope.

Mapping application service requirements to provider features

After verifying that the assigned providers comply with cloud conditions we still need to check if each selected provider supports all functionality required by the application services assigned to it. To do so, we use the method described in Section 6.4.2 to verify, for each application service, if its requirements can be mapped to a feature selection. If the mapping cannot be found, it means that the assigned provider does not support the application service being mapped. This information is added to the constraint base to further reduce the search space and a new candidate solution is selected to be evaluated.

Generating a complete configuration

After mapping service requirements to a selection of features, we know that the assigned providers support the application services assigned to them. However, as there may be limits in the resources offered by a provider, conflicting features, or other constraints in cloud offerings, we still need to check if all services assigned to a provider can be deployed together and to generate a configuration for each selected provider. This process is done by evaluating the cloud providers' feature models against the feature selections obtained in the previous step.

This is a complex and time consuming task, which involves translating the feature model together with feature selections into a constraint program, and solving it to find a valid configuration that matches the requirements. This process is explained in detail in Section 4.3.4. If a valid configuration is found for all selected cloud providers, it is a valid solution to the problem. The output of this process is an assignment from cloud variables to clouds, and a set of feature model configurations, one for each selected provider. In case of failure, one of the selected providers does not supports the deployment of all the assigned services together and new candidate solution should be tried. Information about the cloud provider for which the feature model analysis failed is added to the constraint base.

These validation steps are executed for evaluating the validity of a candidate solution, which is represented by an assignment from clouds variables to clouds. The query for a valid solution is encoded as a constraint program. In this encoding, each cloud variable defined in the requirements is represented as an enumerated variable, for which the domain is defined by the set of available cloud providers. Besides this, custom constraint propagators are used to implement the verification steps described above. A solution to this constraint problem corresponds to a valid selection of cloud providers and the generated configurations can be obtained from the results of the last verification step.

By applying this process to multi-cloud environment requirements we can obtain a selection of cloud providers and a set of configurations for their respective feature models. Decomposition of the verification of candidate solutions is employed to avoid executing more expensive calculations when not needed and enable reuse intermediate results. Once a configuration is obtained it can be used to guide the generation of configuration artifacts such as configuration scripts and files, API calls, etc to set up a multi-cloud environment.

6.6 Adaptation of Multi-Cloud Environments

In the preceding section, we described how to obtain a configuration for a multi-cloud environment from its requirements specification. However, as application context and requirements changes over the time the multi-cloud environment may need to adapt accordingly, in order to support new application needs. As explained in the introduction of this chapter and in Section 6.2, these changes will lead to new multi-cloud environment requirements. From these new multi-cloud environment requirements, we are interested in finding a new configuration that supports them; and the reconfiguration mechanisms available to transition the running environment from the current configuration to this new configuration.

Changes in multi-cloud environment requirements can lead to three different adaptation scenarios.

- No reconfiguration is needed when the new multi-cloud environments are already met by the current configuration. For instance, the initial requirements for *web frontend* service (see Figure 6.5) specify that it requires an executing container with at least 2GB of RAM. If this service was initially deployed to the OpenShift provider, it would be using a `large` container that provides 4GB of RAM (see Figures 6.7 and 6.9). Hence, if this application service requirements were later updated to need up to 4GB of RAM, no reconfiguration would be needed.
- When the new requirements are supported by the current selection of cloud providers but using different cloud services, a provider reconfiguration is required. For example, if the *web frontend* service were initially deployed to OpenShift provider, it would be using the `Tomcat` feature (see Figures 6.5 and 6.9). If the new service requirements request support for a different application server, such as NodeJS, then the configuration would need to be updated to select `NodeJS` as `Cartridge` type (see Figures 6.7 and 6.9). This scenario occurs, when new requirements for an application service are mapped to a feature selection that is not supported by the current configuration. In this case, cloud provider reconfiguration constraints and operations must be evaluated in order to generate a new configuration.
- Finally, there are cases in which new requirements for one or more application services are not supported by the currently selected cloud providers. In this case, we need to find a new selection of cloud providers that support the new requirements,

```
MigrationService that supportsInput some (ResourceType that isExportedBy
    value user_db) and supportsOutput some (ResourceType that isImportedBy
    value X_user_db)
```

Figure 6.13 Migration operation query

and verify if it is possible to migrate these application services across different providers.

To obtain a reconfiguration plan from new multi-cloud environment requirements we use the same approach for the initial setup of the multi-cloud environment described in Section 6.5 with two minor changes. The first is the inclusion of an extra verification step to identify the migration operations required to move application services between providers used in the source and target configuration. The second modification is on the step responsible for the verification of variability and generation of a complete configuration, which is updated to use the approach described in Chapter 5.

Verification of migration operations

The goal of this step is to verify the availability of migration services capable of migrating the application services that need to be migrated between the providers. As other cloud services, migration services are defined as part of the cloud offer in the cloud or provider ontologies. For verifying the migration operations, we also rely on ontology reasoning to find a migration service that supports an input in the format of the currently used provider and output in the format of the newly selected provider. For instance, if we consider the case of migrating a `PostgreSQL` database service in Heroku to `MySQL` database located in a cloud maintained by Amazon we could use the *AWS Database Migration Service* to perform the migration in an automated way. This service supports migrating data stored in the most popular relational database management systems to a database in Amazon servers.

Figure 6.13 illustrates how such an example is encoded as an ontology class expression. By querying for individuals of this described class expression we obtain the available migration services that support migrating the *user database*.

In this example, user_db and X_user_db represent respectively the database service currently used and the database service to which the data must be migrated. These

correspond to the ontology concepts created when mapping application from service requirements to feature selections (see Section 6.4.2) and their classes match those classes obtained from service requirements. Hence, if the *user database* service is currently deployed to OpenShift using the cloud service provided by feature `PostgreSQL`, then user_db is an individual of the class defined in the mapping model for OpenShift cloud provider.

As discussed in Section 6.4, a cloud service is defined as an ontology class. Each cloud service is expect to include as part of its definition how application data can be imported or exported from it. For instance, in database services it is a common practice to always have at least one way for exporting and importing a backup or dump of the database, but additionally, cloud providers may offer means for migrating data using a direct connection or using multi-master or master-slave synchronization schemas. Likewise, for application services we to find different ways of packaging software according to the programming language or platform ecosystem. Cloud providers will often support at least one way of deploying an application service from a deployable artifact. In the case of specific needs of an application or if the application owner is ready to provide additional means for migrating their application services across different providers it can do this by adding new MigrationServices to the application-specific ontology.

By adding this verification step to the constraint problem, we filter out candidate solutions for which there are no known migration services for migrating the services that need to be migrated to this new configuration. A valid solution will be defined by a new assignment from cloud variables to clouds and a set of migration operations that should be applied to migrate services that are assigned to different providers in the new configuration. The process of generating a reconfiguration plan is an extension of the process for generating an initial configuration, described in Section 6.5.

6.7 Discussion

The approach proposed in this chapter enables obtaining a configuration or a reconfiguration plan to setup or adapt a multi-cloud environment from a description of its requirements. In particular, it is focused on handling the variability in the configuration of cloud providers and the heterogeneity across their service offerings.

When building an adaptive multi-cloud system, our approach is expected to be integrated with other components capable of monitoring and analyzing context changes to identify

new requirements, which will then be used to generate a new reconfiguration plan. Likewise, it does not handle the enacting of the generated reconfiguration plans into the running environment, which includes the generation of configuration files, scripts, API calls, etc. If we compare it to the autonomic computing model [69], the approach presented in this chapter would fit into the *plan function* of the MAPE-K control loop.

Another point that is important to highlight is that the aim of this approach is to support the construction of a multi-cloud environment based on the service offerings from multiple providers. Hence, it does not provide new services or functionality on the top of what is offered by different providers, but it rather aggregates available cloud services from different providers on a per-case basis, according to the needs of the application at hand. For instance, this approach does not directly provide any support for elasticity or autoscaling of application services within or across cloud providers, but rather maps these requirements to cloud services offering these capabilities.

As we mentioned earlier, one of the motivations for consuming resources from multiple clouds is to optimize the performance, availability or costs of a system by selecting services that are best suited for the application. However, the proposed approach does not explicitly provide means for generating optimized configurations. As described in Sections 6.5 and 6.6, during the generation of a configuration or reconfiguration plan only the compliance to requirements and cloud provider offerings is verified.

6.8 Summary

In this chapter, we presented an approach to automate the setup and adaptation of multi-cloud environments. This approach generates a configuration or reconfiguration plan for a multi-cloud environment from a provider-agnostic description of its requirements. Multi-cloud environment requirements are described in terms of concepts from the cloud ontology using a domain-specific modeling language. Meanwhile, cloud providers' service offers are described by means of feature models, which capture their configuration variability, and mappings from feature modeling elements to cloud computing concepts. Semantic reasoning is employed to map the provider-agnostic requirements to provider-specific features across different cloud providers. The use of the previously developed methods for managing variability in cloud environments (see Chapters 4 and 5) ensure the generation of correct configurations, which comply with providers' configuration rules.

During the initial setup of a multi-cloud environment the approach finds a selection of cloud providers that support the described requirements and generates configurations by employing automated analysis of feature models. When requirements change, the approach verifies if some services need to be migrated across different providers and verifies which reconfiguration and migration operations need to be executed.

The presented approach generates a configuration or reconfiguration plan from a description of application needs, thus providing an answer to RQ_3 outlined in Section 1.2.

This chapter concludes the contributions of this thesis. In Chapters 4 and 5 we presented contributions to improve variability management of cloud environments by means of relative cardinalities and temporal constraints. In Chapter 6, we demonstrated how these contributions were integrated into an approach for automating the generation of multi-cloud environment configurations from a description of requirements. In the following chapter, we present an evaluation of these contributions.

Part III

Evaluation

Chapter 7

Evaluation

This chapter presents more details of the tool support developed for the approach described in Chapter 6. This includes information on the artifacts and activities involved in the generation of multi-cloud environment configurations, as well as design choices, algorithms and tools employed in its implementation. In addition, we report on the results of an experimental evaluation conducted to assess the feasibility and performance of the approach. These experiments aim to identify the impact of the variability modeling extensions introduced in Chapters 4 and 5 on the automated analysis of feature models, and how the approach presented in Chapter 6 performs on a realistic scenario based on a case study application. Further information about the developed tools and data from the experimental evaluation is available in the project web page[1].

This chapter is organized as follows: Section 7.1 describes implementation details of the tool support developed for the proposed approach. Section 7.2 reports on the protocol and results of experiments conducted to evaluate the approach. In Section 7.3, we discuss the advantages and limitations of the approach. Sections 7.4 provides a summary of this chapter.

7.1 Implementation details

In order to evaluate the proposed approach, we developed tool support for modeling requirements and feature models, as well as for automatically generating configurations. Throughout this section, we present an overview of the approach implementation.

[1] http://researchers.lille.inria.fr/~sousa/phd/mmcloud/

7.1.1 Modeling

As discussed in Chapter 6, multi-cloud environment requirements and cloud providers offerings are defined using domain-specific modeling languages. The abstract syntax of these modeling languages is defined by means of metamodels, which were presented in Chapters 4, 5 and 6. The language processing implementation is based on the xText [165] framework together with the Eclipse Modeling Framework [166]. Together, these frameworks enable to automatically generate a platform for editing and parsing the artifacts employed in the approach.

Ontologies, which are also employed in the approach, can be described using the standard OWL 2 Web Ontology Language [199], for which many modeling and reasoning tools are available. The ontology metamodel provided by the TwoUse project [200, 201] is used to integrate it into the Eclipse Modeling Framework, enabling ontology concepts to be directly referenced from other artifacts.

7.1.2 Reasoning

As explained in Chapter 6, the goal of the approach proposed in this thesis is to generate a configuration or reconfiguration plan for a multi-cloud environment, from a description of its requirements. This process requires two main inputs: (i) a set of cloud providers' definitions, which describe the cloud services offered by the cloud provider and how they can be configured; and (ii) a description of multi-cloud environment requirements.

Figure 7.1, depicts some of the implementation classes of the reasoning mechanism. The classes in the right box are related to the definition of cloud providers, while the top left box contains the classes that represent multi-cloud environment requirements. The classes in the bottom left box represent a multi-cloud environment configuration, obtained as a result of the reasoning process.

As shown in Section 6.4, a cloud provider is defined by a feature model and mappings from feature model elements to cloud concepts defined in the cloud ontology. Optionally it is also possible to define a provider-specific ontology that establishes the relationship between provider-specific and general cloud computing concepts.

When a cloud provider is initially loaded, feature model cardinalities are evaluated to verify their consistency and infer additional cardinalities that will be required later for generating a complete configuration (see Section 4.2.2). Likewise, temporal constraints

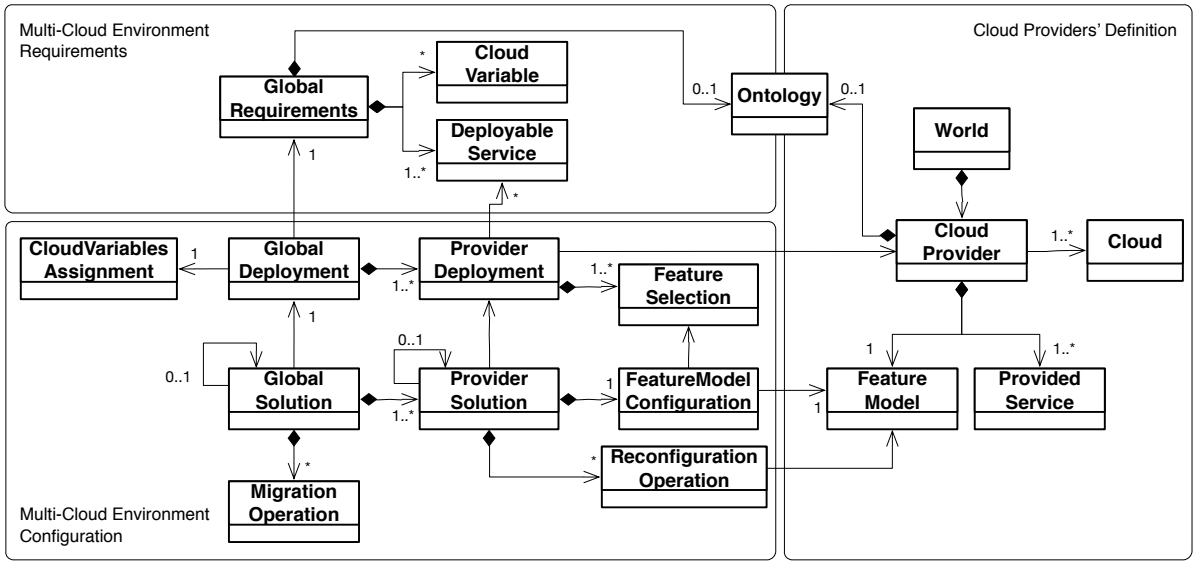


Figure 7.1 Main classes in the reasoner implementation

included in the feature model are translated into a symbolic representation that will be employed during the reconfiguration of a cloud environment (see Section 5.3.2). From this process we obtain a `World` instance, which is loaded with a set of cloud providers.

On the other hand, requirements for a multi-cloud environment (represented by the `GlobalRequirements` class), define a set of *cloud variables* and *deployable services*. The `CloudVariable` class maps directly to the cloud variables defined in multi-cloud requirements, while a *deployable service* represents an application service at an specific context (see Section 6.1.2). For instance, if an application service was assigned to two different cloud variables in the requirements description, two different deployable services will be created for this single application service.

Once *multi-cloud environment requirements* are input into the reasoning mechanism, *cloud variables* defined as part of the requirements, together with their conditions are translated into a constraint programming problem. In this problem, each *cloud variable* is encoded as an enumerated variable whose domain is the set of available *clouds* offered by the *cloud providers* loaded into the `World` instance. Cloud conditions are encoded as custom constraint propagators.

By solving satisfaction over this problem, we can iterate over a set of assignments from cloud variables to clouds that conform to the cloud conditions defined in the requirements. Each *cloud variables assignment* corresponds to a candidate selection of *cloud providers*. Given this assignment, the reasoner will group *deployable services* by the assigned *cloud*

```
1  World world = World.createWorldFromDir(new File("/fr/inria/spirals/world/"));
2  CloudProvider providerX = XtextLoader.INSTANCE.loadCloudProvider(new File("/fr/inria/
       spirals/ProviderX.setup"))
3  world.addCloudProvider(providerX);
4
5  GlobalRequirements requirements = XtextLoader.INSTANCE.loadRequirements(world, new File
       ("/fr/inria/spirals/requirements/MultiCloudExample.config"));
6  GlobalSolution initalConfig = world.validSolutions(requirements).next();
7  for (ProviderSolution providerSolution : intialConfig.getProvidersSolutions()) {
8      System.out.println(providerSolution.getFeatureModelConfiguration().printTree());
9      for (ServiceSolution serviceSolution : providerSolution.getServicesSolutions()) {
10         System.out.println(serviceSolution.getFeatureInstancePath());
11         System.out.println(serviceSolution.getProvidedOWLClassExpression());
12     }
13 }
14
15 GlobalRequirements newRequirements = XtextLoader.INSTANCE.loadRequirements(world,
       getClass().getResource("/fr/inria/spirals/requirements/MultiCloudExample.v2.config"
       ));
16 GlobalEvolution reconfig = world.validSolutions(newRequirements, initialConfig).next();
17 System.out.println(reconfig.getMigrations());
18 for (ProviderSolution providerSolution : reconfig.getProvidersSolutions()) {
19     System.out.println(providerSolution.getReconfigurationOperations());
20 }
```

Figure 7.2 Example of utilization of the reasoner API.

provider and generate a *provider deployment* for each selected *cloud provider*. For each *provider deployment*, the reasoner maps application services' requirements to *feature selections* and analyze their corresponding *feature models* to obtain a *provider solution*. This process is illustrated in Figure 6.2, in Section 6.2.

A *provider solution* includes a *feature model configuration* and may include a set of *reconfiguration operations* in the case of an adaptation. Also, in the case of an adaptation, if the new cloud variables assignment is different from the previous one, some application services may need to be migrated across cloud providers. The *migration operations* are represented as part of the *global solution*, which gathers together a *provider solution* for each selected *cloud provider*.

Figure 7.2, illustrates the use of the API provided for using the approach reasoning mechanism. In the first line, we create a new instance of the reasoning mechanism by automatically loading a set of providers from a directory. Lines 2-3, illustrate how a new cloud provider definition can be loaded from a file. The `XTextLoader` class provides means for loading cloud providers' definitions and multi-cloud environment requirements defined using their respective DSMLs. A cloud provider is defined by two artifacts: a feature model and a set of mappings. In this example, the mappings are defined in the file `ProviderX.setup`, this file itself contains a reference to the feature model, which is defined in a file named `ProviderX.fm`. Lines 5-13, demonstrate how to generate a configuration from requirements. In addition, it also prints out the obtained feature

```
1  public class World {
2    public static World createEmptyWorld();
3    public static World createWorldFromDir(File file);
4
5    public boolean addCloudProvider(CloudProvider cloudProvider);
6
7    public Iterator<GlobalSolution> validSolutions(GlobalRequirements requirements);
8    public Iterator<GlobalSolution> validSolutions(GlobalRequirements requirements,
       GlobalSolution previousSolution);
9
10   public Iterator<GlobalDeployment> validDeployments(GlobalRequirements requirements);
11   public Iterator<GlobalDeployment> validDeployments(GlobalRequirements requirements,
       GlobalSolution previousSolution);
12
13   public Iterator<CloudVariablesAssignment> validAssignments(GlobalRequirements
       requirements);
14   public Iterator<CloudVariablesAssignment> validAssignments(GlobalRequirements
       requirements, GlobalSolution previousSolution);
15
16   public GlobalSolution solve(GlobalDeployment deployment, ValidatingContext context);
17   public GlobalSolution solve(GlobalDeployment deployment, ValidatingContext context,
       GlobalSolution previousSolution);
18
19   public GlobalSolution process(GlobalRequirements requirements,
       CloudVariablesAssignment assignment, ValidatingContext context);
20   public GlobalSolution process(GlobalRequirements requirements,
       CloudVariablesAssignment assignment, ValidatingContext context, GlobalSolution
       previousSolution);
21 }
```

Figure 7.3 Interface of the `World` class.

model configuration, and for each application service the feature instance providing the functionality required by it and the ontology concept that is provided by this given feature instance. Finally, lines 15-20, show how to obtain a reconfiguration from new requirements and a previous configuration.

Figure 7.3, illustrates the main methods of the `World` class. In addition to the method `validSolutions(GlobalRequirements)`, used for obtaining a global solution, it also provides methods for obtaining partial results and processing individual candidate solutions.

Other information

The reasoning mechanism was developed mostly in Groovy programming language (21,028 LOC), with some parts in Java (7,266 LOC). All processing that requires ontology reasoning employs the Hermit reasoner [202]. Likewise, for solving constraint problems it employs the Choco solver [168].

7.2 Experiments and Evaluation

This section reports on a set of experiments conducted to evaluate the contributions of this thesis. These experiments were conducted using the tools presented in the preceding section. The goal of these experiments is to evaluate the feasibility of the approach proposed in Chapter 6 for generating multi-cloud environment configurations and their underlying contributions on variability modeling with feature models extended with relative cardinalities (see Chapter 4) and temporal constraints (see Chapter 5).

The experiments are organized into two categories. The experiments in the first category were conducted with the goal of assessing the impact of relative cardinalities and temporal constraints on the automated analysis of feature models. As these modeling constructs improve the expressive power of feature models, they require additional processing, which may incur in performance penalties. The goal of this evaluation is to understand the impact on the automated analysis of feature models.

The second category of experiments is aimed at evaluating the usefulness of the approach for automating the generation of configuration and reconfiguration plans in multi-cloud environments. Their goal is to assess how the approach performs in a realistic scenario based on a case study application.

In the remainder of this section we describe in detail the protocol and the results obtained for each of the conducted experiments. Sections 7.2.1 and 7.2.2 report on the experimentation of extended feature modeling constructs. In Section 7.2.3, we report on the experimentation of the proposed approach for automating the setup and adaptation of multi-cloud environments. All experiments were run on a late 2013 MacBook Pro Laptop with a 2 GHz Intel Core i7-4750HQ processor and 8GB of DDR3 RAM.

7.2.1 Feature Models with Relative Cardinalities

The experiments described in this section were conducted to assess the impact of the introduction of relative cardinalities to the automated analysis of feature models. We evaluate this impact in the execution of two analysis operations: (i) generation of a complete configuration from a partial one, also known as auto-completion, and (ii) the verification of compliance of a configuration to a feature model. Besides this, we also evaluate the performance of verifying the consistency between multiple relative

cardinalities. These operations and their implementation are described in detail in Section 4.3.

To execute the experiments, we randomly generated feature models of different sizes (50, 100, 250, 500, 1000, 2500 features). Feature models were generated to be similar to those found in cloud environments and are based on our experience on modeling cloud environments as feature models. In feature models for cloud environments, feature cardinalities are concentrated in the higher levels of the feature diagram hierarchy, while features further down the hierarchy often represent optional or alternative feature groups. Therefore, we defined that features in the first three levels of the hierarchy had a 50% probability of having a local cardinality, while in lower levels this chance would be of 1%. For the features chosen to have cardinalities, a random cardinality range was generated with the maximum upper bound of 10 instances.

Later, for each generated feature model, we randomly selected half of the features for which local cardinalities were defined and added extra relative cardinalities to them. For each feature model size, 50 different models were generated and had relative cardinalities added. From this process, we obtained two sets of feature models: one set of random feature models with *local cardinalities*, and another set of the same models augmented with *relative cardinalities*.

As described in Section 4.3.4, in order to verify a partial configuration and generate a complete configuration for a feature model, the first step is to translate the feature model into a constraint satisfaction problem. As relative cardinalities add extra variables and constraints, they have an impact on the time required for this translation. For the first experiment, we measured, for both sets of feature models, the time to fully translate a model to a constraint satisfaction problem and solve it. Figures 7.4 and 7.5 show, in a logarithmic scale, the distribution of execution times by the number of features for both sets of feature models.

As we can see from the chart, translating a feature model with relative cardinalities introduces very little overhead into processing, which in the conducted experiments was on average of 2.3%. In the longest running case found in our experiments, it took around 171s to translate a feature model with 2500 features that can generate configurations with up to 91,845 feature instances with relative cardinalities, compared to 166s to translate it without relative cardinalities.

This result is expected as the number of constraints and variables introduced into the underlying constraint satisfaction problem due to relative cardinalities is very small when

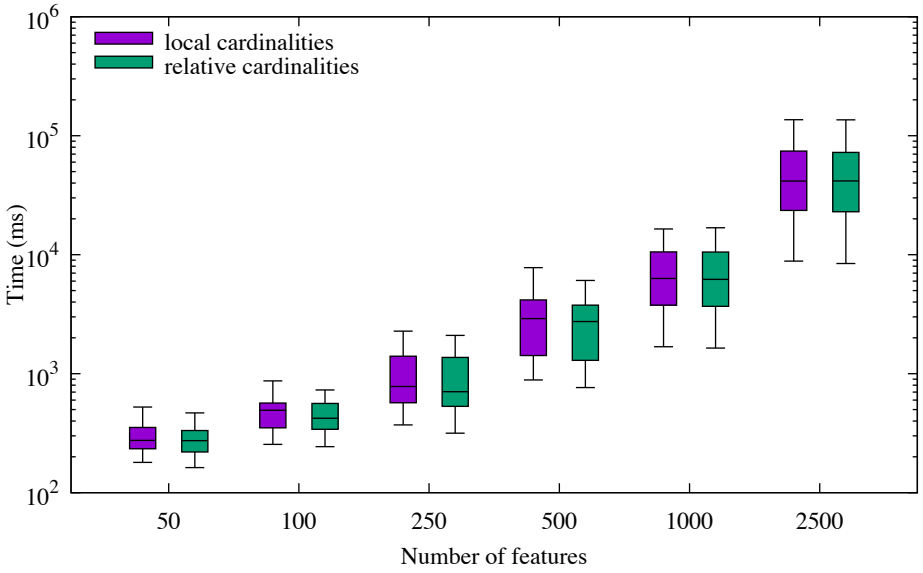


Figure 7.4 Distribution of time to translate a feature model to a CP problem.

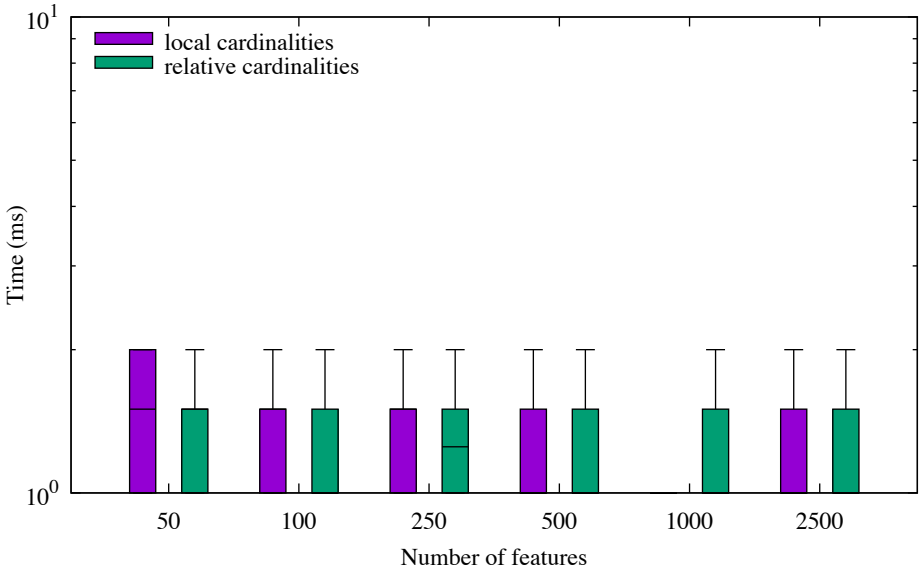


Figure 7.5 Distribution of time to solve the generated CP problem.

compared to the variables and constraints necessary for representing group and feature cardinalities. As illustrated in Figure 7.6, the underlying constraint program size (defined by the number of generated variables and constraints) is virtually the same for feature models with and without relative cardinalities. A relative cardinality will introduce about the same number of variables and constraints of a local cardinality. However, in the generated examples, the number of generated relative cardinalities is very small

compared to the total number of features, hence their limited impact on the translation performance.

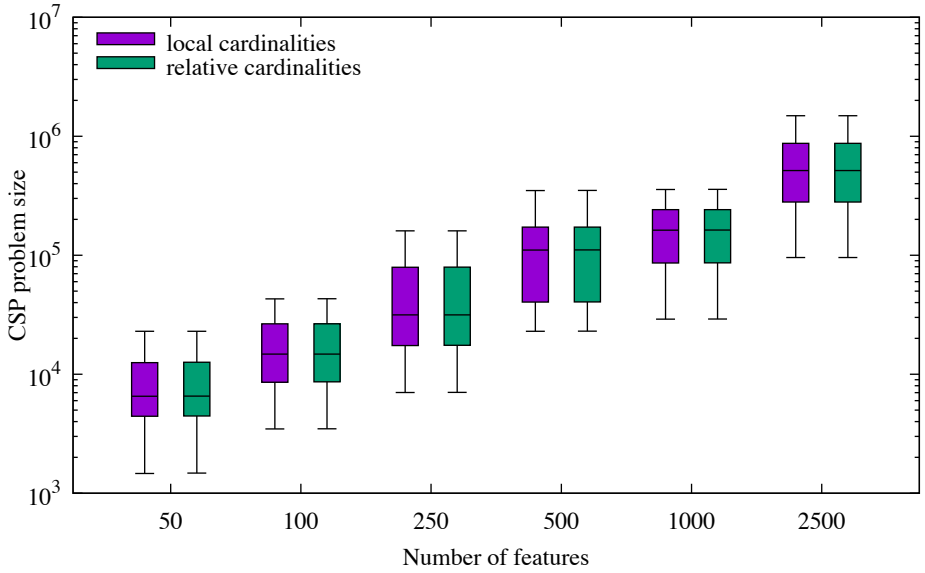


Figure 7.6 Distribution of the underlying constraint program size.

From the results we can also see that the time for translating different feature models with the same number of features can vary greatly. This is because we are dealing with clone-enabled feature models, in which each feature may be instantiable many times, each one with a different decomposition of subfeatures. Hence, according to how the hierarchy was generated and the position in which feature cardinalities were added, the number of possible feature instances can vary greatly and therefore the size of the generated constraint problem varies with it.

For the next experiment, we evaluate the time to check if a configuration conforms to a feature model. Using the translations obtained in the previous experiment we generate valid configurations for feature models in both sets and check their compliance to their respective feature model. Figure 7.7 shows the time distribution for checking a configuration for feature models with local and relative cardinalities. This time includes the translation of the configuration, together with the conversion of feature model constraints to a constraint satisfaction problem, and the resolution of the problem (see Section 4.3.3). From the graph we can see that introducing relative cardinalities did not affect the time to validate a configuration.

For the final experiment, we measured the time to verify the consistency between cardinalities and infer cardinalities that were not described in the model. This includes

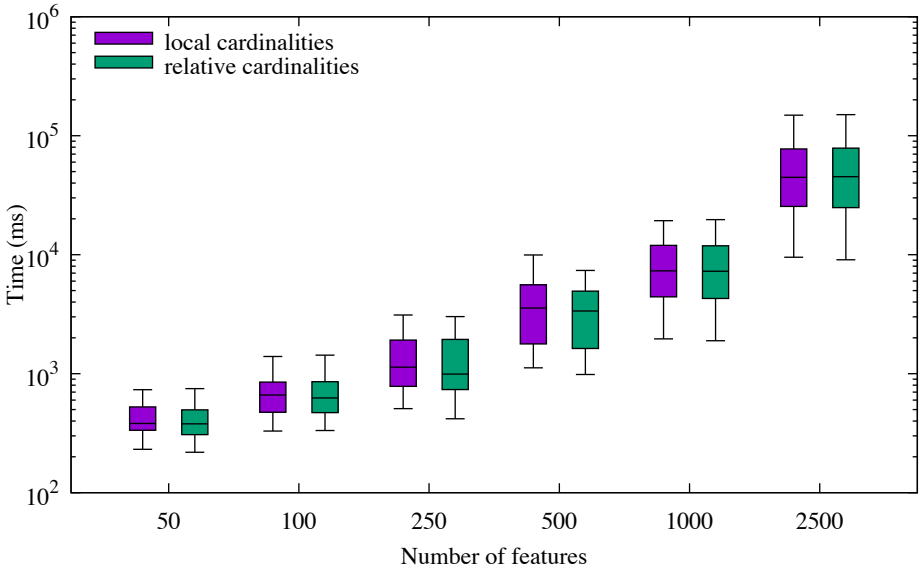


Figure 7.7 Distribution of time to verify the compliance of a configuration to a feature model

the time to translate the consistency conditions described in Section 4.2.2 into a constraint problem and to solve it. This experiment was run only for the set of feature models with relative cardinalities because feature models with only local cardinalities do not require evaluation of relative cardinalities consistency. The average measured time, by the number of features in the model, is shown in Table 7.1.

Table 7.1 Average time for consistency checking and cardinality inference for feature models with relative cardinalities

Features	**Average time (ms)**	**Standard deviation (ms)**
50	120.10	38.30
100	347.53	88.92
250	1,173.07	331.39
500	3,404.63	547.19
1,000	12,089.12	4,438.98
2,500	73,108.42	28,325.58

From the results, we can see that this operation can take substantial time (more than 1 minute on average) for big feature models (with 2500 features). As for other operations, analysis of relative cardinalities consistency depends on the feature model hierarchy. Hence, feature models with higher depth will require more processing time as the cardinality consistency conditions (see Section 4.2.2) are transitive on the parent-children

relationship between features. This operation only needs to be performed during domain design and initial feature model processing. The obtained inferred cardinalities can then be saved and reused during the translation of feature models to a constraint satisfaction problem.

The results obtained indicate that the performance overhead introduced by relative cardinalities is minimal. The number of introduced constraints and variables in the underlying constraint program is about the same required for local feature cardinalities. In addition, most feature models for cloud providers require relative cardinalities in only a relatively small set of features, which further reduces the impact of this extension in this domain. Meanwhile, consistency of relative cardinalities in big and deep feature models can be a threat to the approach, specially in an interactive feature modeling scenario. Still, for the generation or validation of feature model configurations this process only needs to be executed once for verifying the consistency of relative cardinalities and inferring unspecified cardinalities.

7.2.2 Feature Models with Temporal Constraints

Like for relative cardinalities, the experiments described in this section were conducted with the goal of assessing the impact of the introduction of temporal constraints on the analysis of feature models. For this, we evaluate how temporal cardinalities impact the reconfiguration operation (see Sections 5.3.1 and 5.4.2). The reconfiguration operation is executed over an existing feature model configuration and a new set of requirements, which are defined as a partial configuration (see Section 5.3.1), to generate a new complete configuration that is reachable from the current one.

For executing this experiment, we generated 50 different feature models with a structure similar to the ones found in cloud providers. For each feature model, we added features that represent elements commonly found in cloud providers such as application projects, frameworks, computing nodes, available regions and additional services such as databases, message queues, caches, management tools, etc. Later, we augmented these feature model with temporal constraints that associate reconfiguration operations to changes in the configuration of these elements. Examples of such changes include the data center region, computing node size or type, and configuration plan for additional services.

Figure 7.8 depicts an overview of the template used for generating these feature models. For alternative feature groups and features with cardinalities we generated a random number of alternatives and a random upper bound cardinality. For example, in each

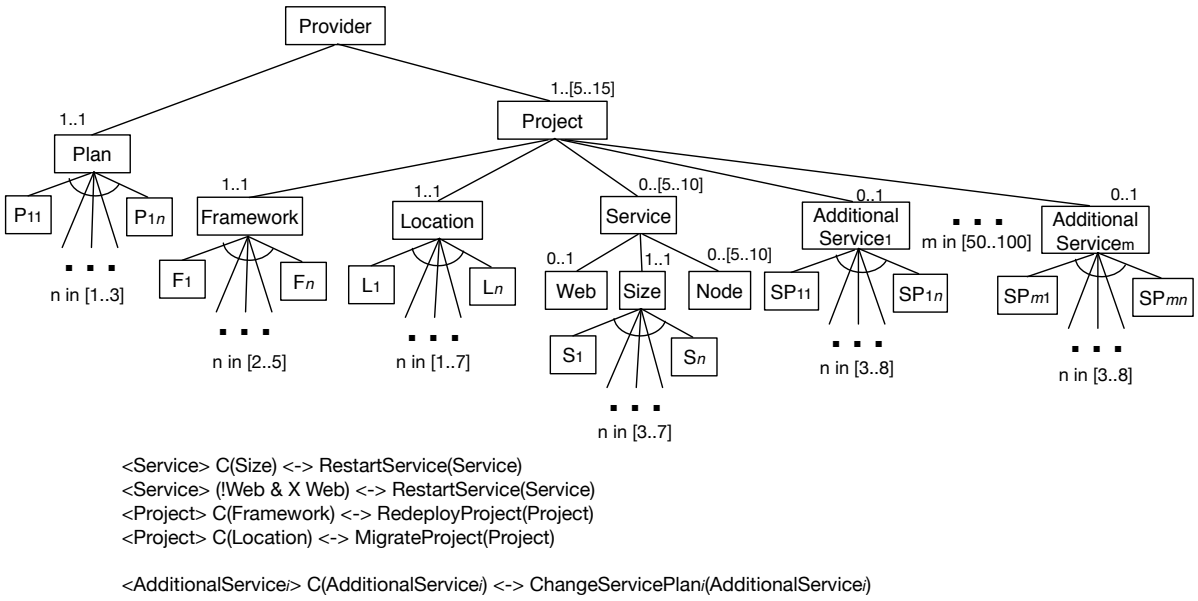


Figure 7.8 Template for generated feature models

generated feature model, the feature `Framework` will have between 2 and 5 different options. Likewise, the cardinality for `Service` feature will have an upper bound between 5 and 10. Besides this, between 50 and 100 additional services, each one with something between 3 to 8 different configurations are also included. Later, we create a copy of these feature models to which we add the reconfiguration constraints shown in Figure 7.8. For each additional service feature (e.g. `AdditionalService`i), we also generate a temporal constraint that associates a configuration change to a reconfiguration operation. By applying these rules, we obtain two sets of feature models, with and without temporal constraints.

In this experiment, we execute the reconfiguration operation for feature models in both groups. For each feature model, we generate an initial configuration and a new partial configuration that can be reachable from the initial one. Then, we measure the time required for analyzing the feature model and generating a new complete configuration. As in the case of previous feature model analysis operations, this also requires translating the feature model into a constraint satisfaction problem and solving it.

Figure 7.9 depicts the time required to translate the generated feature models to a constraint satisfaction problem. As we can see from the values, temporal constraints introduce a substantial overhead on the translation time, which was on average around 2.7 times the time required for translating the corresponding feature model without temporal constraints.

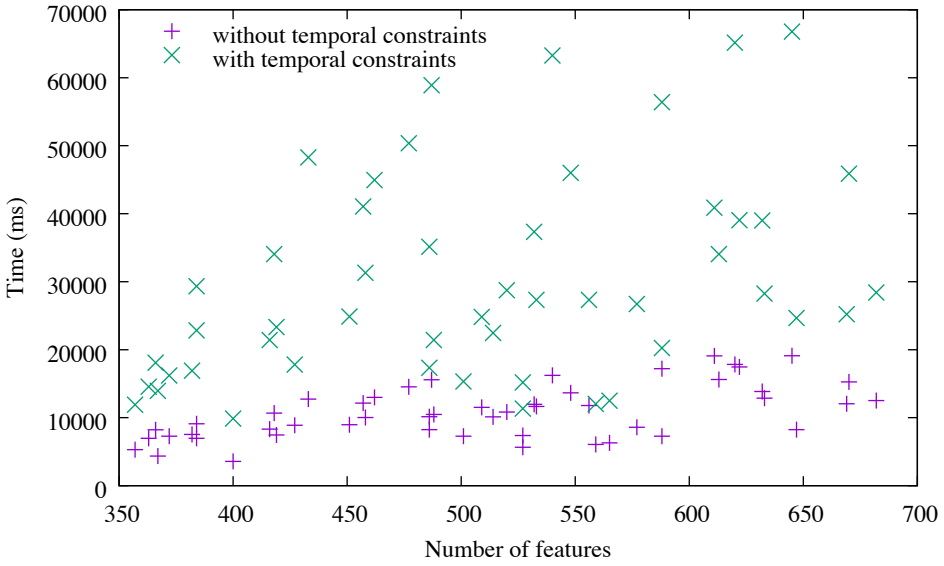


Figure 7.9 Time for translating feature models to constraint programming

This overhead is directly related to the number of constraints and variables that are added to underlying constraint program in order to represent the underlying transition system defined by temporal constraints. Unlike relative cardinalities that introduce only a limited number of additional constraints and variables, temporal constraints may require a much higher number of elements for two main reasons. First the temporal constraints defined using the *change* operator (e.g. `C(Location)`) expand to a larger expression combining all feature group variants. In addition, as we are working with clone-enabled feature models, these extra constraints and variables are generated for each instance of the context feature. These aspects were discussed in detail in Section 5.4.1.

As feature instances are considered, the size of the generated underlying constraint program is directly linked the number of feature instances that can be included in a feature model configuration and the existence of temporal constraints involving these features. Figure 7.10 depicts the relation between the number of feature instances the size of the obtained constraint program for the generated feature models. We can see that for both groups of feature models, the generated constraint program increases linearly with the potential number of instances that can be included in a feature model configuration. However, feature models with temporal constraints will lead to bigger problems, according to the complexity of constraints and the cardinality of the features to which the constraints were applied.

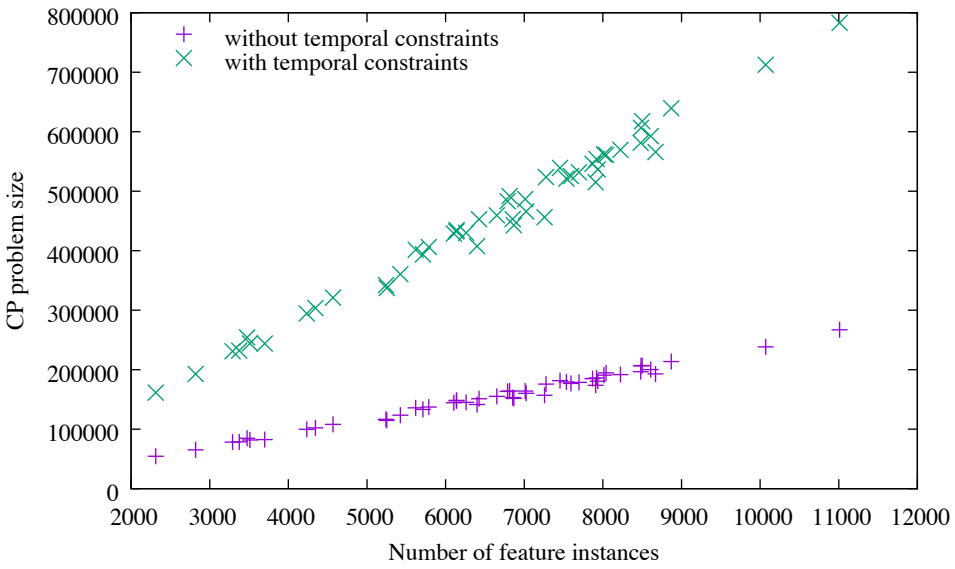


Figure 7.10 Constraint program size

Figures 7.11 and 7.12 depict the time required to translate the generated feature models to constraint programing and solve satisfiability over it according to the CP problem size. We can see that both translation and solving times increase linearly according to the size of the problem. However, the size of problems generated from feature models without temporal constraints is much smaller, leading to faster evaluation.

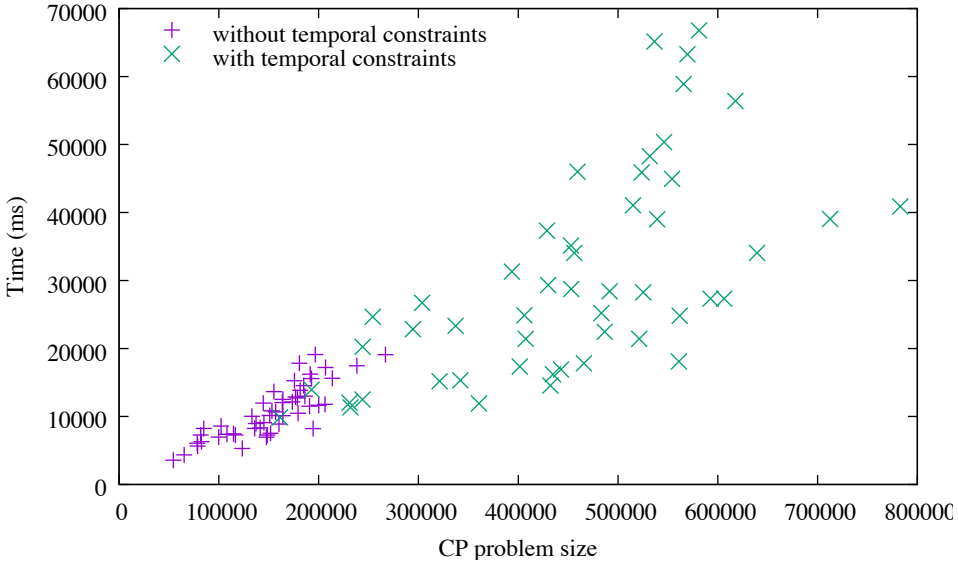


Figure 7.11 Time for translating feature models to CP

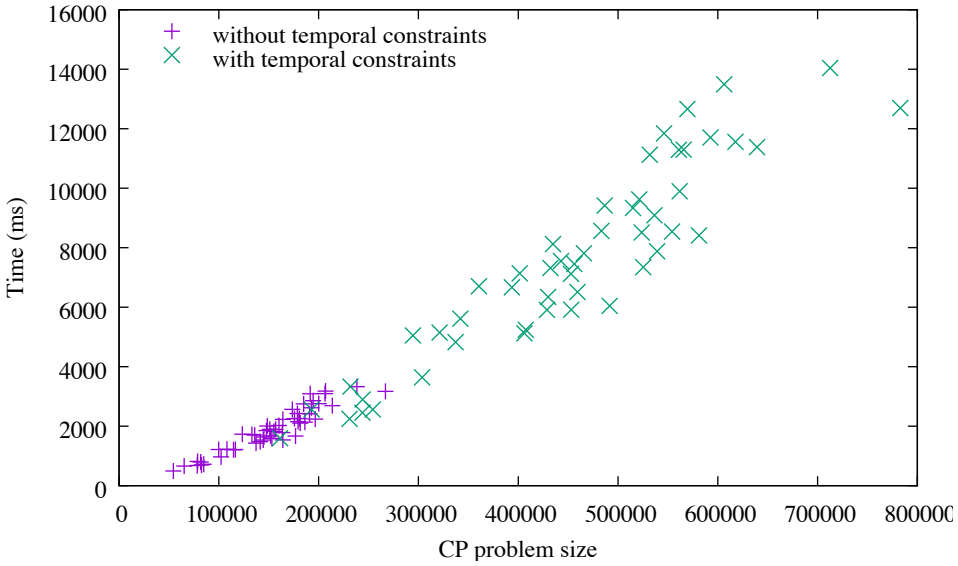


Figure 7.12 Time for finding a reconfiguration by solving the constraint problem

In the conducted experiments, introduction of temporal constraints led to an average increase of about 2.9 times the time required for executing the reconfiguration operation, which includes translating the feature model into constraint programming and solving it. The introduced overhead increases according to the cardinality of features to which temporal constraints are applied and the complexity of the temporal constraints. Hence, the performance and feasibility of their use will depend on the characteristics of the feature model. In the remainder of this section, we evaluate how these introduced variability modeling constructs perform in a more realistic scenario from cloud computing.

7.2.3 Setup and Adaptation of a Multi-Cloud Environment

The previous experiments were conducted to evaluate the impact of introduced variability modeling constructs on the automated analysis of feature models. Here, we present the experiments conducted to evaluate how the proposed approach for setup and adaptation of multi-cloud environments behaves in a more realistic scenario, based on a microservices application.

Case Study Application

The application employed in our case study is the Socks Shop microservices demo [203]. It is an open source demo application designed to demonstrate the development of scalable

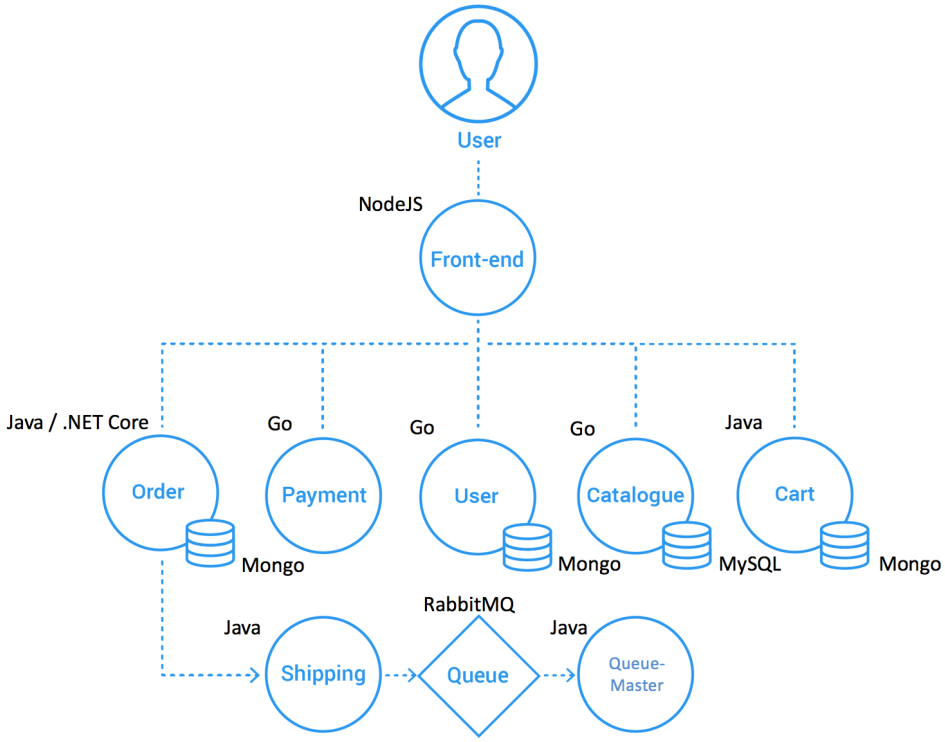


Figure 7.13 Architecture of the Socks Shop microservices demo application [204]

microservices-based applications integrating multiple technologies. Figure 7.13 illustrates its main application services. The Socks Shop is an example of e-commerce application, which includes services for managing product catalog, orders, payment, users, etc. These application services are implemented in different languages and therefore require different functionality from cloud providers.

Based on these application services' needs we defined the requirements for a multi-cloud environments using the DSML proposed in Section 6.3. Besides the application service requirements, we also included the following multi-cloud requirements.

- European users' data should be stored in data center located in western Europe;
- Both the front-end and catalogue services should be simultaneously deployed to two different clouds maintained by different providers. The same clouds should be used for both services as they communicate very frequently.

By combining these requirements, we obtained a multi-cloud environment requirements description that includes 12 services, with 3 cloud variables and 14 instance groups.

Cloud Providers

For this experiment, we are interested in assessing the time required by the approach to generate an initial setup from multi-cloud requirements. For this, we generate random cloud providers offerings descriptions and measure the time required for obtaining a proper multi-cloud environment configuration. To generate a cloud provider definition, we need to define the configuration variability by means of a feature model and the services' offerings through a mapping model, as described in Section 6.4.

For generating the feature models, we employ a template similar to the one employed in Section 7.2.2. In addition to it, we also add relative cardinalities to key features, representing computing nodes and services and allow for the hierarchy of the feature model to assume different structures, in which key features representing user plans, node size, frameworks, among others are associated to different contexts such as global, project, service or can be absent. This is done as an effort to generate a set of feature models that resemble actual cloud providers, which are heterogeneous and do not follow a standard or predefined hierarchy neither provide a common set of features.

In the mapping model, features are assigned to elements from a set of ontology concepts that represent commonly offered services and configuration options from different cloud providers. For example, each feature that represents a location alternative in the feature model is assigned to an ontology concept that corresponds to a random cloud region such as USEast, WesternEurope, AsiaPacific, etc. Likewise, if the generated feature model offers 4 different application frameworks, each one of this will be mapped to a different ontology concept from a pool that includes common development platforms such as Java, Python, PHP, JavaWebContainer, Ruby, etc. We apply this process to generate a set of 15 different cloud providers that are used in this evaluation.

Initial Setup of a Multi-Cloud Environment

Given the multi-cloud environment requirements for the case study application and the set of generated cloud providers, we measure the time to generate an initial configuration. This includes the time to spent on mapping application services' requirements to cloud providers' features, and on analyzing the variability in feature models. This experiment was executed 20 times, and the obtained results are shown in Table 7.2.

From the obtained results we identified that most of the time for generating an initial configuration was spent on mapping application services' requirements to features. Most

Table 7.2 Average time for operations in the generation of a multi-cloud configuration.

Operation	Average time (ms)	Standard deviation (ms)
Map service requirements to feature selections	26,860.00	235.56
Analyze feature models and generate configuration	4,715.22	436.65
Overall process	32,010.11	494.99

of this mapping time is spent on the execution of ontology queries required to identify equivalence relationships between the concepts employed in application requirements description. Still, this time tends to decrease over the search as previously identified relationships are reused across different evaluations. For instance, during this experiment, three different selections of providers were evaluated before finding a matching solution. Therefore, the 26860ms spent on mapping requirements to cloud services correspond to three different executions, which took on average 21661.44ms, 3177.44ms and 2021.11ms respectively. Meanwhile, the analysis of feature models was executed only once.

The obtained results indicate that the performance for the generation of an initial setup is very acceptable as in less than a minute it was possible to generate a multi-cloud environment configuration that conforms to the requirements and to the providers' configuration rules. However, performance is not be the only factor that should be considered during the initial setup of a multi-cloud environment and other aspects such as the quality and costs of the generated configurations should also be considered. Further investigation needs to be conducted to assess the utility of the generated configurations with those crafted by developers as well as if the proposed approach improves their productivity.

Adaptation of a Multi-Cloud Environment

In the previous experiment, we evaluated the generation of a configuration for a multi-cloud environment from its requirements. For this experiment, we are interested in assessing the time required by the approach to generate a reconfiguration plan to adapt a running multi-cloud environment to support a new set of requirements.

For this, we define a set of modifications to be applied to the initial multi-cloud environment requirements. These modifications are used to simulate a change in application needs, which will trigger reevaluation of requirements against cloud providers offerings to generate a reconfiguration plan. These changes were classified into the following tree categories, which correspond to the different adaptation scenarios described in Section 6.6:

- requirement changes that do not imply in any reconfiguration because the current configuration already supports the new requirements;
- requirement changes that are supported by the current provider selection, but require reconfiguring some providers;
- requirement changes that require migrating application services because the current selection does not support new requirements.

For each category, we designed 4 different requirements change scenarios and measured the time required for obtaining a reconfiguration plan (see Section 6.6) from the initial configuration obtained in the previous experiment to a new configuration that supports the new requirements. Each requirement changes scenarios do a small change either in application service requirements, cloud variables conditions and instance groups. Like in the previous experiment, we also run the experiment 20 times and measure the time spent on mapping application services' requirements to cloud providers' features and analyzing the variability. We also employed the same cloud providers generated for the previous experiment. Figure 7.14 depicts the average times obtained from this experiment together with the results obtained for the initial setup.

From the results, we can see that for the first scenario, in which no adaptation is required, that only a small time is spent on mapping the changed application service requirements to features from currently selected providers. As the current feature model configuration already support the new requirements, analysis of variability is not necessary.

In the second scenario, the changed application service requirements are also mapped to a set of supporting features from the currently selected cloud provider. However, unlike the first case, the currently selected features do not support the new requirements and the feature model needs to be reanalyzed to generate a new configuration and identify reconfiguration operations to be executed in the cloud provider.

In the two previous scenarios, the changes introduced in the requirements were already supported by the previous selection of cloud providers. Thus, only the changed application service requirements needed to be remapped to identify the feature selection that supports

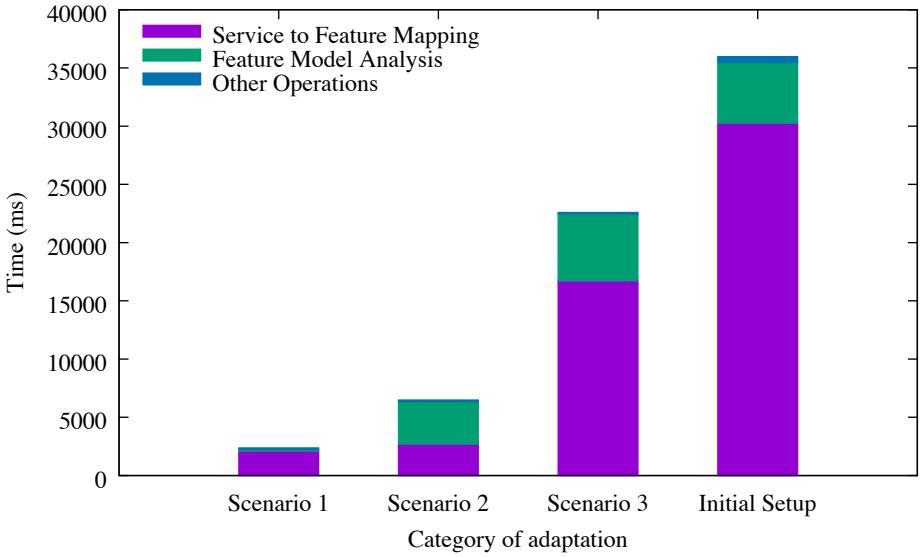


Figure 7.14 Average time for operations in the generation of a reconfiguration plan

the required functionality. However, in the third scenario, we have introduced changes that are not supported by the current selection of cloud providers. Due to constraints on the location and colocation of services, these changes may require a set of services to be migrated to a new cloud provider. This process requires remapping a set of services to features in the newly selected provider, which requires substantial more time than in the previous cases. Despite this, this process still takes less time than the initial setup as not all services need to be migrated to a new provider and a complete reevaluation is not required.

7.3 Discussion

The first thing we can conclude from the results of the experiments is that the overhead introduced by relative cardinalities and temporal constraints depends on the structure of the feature model and to which feature they are applied.

When it comes to relative cardinalities, the performance penalty is very small due to two main factors. First, the number of variables and constraints introduced by a relative cardinality is small, comparable to a local cardinality. Second, the use of relative cardinalities on variability modeling has a role similar to additional cross-tree constraints, in which their goal is to define additional rules that cannot be described by the simple decomposition of features in the feature diagram hierarchy. If the number of additional

constraints or relative cardinalities is very important, compared to the feature model size, it may be a signal that feature modeling is not the most appropriate variability modeling notation.

On the other hand, temporal constraints may lead a much larger number of variables and constraints in the underlying constraint program. Their impact will be directly linked to the complexity of the constraints and the number of feature instances to which they are applied. The decision to use temporal constraints in a DSPL should be based on the characteristics of the system variability and the performance requirements for the adaptation.

In our case study, we identified that most of the time required to generate an initial configuration was spent in mapping application service requirements to features, while the analysis of extended feature models represented only a small part of this process. The time required to generate an initial configuration was on average of 31s and reconfiguration plans were generated in under 20s. However, it is important to point out that for the initial setup, the time required to generate a configuration for a multi-cloud environment is not an issue, especially when compared to the effort that would be required from developers to achieve a comparable result with no tool support.

However, when it comes to the adaptation of a multi-cloud environment the performance may be an issue. Still, it is important to bear in mind that the proposed approach does not directly handle short-term scalability or elasticity of computing resources. The elasticity policy is considered as part of the multi-cloud environment configuration and can be adapted to match a trend on the consumption of resources. Hence, for this kind of adaptation, which is more long-term, the time required for obtaining an adaptation is not a very important issue.

The main threat to the validity of this evaluation is the bias that may be introduced by the generated cloud providers and the selected application. As the performance of feature model analysis operations is related to the structure, cardinalities and constraints of the feature models, the way cloud providers were generated may influence their analysis. In order to minimize this bias, we generated cloud provider offerings that are based on constraints that can be found in cloud providers that were discussed in Chapters 4 and 5 as well as in related work [205, 23].

One of the difficulties to use actual cloud providers is the lack of structured information about their offerings and configuration constraints. In most cases, this information is only described by means of textual documentation in natural language describing the

steps that must be followed by developers to setup or reconfigure a given cloud service. In order to apply this approach in a real use-case, a complete description of cloud provider offerings would be required. We consider that this description should be directly provided by cloud providers as they have a more complete knowledge of their offerings.

7.4 Summary

In this chapter, we presented implementation details about the tool support developed for our approach and the results obtained from its experimental evaluation. First, we presented an overview of the modeling and reasoning mechanisms that were developed to support the setup and adaptation of multi-cloud environment. The developed tools were then used to conduct an experimental evaluation on the impact of the feature modeling constructs we introduced and to evaluate the feasibility and performance in a realistic scenario. The results indicate that the approach is feasible for planning the initial setup or adaptation of multi-cloud environments.

In the following chapter, we summarize the main contributions of this thesis and present the perspectives for future work.

Part IV

Final Remarks

Chapter 8

Conclusions

Cloud computing has changed the way software is developed and consumed by delivering computing capabilities via the Internet. The *pay-as-you-go* model promoted by cloud computing reduced the need for upfront investments in computing infrastructure and personnel, giving more flexibility for organizations to scale their computing resources according to their needs.

Together with its advances, cloud computing also brought new concerns about provider dependence, vendor lock-in and data confidentiality. These concerns motivated the development of multi-cloud systems, which consume resources and services from multiple clouds. By combining cloud services from different providers, cloud customers seek to reduce their dependence on cloud providers.

However, developing multi-cloud systems is still very complex and challenging, mainly due to the heterogeneity across cloud providers as well as the lack of standardization and interoperability. The lack of supporting tools is also a limiting factor, which hinders further adoption of multi-cloud computing. Though several approaches have been proposed to deal with many of the issues related to the development of multi-cloud systems, the variability that exists in the configuration of cloud environments is often overlooked. This negligence implies that configuration rules imposed by cloud providers are not taken into account, which may ultimately lead to incorrect or invalid cloud deployments.

Based on this limitation, in this thesis we have proposed an approach to automate the setup and adaptation of multi-cloud environments that embraces variability management. This approach is aimed at shielding developers from the complexities related to the

configuration of cloud providers and the heterogeneity across their offerings. It achieves this goal by providing means for developers to describe their application needs in terms of the functionality and properties required from a multi-cloud environment; and to automatically generate a configuration (or reconfiguration plan) for setting up a new environment (or adapting a running one) to match these needs.

Variability management is employed by the approach to ensure that generated configurations or reconfiguration plans comply with cloud providers' configuration rules. In the first two contributions, described in Chapters 4 and 5, we proposed extended variability management mechanisms to handle the variability that arises in the configuration of cloud providers. In the third contribution, described in Chapter 6, previous contributions on variability management were integrated into an approach that handles the heterogeneity across providers' offerings and supports the generation of configurations or reconfiguration plans for a multi-cloud environment from a description of its requirements.

This chapter summarizes the contributions of this thesis and how they contribute to achieve the goals of this thesis. Additionally, it provides a discussion about its limitations and perspectives for future work.

8.1 Contribution Summary

The contributions of this work are aimed at targeting the problem identified in Chapter 1. While the first two contributions are centered around variability management, the third one integrates these previous contributions into an approach for automating the setup and adaptation of multi-cloud environments.

Feature Models and Relative Cardinalities

In Chapter 4 we demonstrated how feature model can be extended to support multiple interpretations of feature cardinality scope. We redefined syntax and semantics of cardinality-based feature models to support relative cardinalities and investigated how these changes affected consistency of cardinalities and additional cross-tree constraints.

In addition, we demonstrated how a method for automated analysis of clone-enabled cardinality-based feature models could be extended to handle relative cardinalities. This method is based on a translation to a constraint program, which can be fed into a constraint programming solver to either validate or generate feature model configurations.

The amount of extra variables and constraints introduced into the constraint program to model a relative cardinality is comparable to the amount required by a local feature cardinality. Because of this, the impact of relative cardinalities on the performance of automated analysis operations is small.

The use of relative cardinalities to model variability in cloud environments, enabled to capture constraints that were not possible to model with previous modeling constructs.

Feature Models and Temporal Constraints

In Chapter 5, we proposed a method for integrating temporal constraints with feature models for the definition of Dynamic Software Product Lines. Temporal constraints provide a declarative way to specify constraints on how an adaptive system built as a DSPL should transition between its valid configurations. Using concepts from model-checking theory, we demonstrated how temporal constraints can be integrated into variability modeling.

Besides this, we demonstrated how automated analysis of feature models can be extended to handle temporal constraints. We then extended the method proposed for handling clone-enabled cardinality-based feature models, developed in the previews contribution, to support reasoning over temporal constraints. Temporal constraints may introduce a large number of variables and constraints in the corresponding constraint program. Hence, their feasibility will depend on the complexity of constraints, the structure of the feature model and the performance requirements of the application.

By employing temporal constraints, it was possible to model complex reconfiguration rules that arise in cloud computing environments, and that could not be represented by current variability modeling approaches.

Automated Setup and Adaptation of Multi-Cloud Environments

In our final contribution, presented in Chapter 6, we presented an approach for generating a configuration or reconfiguration plan for a multi-cloud environment from a description of its requirements. This approach leverages the two previous contributions on managing variability of cloud environments and integrates them with a mechanism for semantically mapping provider-independent requirements to provider-specific cloud services.

The use of variability management mechanisms ensures the generation of correct configurations, which comply with cloud providers' configuration and reconfiguration rules. Meanwhile, semantic reasoning is employed to establish a mapping from requirements to cloud providers' offerings. These two mechanisms are integrated into a search algorithm that looks for candidate cloud providers that support the described requirements and then generates a proper configuration.

Together, these contributions support the automated setup and adaptation of multi-cloud environments by handling the complexities linked to the variability in the configuration environments and the heterogeneity across cloud computing providers. It provides means to specify application needs using provider-independent concepts and a systematic way for mapping these requirements to actual cloud service offerings from different providers. In addition, it provides means for managing the variability in the configuration of cloud providers, thus ensuring the generation of consistent configurations.

8.2 Perspectives

In this section we discuss about aspects related to the setup and adaptation of multi-cloud environments that were not handled by our approach that could be investigated in future research.

Generation of optimized configurations

One of the reasons for consuming resources from multiple providers is to improve the quality of service or reduce costs by selecting services that are best fitted for application needs. One of the difficulties in generating optimized configurations is to obtain information concerning the quality of service and costs of cloud services. As for other aspects, there is a big heterogeneity in how these information is made available across different providers and this kind of information can be based on different concepts, attributes, and measuring units, further complicating their comparison. Besides this, the actual costs and performance of cloud services may depend on the actual system utilization, which is also difficult to predict.

Another difficulty, is the increased complexity of solving an optimization problem rather than just verifying the compliance of the application. This difficulty arises when looking

for an optimal selection of a cloud providers, but also when looking for an optimized configuration for each selected provider.

Monitoring and analysis of multi-cloud environment changes

In order to build a self-adaptive multi-cloud environment, we need to be capable of monitoring context changes and identify when the system needs to adapt to continue meeting its goals. One challenge that arises in multi-cloud environments is monitoring heterogeneous cloud environments. Each cloud provider may provide different monitoring capabilities, based on different concepts and measuring units, which complicates their analysis. The approach proposed in this work only handles the generation of a configuration or reconfiguration plan from a description of the application needs. Hence, it is still necessary to investigate how to determine new application needs from the analysis of context changes.

Handling the evolution of cloud market

Cloud providers are constantly updating their service offer by introducing new services, deprecating old ones, changing configuration options or the way cloud services can be integrated into a cloud environment. As cloud offerings evolve, a multi-cloud environment may need to be reconfigured in order to keep it up to date with supported cloud services. Besides this, newly available cloud services may be best fitted to application needs. Evolution of cloud offerings is related to the evolution of variability in DSPLs [138, 139]. The main issue that arises when cloud offerings evolve is how to keep running cloud environments synchronized to cloud provider offerings.

Unbounded feature cardinalities

Though the feature models employed in our approach only employ bounded finite cardinalities, in many cloud providers resource limits for some services are not specified. One of the main properties of the cloud computing paradigm is this illusion of infinite resources [2]. This does not imply that the resources are actually infinite, but rather that their number is much larger than what is generally need by most applications. Handling feature models with unbounded cardinalities would enable to better capture the variability in such providers and support their management by means of the approach proposed in this thesis. Unbounded feature cardinalities require a different method for

analyzing feature models as it is not possible to represent an infinite number of possible feature instances using only standard constraint programing constructs.

References

[1] Peter Mell and Tim Grance. The NIST definition of cloud computing. Technical report, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, 2011.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010. ISSN 0001-0782. doi: 10.1145/1721654.1721672. URL http://doi.acm.org/10.1145/1721654.1721672.

[3] Carlos M. DaSilva, Peter Trkman, Kevin Desouza, and Jaka Lindič. Disruptive technologies: a business model perspective on cloud computing. *Technology Analysis & Strategic Management*, 25(10):1161–1173, 2013. doi: 10.1080/09537325.2013.843661. URL http://dx.doi.org/10.1080/09537325.2013.843661.

[4] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. Cloud computing — the business perspective. *Decision Support Systems*, 51(1):176 – 189, 2011. ISSN 0167-9236. doi: https://doi.org/10.1016/j.dss.2010.12.006. URL http://www.sciencedirect.com/science/article/pii/S0167923610002393.

[5] Peter K. Ross and Michael Blumenstein. Cloud computing as a facilitator of sme entrepreneurship. *Technology Analysis & Strategic Management*, 27(1):87–101, 2015. doi: 10.1080/09537325.2014.951621. URL http://dx.doi.org/10.1080/09537325.2014.951621.

[6] M. A. AlZain, E. Pardede, B. Soh, and J. A. Thom. Cloud computing security: From single to multi-clouds. In *2012 45th Hawaii International Conference on System Sciences*, pages 5490–5499, Jan 2012. doi: 10.1109/HICSS.2012.153.

[7] Nikolay Grozev and Rajkumar Buyya. Inter-cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience*, 44(3):369–390, 2014. ISSN 1097-024X. doi: 10.1002/spe.2168. URL http://dx.doi.org/10.1002/spe.2168.

[8] Dana Petcu. Consuming resources and services from multiple clouds. *Journal of Grid Computing*, 12(2):321–345, 2014.

[9] Adel Nadjaran Toosi, Rodrigo N. Calheiros, and Rajkumar Buyya. Interconnected cloud computing environments: Challenges, taxonomy, and survey. *ACM Comput. Surv.*, 47(1):7:1–7:47, May 2014. ISSN 0360-0300. doi: 10.1145/2593512. URL http://doi.acm.org/10.1145/2593512.

[10] Yehia Elkhatib. Mapping cross-cloud systems: Challenges and opportunities. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, pages 77–83, Berkeley, CA, USA, 2016. USENIX Association. URL http://dl.acm.org/citation.cfm?id=3027041.3027054.

[11] Dana Petcu, Beniamino Di Martino, Salvatore Venticinque, Massimiliano Rak, Tamás Máhr, Gorka Esnal Lopez, Fabrice Brito, Roberto Cossu, Miha Stopar, Svatopluk Šperka, et al. Experiences in building a mosaic of clouds. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):12, 2013.

[12] Antonio Brogi, Ahmad Ibrahim, Jacopo Soldani, José Carrasco, Javier Cubo, Ernesto Pimentel, and Francesco D'Andria. Seaclouds: A european project on seamless management of multi-cloud applications. *SIGSOFT Softw. Eng. Notes*, 39(1):1–4, February 2014. ISSN 0163-5948. doi: 10.1145/2557833.2557844. URL http://doi.acm.org/10.1145/2557833.2557844.

[13] N. Ferry, H. Song, A. Rossini, F. Chauvel, and A. Solberg. Cloudmf: Applying mde to tame the complexity of managing multi-cloud applications. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 269–277, Dec 2014. doi: 10.1109/UCC.2014.36.

[14] Nikolay Grozev and Rajkumar Buyya. Multi-cloud provisioning and load distribution for three-tier applications. *ACM Trans. Auton. Adapt. Syst.*, 9(3): 13:1–13:21, October 2014. ISSN 1556-4665. doi: 10.1145/2662112. URL http://doi.acm.org/10.1145/2662112.

[15] Subharthi Paul, Raj Jain, Mohammed Samaka, and Jianli Pan. Application delivery in multi-cloud environments using software defined networking. *Computer Networks*, 68(Supplement C):166 – 186, 2014. ISSN 1389-1286. doi: https://doi.org/10.1016/j.comnet.2013.12.005. URL http://www.sciencedirect.com/science/article/pii/S1389128614000826. Communications and Networking in the Cloud.

[16] Pooyan Jamshidi, Claus Pahl, and Nabor C. Mendonça. Pattern-based multi-cloud architecture migration. *Software: Practice and Experience*, 47(9):1159–1184, 2017. ISSN 1097-024X. doi: 10.1002/spe.2442. URL http://dx.doi.org/10.1002/spe.2442. spe.2442.

[17] Xiaogang Wang, Jian Cao, and Yang Xiang. Dynamic cloud service selection using an adaptive learning mechanism in multi-cloud computing. *Journal of Systems and Software*, 100(Supplement C):195 – 210, 2015. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2014.10.047. URL http://www.sciencedirect.com/science/article/pii/S0164121214002398.

[18] Leonard Heilig, Eduardo Lalla-Ruiz, and Stefan Voß. A cloud brokerage approach for solving the resource management problem in multi-cloud environments. *Computers & Industrial Engineering*, 95(Supplement C):16 – 26, 2016. ISSN 0360-8352. doi: https://doi.org/10.1016/j.cie.2016.02.015. URL http://www.sciencedirect.com/science/article/pii/S0360835216300420.

[19] Leonard Heilig, Rajkumar Buyya, and Stefan Voß. Location-aware brokering for consumers in multi-cloud computing environments. *Journal of Network and Computer Applications*, 95(Supplement C):79 – 93, 2017. ISSN 1084-8045. doi:

https://doi.org/10.1016/j.jnca.2017.07.010. URL http://www.sciencedirect.com/science/article/pii/S1084804517302333.

[20] Elisabetta Di Nitto, Peter Matthews, Dana Petcu, and Arnor Solberg. *Model-Driven Development and Operation of Multi-Cloud Applications: The MODAClouds Approach*. Springer Publishing Company, Incorporated, 1st edition, 2017. ISBN 3319460307, 9783319460307. doi: 10.1007/978-3-319-46031-4.

[21] Alessandro Ferreira Leite, Vander Alves, Genaína Nunes Rodrigues, Claude Tadonki, Christine Eisenbeis, and Alba Cristina Magalhaes Alves de Melo. Dohko: an autonomic system for provision, configuration, and management of inter-cloud environments based on a software product line engineering method. *Cluster Computing*, 20(3):1951–1976, Sep 2017. ISSN 1573-7543. doi: 10.1007/s10586-017-0897-1. URL https://doi.org/10.1007/s10586-017-0897-1.

[22] Jose Luis Lucas Simarro, Rafael Moreno-Vozmediano, Ruben S Montero, and Ignacio Martín Llorente. Dynamic placement of virtual machines for cost optimization in multi-cloud environments. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 1–7. IEEE, 2011.

[23] Jesús García-Galán, Pablo Trinidad, Omer F. Rana, and Antonio Ruiz-Cortés. Automated configuration support for infrastructure migration to the cloud. *Future Generation Computer Systems*, 55:200–212, 2016.

[24] Erik Wittern, Jörn Kuhlenkamp, and Michael Menzel. Cloud service selection based on variability modeling. *Service-Oriented Computing*, pages 127–141, 2012.

[25] 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. ISSN 0167-739X. doi: 10.1016/j.future.2011.05.009. URL http://dx.doi.org/10.1016/j.future.2011.05.009.

[26] Clément Quinton, Daniel Romero, and Laurence Duchien. SALOON: a platform for selecting and configuring cloud environments. *Software: Practice and Experience*, 46(1):55–78, 2016. ISSN 1097-024X. doi: 10.1002/spe.2311.

[27] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 3642063640, 9783642063640.

[28] N. Ferry, A. Rossini, F. Chauvel, B. Morin, and A. Solberg. Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 887–894, June 2013. doi: 10.1109/CLOUD.2013.133.

[29] Dana Petcu. Multi-cloud: Expectations and current approaches. In *Proc. International Workshop on Multi-cloud Applications and Federated Clouds ((MultiCloud '13)*, pages 1–6, Prague, Czech Republic, 2013. ISBN 978-1-4503-2050-4.

[30] Thomas Erl. *SOA: Principles of Service Design*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2007. ISBN 9780132344821.

[31] M Fowler. Microservices. http://martinfowler.com/articles/microservices.html, 2014.

[32] G. Sousa, W. Rudametkin, and L. Duchien. Automated setup of multi-cloud environments for microservices applications. In *Proceedings of the 9th International Conference on Cloud Computing*, CLOUD'16, pages 327–334, June 2016. doi: 10.1109/CLOUD.2016.0051. URL http://doi.org/10.1109/CLOUD.2016.0051.

[33] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. Extending dynamic software product lines with temporal constraints. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '17, pages 129–139, May 2017. doi: 10.1109/SEAMS.2017.6. URL http://doi.org/10.1109/SEAMS.2017.6.

[34] Gustavo Sousa, Walter Rudametkin, and Laurence Duchien. Extending feature models with relative cardinalities. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 79–88, September 2016. doi: 10.1145/2934466.2934475. URL http://doi.acm.org/10.1145/2934466.2934475.

[35] 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 Generation computer systems*, 25 (6):599–616, 2009.

[36] Keith Jeffery, Geir Horn, and Lutz Schubert. A vision for better cloud applications. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, pages 7–12. ACM, 2013.

[37] George Baryannis, Panagiotis Garefalakis, Kyriakos Kritikos, Kostas Magoutis, Antonis Papaioannou, Dimitris Plexousakis, and Chrysostomos Zeginis. Lifecycle management of service-based applications on multi-clouds: a research roadmap. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, pages 13–20. ACM, 2013.

[38] Chrysostomos Zeginis, Kyriakos Kritikos, Panagiotis Garefalakis, Konstantina Konsolaki, Kostas Magoutis, and Dimitris Plexousakis. Towards cross-layer monitoring of multi-cloud service-based applications. In *European Conference on Service-Oriented and Cloud Computing*, pages 188–195. Springer, 2013.

[39] Kyriakos Kritikos and Dimitris Plexousakis. Multi-cloud application design through cloud service composition. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 686–693. IEEE, 2015.

[40] Jörg Domaschka, Frank Griesinger, Daniel Baur, and Alessandro Rossini. Beyond mere application structure thoughts on the future of cloud orchestration tools. *Procedia Computer Science*, 68:151–162, 2015.

[41] Daniel Baur and Jörg Domaschka. Experiences from building a cross-cloud orchestration tool. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, CrossCloud '16, pages 4:1–4:6, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4294-0. doi: 10.1145/2904111.2904116. URL http://doi.acm.org/10.1145/2904111.2904116.

[42] Alessandro Rossini, Kiriakos Kritikos, Nikolay Nikolov, Jörg Domaschka, Frank Griesinger, Daniel Seybold, Daniel Romero, Michal Orzechowski, Georgia Kapitsaki, and Achilleas Achilleos. The cloud application modelling and execution language (camel). Technical report, PaaSage consortium, 2017.

[43] Danilo Ardagna, Elisabetta Di Nitto, Giuliano Casale, Dana Petcu, Parastoo Mohagheghi, Sébastien Mosser, Peter Matthews, Anke Gericke, Cyril Ballagny, Francesco D'Andria, et al. Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds. In *Proceedings of the 4th international workshop on modeling in software engineering*, pages 50–56. IEEE Press, 2012.

[44] Michele Guerriero, Michele Ciavotta, Giovanni Paolo Gibilisco, and Danilo Ardagna. A model-driven devops framework for qos-aware cloud applications. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2015 17th International Symposium on*, pages 345–351. IEEE, 2015.

[45] Alexander Bergmayr, Alessandro Rossini, Nicolas Ferry, Geir Horn, Leire Orue-Echevarria, Arnor Solberg, and Manuel Wimmer. The evolution of cloudml and its manifestations. In *Proceedings of the 3rd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, pages 1–6, 2015.

[46] Nicolas Ferry, Franck Chauvel, Alessandro Rossini, Brice Morin, and Arnor Solberg. Managing multi-cloud systems with cloudmf. In *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, pages 38–45. ACM, 2013.

[47] Nicolas Ferry, Gunnar Brataas, Alessandro Rossini, Franck Chauvel, and Arnor Solberg. Towards bridging the gap between scalability and elasticity. In *CLOSER*, pages 746–751, 2014.

[48] Nicolas Ferry, Franck Chauvel, Hui Song, and Arnor Solberg. Continous deployment of multi-cloud systems. In *Proceedings of the 1st International Workshop on Quality-Aware DevOps*, pages 27–28. ACM, 2015.

[49] Beniamino Di Martino, Dana Petcu, Roberto Cossu, Pedro Goncalves, Tamás Máhr, and Miguel Loichate. Building a mosaic of clouds. In *European Conference on Parallel Processing*, pages 571–578. Springer, 2010.

[50] Salvatore Venticinque, Rocco Aversa, Beniamino Di Martino, Massimilano Rak, and Dana Petcu. A cloud agency for sla negotiation and management. In *European Conference on Parallel Processing*, pages 587–594. Springer, 2010.

[51] Massimiliano Rak, Salvatore Venticinque, Gorka Echevarria, Gorka Esnal, et al. Cloud application monitoring: The mosaic approach. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 758–763. IEEE, 2011.

[52] Francesco Moscato, Rocco Aversa, Beniamino Di Martino, Teodor-Florin Fortiş, and Victor Munteanu. An analysis of mosaic ontology for cloud resources annotation. In *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, pages 973–980. IEEE, 2011.

[53] Dana Petcu, Ciprian Crăciun, Marian Neagul, Iñigo Lazcanotegui, and Massimiliano Rak. Building an interoperability api for sky computing. In *High Performance*

Computing and Simulation (HPCS), 2011 International Conference on, pages 405–411. IEEE, 2011.

[54] G. Cretella and B. Di Martino. Towards a semantic engine for cloud applications development. In *2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 198–203, July 2012. doi: 10.1109/CISIS.2012.159.

[55] Dana Petcu, Silviu Panica, Călin Şandru, Ciprian Dorin Crăciun, and Marian Neagul. Experiences in building an event-driven and deployable platform as a service. In *International Conference on Web Information Systems Engineering*, pages 666–672. Springer, 2012.

[56] Alba Amato, Luca Tasquier, and Adrian Copie. Vendor agents for iaas cloud interoperability. *Intelligent Distributed Computing VI*, pages 271–280, 2013.

[57] Dana Petcu, Georgiana Macariu, Silviu Panica, and Ciprian Crăciun. Portable cloud applications—from theory to practice. *Future Generation Computer Systems*, 29(6):1417–1430, 2013.

[58] Victor Ion Munteanu, Călin Şandru, and Dana Petcu. Multi-cloud resource management: cloud service interfacing. *Journal of Cloud Computing*, 3(1):3, 2014.

[59] Alba Amato, Giuseppina Cretella, Beniamino Di Martino, Luca Tasquier, and Salvatore Venticinque. Semantic engine and cloud agency for vendor agnostic retrieval, discovery, and brokering of cloud services. In *International Conference on Intelligent Cloud Computing*, pages 8–25. Springer, 2014.

[60] Francesco DAndria, Stefano Bocconi, Jesus Gorronogoitia Cruz, James Ahtes, and Dimitris Zeginis. Cloud4soa: Multi-cloud application management across paas offerings. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 407–414. IEEE, 2012.

[61] Eleni Kamateri, Nikolaos Loutas, Dimitris Zeginis, James Ahtes, Francesco D'Andria, Stefano Bocconi, Panagiotis Gouvas, Giannis Ledakis, Franco Ravagli, Oleksandr Lobunets, et al. Cloud4soa: A semantic-interoperability paas solution for multi-cloud platform management and portability. In *European Conference on Service-Oriented and Cloud Computing*, pages 64–78. Springer, 2013.

[62] Antonio Corradi, Luca Foschini, Alessandro Pernafini, Filippo Bosi, Vincenzo Laudizio, and Maria Seralessandri. Cloud paas brokering in action: The cloud4soa management infrastructure. In *Vehicular Technology Conference (VTC Fall), 2015 IEEE 82nd*, pages 1–7. IEEE, 2015.

[63] Fawaz Paraiso, Nicolas Haderer, Philippe Merle, Romain Rouvoy, and Lionel Seinturier. A federated multi-cloud paas infrastructure. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 392–399. IEEE, 2012.

[64] Fawaz Paraiso, Philippe Merle, and Lionel Seinturier. soCloud: a service-oriented component-based PaaS for managing portability, provisioning, elasticity, and high availability across multiple clouds. *Computing*, 98(5):539–565, 2016.

[65] Antonio Brogi, Michela Fazzolari, Ahmad Ibrahim, Jacopo Soldani, Jose Carrasco, Javier Cubo, Francisco Durán, Ernesto Pimentel, Elisabetta Di Nitto, and Francesco D Andria. Adaptive management of applications across multiple clouds: The seaclouds approach. *CLEI Electronic Journal*, 18(1):2–2, 2015.

[66] Marc Oriol, Diego Perez, Simone Zenzaro, Mattia Buccarella, and Javi Cubo. Seaclouds project d3.3 - seaclouds discovery and adaptation components prototype. Technical Report D3.3, SeaClouds Project, August 2015.

[67] Andrea Turli, Diego Perez, and Javier Cubo. Seaclouds project d5.4.3 - final version of sw platform. Technical Report D5.4.3, SeaClouds Project, March 2016.

[68] Antonio Brogi, Jose Carrasco, Javier Cubo, Francesco D'Andria, Elisabetta Di Nitto, Michele Guerriero, Diego Pérez, Ernesto Pimentel, and Jacopo Soldani. Seaclouds: An open reference architecture for multi-cloud governance. In *Software Architecture: 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28–December 2, 2016, Proceedings 10*, pages 334–338. Springer, 2016.

[69] Autonomic Computing et al. An architectural blueprint for autonomic computing. *IBM White Paper*, 31, 2006.

[70] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Tosca: Portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer, 2014.

[71] A. Ferreira Leite, V. Alves, G. Nunes Rodrigues, C. Tadonki, C. Eisenbeis, and A.C. Magalhaes Alves de Melo. Automating resource selection and configuration in inter-clouds through a software product line method. In *Proc. 8th IEEE Int. Conf. Cloud Computing*, pages 726–733, 2015.

[72] T. Liu, Y. Katsuno, K. Sun, Y. Li, T. Kushida, Y. Chen, and M. Itakura. Multi cloud management for unified cloud services across cloud sites. In *2011 IEEE International Conference on Cloud Computing and Intelligence Systems*, pages 164–169, Sept 2011. doi: 10.1109/CCIS.2011.6045053.

[73] Linh Manh Pham, Alain Tchana, Didier Donsez, Vincent Zurczak, Pierre-Yves Gibello, and Noel De Palma. An adaptable framework to deploy complex applications onto multi-cloud platforms. In *Computing & Communication Technologies-Research, Innovation, and Vision for the Future (RIVF), 2015 IEEE RIVF International Conference on*, pages 169–174. IEEE, 2015.

[74] Linh Manh Pham, Alain Tchana, Didier Donsez, Noel De Palma, Vincent Zurczak, and Pierre-Yves Gibello. Roboconf: a hybrid cloud orchestrator to deploy complex applications. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 365–372. IEEE, 2015.

[75] Duc-Hung Le, Hong-Linh Truong, Georgiana Copil, Stefan Nastic, and Schahram Dustdar. Salsa: A framework for dynamic configuration of cloud services. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 146–153. IEEE, 2014.

[76] Marc E Frincu and Ciprian Craciun. Multi-objective meta-heuristics for scheduling applications with high availability requirements and cost constraints in multi-

cloud environments. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 267–274. IEEE, 2011.

[77] Johan Tordsson, Rubén S Montero, Rafael Moreno-Vozmediano, and Ignacio M Llorente. Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers. *Future Generation Computer Systems*, 28(2):358–367, 2012.

[78] Jose Luis Lucas-Simarro, Rafael Moreno-Vozmediano, Ruben S Montero, and Ignacio M Llorente. Cost optimization of virtual infrastructures in dynamic multi-cloud scenarios. *Concurrency and Computation: Practice and Experience*, 27(9): 2260–2277, 2015.

[79] Sanjaya K Panda and Prasanta K Jana. Efficient task scheduling algorithms for heterogeneous multi-cloud environment. *The Journal of Supercomputing*, 71(4): 1505–1533, 2015.

[80] P. Silva, C. Perez, and F. Desprez. Efficient heuristics for placing large-scale distributed applications on multiple clouds. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 483–492, May 2016. doi: 10.1109/CCGrid.2016.77.

[81] Yashaswi Singh, Farah Kandah, and Weiyi Zhang. A secured cost-effective multi-cloud storage in cloud computing. In *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, pages 619–624. IEEE, 2011.

[82] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 247–258. ACM, 2013.

[83] Everton Cavalcante, André Almeida, Thais Batista, Nélio Cacho, Frederico Lopes, Flavia C. Delicato, Thiago Sena, and Paulo F. Pires. Exploiting software product lines to develop cloud computing applications. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 179–187, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1095-6. doi: 10.1145/2364412.2364442. URL http://doi.acm.org/10.1145/2364412.2364442.

[84] André Almeida, Everton Cavalcante, Thais Batista, Nélio Cacho, Frederico Lopes, Flávia Coimbra Delicato, and Paulo F Pires. Dynamic adaptation of cloud computing applications. In *SEKE*, pages 67–72, 2013.

[85] Ralph Mietzner, Andreas Metzger, Frank Leymann, and Klaus Pohl. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, PESOS '09, pages 18–25, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3716-0. doi: 10.1109/PESOS.2009.5068815. URL http://dx.doi.org/10.1109/PESOS.2009.5068815.

[86] Stefan T. Ruehl and Urs Andelfinger. Applying software product lines to create customizable software-as-a-service applications. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 16:1–16:4, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0789-5. doi: 10.1145/2019136.2019154. URL http://doi.acm.org/10.1145/2019136.2019154.

[87] Stefan T Ruehl, Urs Andelfinger, Andreas Rausch, and Stephan AW Verclas. Toward realization of deployment variability for software-as-a-service applications. In *Cloud computing (cloud), 2012 ieee 5th international conference on*, pages 622–629. IEEE, 2012.

[88] Julia Schroeter, Peter Mucha, Marcel Muth, Kay Jugel, and Malte Lochau. Dynamic configuration management of cloud-based applications. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 171–178, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1095-6. doi: 10.1145/2364412.2364441. URL http://doi.acm.org/10.1145/2364412.2364441.

[89] Stefan Walraven, Dimitri Van Landuyt, Eddy Truyen, Koen Handekyn, and Wouter Joosen. Efficient customization of multi-tenant software-as-a-service applications with service lines. *Journal of Systems and Software*, 91:48–62, 2014.

[90] Pooyan Jamshidi and Claus Pahl. Orthogonal variability modeling to support multi-cloud application configuration. In *European Conference on Service-Oriented and Cloud Computing*, pages 249–261. Springer, 2014.

[91] Andreas Metzger, Andreas Bayer, Daniel Doyle, Amir Molzam Sharifloo, Klaus Pohl, and Florian Wessling. Coordinated run-time adaptation of variability-intensive systems: An application in cloud computing. In *Proceedings of the 1st International Workshop on Variability and Complexity in Software Design*, VACE '16, pages 5–11, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4176-9. doi: 10.1145/2897045.2897049. URL http://doi.acm.org/10.1145/2897045.2897049.

[92] Tam Le Nhan, Gerson Sunyé, and Jean-Marc Jézéquel. A model-based approach for optimizing power consumption of iaas. In *Network Cloud Computing and Applications (NCCA), 2012 Second Symposium on*, pages 31–39. IEEE, 2012.

[93] Tam Le Nhan, Gerson Sunyé, and Jean-Marc Jézéquel. A model-driven approach for virtual machine image provisioning in cloud computing. *ESOCC*, 7592:107–121, 2012.

[94] Jesús García-Galán, Omer Rana, Pablo Trinidad Martín Arroyo, and Antonio Ruiz Cortés. Migrating to the cloud: a software product line based analysis. In *3Rd International Conference on Cloud Computing and Services Science, CLOSER 2013, Aachen, Alemania*, 2013.

[95] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 1st edition, 2001. ISBN 0201703327, 9780201703320.

[96] B Joseph Pine. *Mass customization: the new frontier in business competition*. Harvard Business Press, 1993.

[97] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, page 7. ACM, 2013.

[98] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[99] 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, DTIC Document, 1990.

[100] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 311–320. IEEE, 2008.

[101] Carlos Parra, Anthony Cleve, Xavier Blanc, and Laurence Duchien. Feature-based Composition of Software Architectures. In Muhammad Ali Babar and Ian Gorton, editors, *4th European Conference on Software Architecture*, volume 6285 of *Lecture Notes in Computer Science*, pages 230–245, Copenhagen, Denmark, August 2010. doi: 10.1007/978-3-642-15114-9_18. URL https://hal.inria.fr/inria-00512716.

[102] 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, 2014.

[103] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. Feature-model interfaces: the highway to compositional analyses of highly-configurable systems. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 667–678. IEEE, 2016.

[104] Jesús García-Galán, José María García, Pablo Trinidad, and Pablo Fernández. Modelling and analysing highly-configurable services. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*, pages 114–122. ACM, 2017.

[105] H. Hartmann and T. Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *2008 12th International Software Product Line Conference*, pages 12–21, Sept 2008. doi: 10.1109/SPLC.2008.15.

[106] R. Capilla, Ó Ortiz, and M. Hinchey. Context variability for context-aware systems. *Computer*, 47(2):85–87, Feb 2014. ISSN 0018-9162. doi: 10.1109/MC.2014.33.

[107] Kim Mens, Rafael Capilla, Nicolás Cardozo, and Bruno Dumas. A taxonomy of context-aware software variability approaches. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 119–124, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4033-5. doi: 10.1145/2892664.2892684. URL http://doi.acm.org/10.1145/2892664.2892684.

[108] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries. In *Generic Programming*, pages 25–39. Springer, 2000.

[109] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. 9th Int. Conf. Software Product Lines (SPLC'05)*, pages 7–20, Rennes, France, September 2005. ISBN 978-3-540-32064-7. doi: 10.1007/11554844_3.

[110] Yves Bontemps, Patrick Heymans, Pierre-Yves Schobbens, and Jean-Christophe Trigaux. Semantics of foda feature diagrams. In *Proceedings SPLC 2004 Workshop*

on Software Variability Management for Product Derivation–Towards Tool Support, pages 48–58, 2004.

[111] Jing Sun, Hongyu Zhang, Yuan Fang, and Li Hai Wang. Formal semantics and verification for feature modeling. In *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, pages 303–312. IEEE, 2005.

[112] Shaofeng Fan and Naixiao Zhang. Feature model based on description logics. In *Knowledge-Based Intelligent Information and Engineering Systems*, pages 1144–1151. Springer, 2006.

[113] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2): 456–479, 2007.

[114] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615 – 636, 2010. ISSN 0306-4379. doi: http://dx.doi.org/10.1016/j.is.2010.01.001. URL http://www.sciencedirect.com/science/article/pii/S0306437910000025.

[115] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 231–240, Pittsburgh, PA, USA, 2009. Carnegie Mellon University. URL http://dl.acm.org/citation.cfm?id=1753235.1753267.

[116] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. Sat-based analysis of large real-world feature models is easy. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, pages 91–100, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3613-0. doi: 10.1145/2791060.2791070. URL http://doi.acm.org/10.1145/2791060.2791070.

[117] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with UML multiplicities. In *Proc. 6th World Conference Integrated Design & Process Technology*, 2002.

[118] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W. Eisenecker. Generative programming for embedded software: An industrial experience report. In *Proc. 1st ACM SIGPLAN/SIGSOFT Conf. Generative Programming and Component Engineering*, pages 156–172, October 2002.

[119] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In *Proc. 3rd Software Product Line Conference*, pages 266–283, 2004.

[120] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[121] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints: A progress report. In *Proc. Int. Workshop Software Factories*, pages 16–20, 2005.

[122] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. Using java csp solvers in the automated analyses of feature models. *Lecture notes in computer science*, 4143:399, 2006.

[123] Jose Ramón Salazar Ramírez, Pablo Sánchez Barreiro, and Lidia Fuentes Fernández. Herramienta para el modelado y configuración de modelos de características. *Málaga*, 2009.

[124] Nadia Gamez and Lidia Fuentes. Software product line evolution with cardinality-based feature models. In *Proc. 12th Int. Conf. Software Reuse*, pages 102–118, 2011.

[125] Abel Gómez and Isidro Ramos. Automatic tool support for cardinality-based feature modeling with model constraints for information systems development. In *Information Systems Development*, pages 271–284. Springer, 2011.

[126] Raúl Mazo, Camille Salinesi, Daniel Diaz, and Alberto Lora-Michiels. Transforming attribute and clone-enabled feature models into constraint programs over finite domains. In *Proc. 6th Int. Conf. Evaluation of Novel Approaches to Software Engineering*, June 2011. URL https://hal.archives-ouvertes.fr/hal-00707546.

[127] Clément Quinton, Daniel Romero, and Laurence Duchien. Cardinality-based feature models with constraints: A pragmatic approach. In *Proc. 17th Int. Software Product Line Conference*, pages 162–166, 2013.

[128] Ahmet Serkan Karataş, Halit Oğuztüzün, and Ali Doğru. From extended feature models to constraint logic programming. *Science of Computer Programming*, 78(12): 2295 – 2312, 2013. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2012.06.004. URL http://www.sciencedirect.com/science/article/pii/S0167642312001153.

[129] Ahmet Serkan Karataş and Halit Oğuztüzün. Attribute-based variability in feature models. *Requirements Engineering*, 21(2):185–208, 2016.

[130] D. Dhungana, A. Falkner, and A. Haselbock. Configuration of cardinality-based feature models using generative constraint satisfaction. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 100–103, Aug 2011. doi: 10.1109/SEAA.2011.24.

[131] Thomas Schnabel, Markus Weckesser, Roland Kluge, Malte Lochau, and Andy Schürr. Cardygan: Tool support for cardinality-based feature models. In *Proc. 10th Int. Workshop Variability Modelling Software-Intensive Systems*, pages 33–40, 2016.

[132] Markus Weckesser, Malte Lochau, Thomas Schnabel, Björn Richerzhagen, and Andy Schürr. Mind the gap! automated anomaly detection for potentially unbounded cardinality-based feature models. In *International Conference on Fundamental Approaches to Software Engineering*, pages 158–175. Springer, 2016.

[133] Raphael Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. A formal semantics for feature cardinalities in feature diagrams. In *Proc. 5th Workshop Variability Modeling Software-Intensive Systems*, pages 82–89, 2011. ISBN 978-1-4503-0570-9.

[134] Mike Hinchey, Sooyong Park, and Klaus Schmid. Building dynamic software product lines. *Computer*, 45(10):22–26, 2012. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2012.332.

[135] Nelly Bencomo, Gordon S Blair, Carlos A Flores-Cortés, and Peter Sawyer. Reflective component-based technologies to support dynamic variability. In *VaMoS*, pages 141–150, 2008.

[136] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Models@run.time to support dynamic adaptation. *Computer*, 42(10):44–51, Oct 2009. ISSN 0018-9162. doi: 10.1109/MC.2009.327.

[137] Nelly Bencomo, Svein Hallsteinsen, and Eduardo Santana De Almeida. A view of the dynamic software product line landscape. *Computer*, 45(10):36–41, 2012.

[138] J. Bosch and R. Capilla. Dynamic variability in software-intensive embedded system families. *Computer*, 45(10):28–35, Oct 2012. ISSN 0018-9162. doi: 10.1109/MC.2012.287.

[139] Luciano Baresi and Clément Quinton. Dynamically evolving the structural variability of dynamic software product lines. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2015 IEEE/ACM 10th International Symposium on*, pages 57–63. IEEE, 2015.

[140] Karsten Saller, Malte Lochau, and Ingo Reimund. Context-aware DSPLs: Model-based Runtime Adaptation for Resource-constrained Systems. In *Proc. 17th Int. Software Product Line Conf. (SPLC'13) Co-located Workshops*, pages 106–113, Tokyo, Japan, August 2013. ISBN 978-1-4503-2325-3. doi: 10.1145/2499777.2500716.

[141] Aitor Murguzur, Rafael Capilla, Salvador Trujillo, Óscar Ortiz, and Roberto E. Lopez-Herrejon. Context variability modeling for runtime configuration of service-based dynamic software product lines. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*, SPLC '14, pages 2–9, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2739-8. doi: 10.1145/2647908.2655957. URL http://doi.acm.org/10.1145/2647908.2655957.

[142] Rafael Capilla, Mike Hinchey, and Francisco J. Díaz. Collaborative Context Features for Critical Systems. In *Proc. 9th Int. Workshop Variability Modelling of Software-intensive Systems (VaMoS'15)*, pages 43–50, Hildesheim, Germany, January 2015. ISBN 978-1-4503-3273-6. doi: 10.1145/2701319.2701322.

[143] Nelly Bencomo, Peter Sawyer, Gordon S Blair, and Paul Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *The 2nd International Workshop on Dynamic Software Product Lines (DSPL 2008), with SPLC 2008*, 2008.

[144] Nadia Gamez, Lidia Fuentes, and Miguel A. Aragüez. *Autonomic Computing Driven by Feature Models and Architecture in FamiWare*, pages 164–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-23798-0. doi: 10.1007/978-3-642-23798-0_16. URL https://doi.org/10.1007/978-3-642-23798-0_16.

[145] Luciano Baresi, Sam Guinea, and Liliana Pasquale. Service-oriented dynamic software product lines. *Computer*, 45(10):42–48, 2012.

[146] Amanda S Nascimento, Cecilia MF Rubira, and Fernando Castor. Using cvl to support self-adaptation of fault-tolerant service compositions. In *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*, pages 261–262. IEEE, 2013.

[147] Carlo Ghezzi and Amir Molzam Sharifloo. Dealing with non-functional requirements for adaptive systems via dynamic software product-lines. In *Software Engineering for Self-Adaptive Systems II*, pages 191–213. Springer, 2013.

[148] Indika Kumara, Jun Han, Alan Colman, and Malinda Kapuruge. Runtime evolution of service-based multi-tenant saas applications. In *International Conference on Service-Oriented Computing*, pages 192–206. Springer, 2013.

[149] Davide Brugali, Rafael Capilla, and Mike Hinchey. Dynamic variability meets robotics. *Computer*, 48(12):94–97, 2015.

[150] Aitor Murguzur, Salvador Trujillo, Hong-Linh Truong, Schahram Dustdar, Oscar Ortiz, and Goiuria Sagardui. Runtime variability for context-aware smart workflows. *IEEE Software*, 2015.

[151] Martin Pfannemüller, Christian Krupitzer, Markus Weckesser, and Christian Becker. A dynamic software product line approach for adaptation planning in autonomic computing systems. In *Autonomic Computing (ICAC), 2017 IEEE International Conference on*, pages 247–254. IEEE, 2017.

[152] Nelly Bencomo, Paul Grace, Carlos Flores, Danny Hughes, and Gordon Blair. Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems. In *Proc. 30th Int. Conf. Software Engineering (ICSE'08)*, pages 811–814, Leipzig, Germany, May 2008. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368207.

[153] Pete Sawyer, Raul Mazo, Daniel Diaz, Camille Salinesi, and Danny Hughes. Using constraint programming to manage configurations in self-adaptive systems. *Computer*, 45(10):56–63, 2012.

[154] Mathieu Acher, Philippe Collet, Franck Fleurey, Philippe Lahire, Sabine Moisan, and Jean-Paul Rigault. Modeling context and dynamic adaptations with feature models. In *4th International Workshop Models@ run. time at Models 2009 (MRT'09)*, page 10, 2009.

[155] Rafael Capilla and Jan Bosch. Binding Time and Evolution. In Rafael Capilla, Jan Bosch, and Kyo-Chul Kang, editors, *Systems and Software Variability Management: Concepts, Tools and Experiences*, pages 57–73. Springer Berlin Heidelberg, Berlin, Germany, 2013. ISBN 978-3-642-36583-6. doi: 10.1007/978-3-642-36583-6_4.

[156] Vander Alves, Daniel Schneider, Martin Becker, Nelly Bencomo, and Paul Grace. Comparative study of variability management in software product lines and runtime adaptable systems. In *Third International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 9–17, January 2009.

[157] Mahdi Bashari, Ebrahim Bagheri, and Weichang Du. Dynamic software product line engineering: a reference framework. *International Journal of Software Engineering and Knowledge Engineering*, 27(02):191–234, 2017.

[158] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger M Kienle, Marin Litoiu, Hausi A Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. *Software engineering for self-adaptive systems*, 5525:48–70, 2009.

[159] A. Safilian and T. Maibaum. Hierarchical multiset theories of cardinality-based feature diagrams. In *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 136–143, July 2016. doi: 10.1109/TASE.2016.14.

[160] Aliakbar Safilian, Tom Maibaum, and Zinovy Diskin. The semantics of cardinality-based feature models via formal languages. In *Proc. 20th Int. Symp. Formal Methods*, pages 453–469, 2015.

[161] Red Hat OpenShift. OpenShift Online User Guide. https://access.redhat.com/documentation/en-US/OpenShift_Online/2.0/html/User_Guide/index.html, 2014.

[162] Wei Zhang, Hua Yan, Haiyan Zhao, and Zhi Jin. A BDD-based approach to verifying clone-enabled feature models' constraints and customization. In *Proc. 10th Int. Conf. Software Reuse*, pages 186–199, 2008.

[163] Clément Quinton, Andreas Pleuss, Daniel Le Berre, Laurence Duchien, and Goetz Botterweck. Consistency checking for the evolution of cardinality-based feature models. In *Proc. 18th Int. Software Product Line Conference*, pages 122–131, 2014.

[164] Don Batory, David Benavides, and Antonio Ruiz-Cortes. Automated analysis of feature models: Challenges ahead. *Commun. ACM*, 49(12):45–47, December 2006. ISSN 0001-0782. doi: 10.1145/1183236.1183264. URL http://doi.acm.org/10.1145/1183236.1183264.

[165] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013. ISBN 1782160302, 9781782160304.

[166] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[167] OMG OMG. Object constraint language (OCL), version 2.3.1. http://www.omg.org/spec/OCL/2.3.1/, 2012.

[168] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2015. URL http://www.choco-solver.org.

[169] Markus Stumptner, Gerhard E Friedrich, and Alois Haselböck. Generative constraint-based configuration of large technical systems. *AI EDAM*, 12(4):307–320, 1998.

[170] C. Salinesi, R. Mazo, O. Djebbi, D. Diaz, and A. Lora-Michiels. Constraints: The core of product line engineering. In *2011 FIFTH INTERNATIONAL CONFER-*

ENCE ON RESEARCH CHALLENGES IN INFORMATION SCIENCE, pages 1–10, May 2011. doi: 10.1109/RCIS.2011.6006825.

[171] Arnaud Hubaux and Patrick Heymans. On the evaluation and improvement of feature-based configuration techniques in software product lines. In *Proc. 31st Int. Conf. Software Engineering (ICSE'09)*, pages 367–370, Vancouver, Canada, May 2009. doi: 10.1109/ICSE-COMPANION.2009.5071023.

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

[173] Upgrading your ClearDB database. https://web.archive.org/web/20161117190535/https://devcenter.heroku.com/articles/cleardb#upgrading-your-cleardb-database, 2016.

[174] Upgrading heroku postgres databases. https://web.archive.org/web/20161124234722/https://devcenter.heroku.com/articles/upgrading-heroku-postgres-databases, 2016.

[175] Setting a buildpack on an application. https://web.archive.org/web/20170121215709/https://devcenter.heroku.com/articles/buildpacks#setting-a-buildpack-on-an-application, 2016.

[176] Scaling the dyno size. https://web.archive.org/web/20161106120713/https://devcenter.heroku.com/articles/scaling#scaling-the-dyno-size, 2016.

[177] Migrating an application to another region. https://web.archive.org/web/20161024050822/https://devcenter.heroku.com/articles/app-migration, 2016.

[178] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. Context Aware Reconfiguration in Software Product Lines. In *Proc. 10th Int. Workshop Variability Modelling of Software-intensive Systems (VaMoS'16)*, pages 41–48, Salvador, Brazil, 2016. ISBN 978-1-4503-4019-9. doi: 10.1145/2866614.2866620.

[179] Jaejoon Lee and K. C. Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *Proc. 10th Int. Software Product Line Conf. (SPLC'06)*, pages 131–140, Baltimore, MD, USA, August 2006. doi: 10.1109/SPLINE.2006.1691585.

[180] Johannes Bürdek, Sascha Lity, Malte Lochau, Markus Berens, Ursula Goltz, and Andy Schürr. Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints. In *Proc. 8th Int. Workshop Variability Modelling of Software-intensive Systems (VaMoS'14)*, Sophia Antipolis, France, January 2013. ISBN 978-1-4503-2556-1. doi: 10.1145/2556624.2556627.

[181] Malte Lochau, Johannes Bürdek, Stefan Hölzle, and Andy Schürr. Specification and automated validation of staged reconfiguration processes for dynamic software product lines. *Software & Systems Modeling*, 16(1):125–152, Feb 2017. ISSN 1619-1374. doi: 10.1007/s10270-015-0470-4. URL https://doi.org/10.1007/s10270-015-0470-4.

[182] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, 42 (10):37–43, Oct 2009. ISSN 0018-9162. doi: 10.1109/MC.2009.309.

[183] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN 026202649X, 9780262026499.

[184] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer Science & Business Media, 2012.

[185] Zohar Manna and Amir Pnueli. A Hierarchy of Temporal Properties (Invited Paper, 1989). In *Proc. 9th Annu. ACM Symp. Principles of Distributed Computing (PODC'90)*, pages 377–410, Quebec City, Canada, August 1990. ISBN 0-89791-404-X. doi: 10.1145/93385.93442.

[186] Maurice H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An Action/State-Based Model-Checking Approach for the Analysis of Communication Protocols for Service-Oriented Applications. In *Proc. 12th Int. Workshop Formal Methods for Industrial Critical Systems (FMICS'07)*, pages 133–148, Berlin, Germany, July 2008. ISBN 978-3-540-79707-4. doi: 10.1007/978-3-540-79707-4_11.

[187] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. State/Event-Based Software Model Checking. In *Proc. 4th Int. Conf. Integrated Formal Methods (IFM'04)*, pages 128–147, Canterbury, UK, April 2004. ISBN 978-3-540-24756-2. doi: 10.1007/978-3-540-24756-2_8.

[188] Charles Pecheur and Franco Raimondi. Symbolic Model Checking of Logics with Actions. In *Proc. 4th Workshop Model Checking and Artificial Intelligence (MoChArt)*, pages 113–128, Riva del Garda, Italy, August 2006. ISBN 978-3-540-74128-2. doi: 10.1007/978-3-540-74128-2_8.

[189] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 23–34. IEEE, 2007.

[190] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10(1):47–71, 1997. ISSN 1572-8102. doi: 10.1023/A:1008615614281.

[191] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Roberto Sebastiani. Improving the Encoding of LTL Model Checking into SAT. In *Proc. 3rd Int. Workshop Model Checking and Abstract Interpretation (VMCAI'02)*, pages 196–207, Venice, Italy, January 2002. ISBN 978-3-540-47813-3. doi: 10.1007/3-540-47813-2_14.

[192] Z. Conghua and J. Shiguang. Sat-based bounded model checking for se-ltl. In *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, volume 3, pages 582–587, July 2007. doi: 10.1109/SNPD.2007.242.

[193] Nazareno Aguirre and Tom Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 271–274. IEEE, 2002.

[194] OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). https://www.w3.org/TR/owl2-syntax/, 2012.

[195] Thomas R Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, 43(5-6):907–928, 1995.

[196] Christina Feilmayr and Wolfram Wöß. An analysis of ontologies and their success factors for application to business. *Data & Knowledge Engineering*, 101:1–23, 2016.

[197] Dean Allemang and James Hendler. *Semantic web for the working ontologist: effective modeling in RDFS and OWL*. Morgan Kaufmann, 2014.

[198] OWL 2 Web Ontology Language Manchester Syntax (Second Edition). https://www.w3.org/TR/owl2-manchester-syntax/, 2012.

[199] OWL 2 Web Ontology Language Document Overview (Second Edition)). https://www.w3.org/TR/owl2-overview/, 2012.

[200] Steffen Staab, Tobias Walter, Gerd Gröner, and Fernando Silva Parreiras. Model driven engineering with ontology technologies. *Reasoning Web*, 6325:62–98, 2010.

[201] TwoUse Toolkit: Semantic Web + Model Driven Development. https://west.uni-koblenz.de/en/research/software/twouse, 2017.

[202] Boris Motik, Rob Shearer, and Ian Horrocks. Hypertableau Reasoning for Description Logics. *Journal of Artificial Intelligence Research*, 36:165–228, 2009.

[203] Sock shop : A microservice demo application. https://github.com/microservices-demo/microservices-demo, 2017.

[204] Sock shop : A microservice demo application - design. https://github.com/microservices-demo/microservices-demo/blob/master/internal-docs/design.md, 2017.

[205] Clément Quinton. *Cloud Environment Selection and Configuration: A Software Product Lines-Based Approach*. Theses, Université Lille 1, October 2014. URL https://tel.archives-ouvertes.fr/tel-01079956.