Feature-Based Configuration Management of Reconfigurable Cloud Applications

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Medieninf. Julia Schroeter
geboren am 01. Juli 1981 in Bautzen

Betreuender Hochschullehrer
Prof. Dr. rer. nat. habil. Uwe Aßmann
(Technische Universität Dresden)

Externer Gutachter
Prof. Dr. Vander Alves
(Universidade de Brasília, Brasilien)

Tag der Verteidigung
11. April 2014

Dresden im Dezember 2013

b

Abstract

A recent trend in software industry is to provide enterprise applications in the cloud that are accessible everywhere and on any device. As the market is highly competitive, customer orientation plays an important role. Companies therefore start providing applications as a service, which are directly configurable by customers in an online *self-service portal*. However, customer configurations are usually deployed in separated application instances. Thus, each instance is provisioned manually and must be maintained separately. Due to the induced redundancy in software and hardware components, resources are not optimally utilized. A *multi-tenant* aware application architecture eliminates redundancy, as a single application instance serves multiple customers renting the application. The combination of a configuration self-service portal with a multi-tenant aware application architecture allows serving customers just-in-time by automating the deployment process. Furthermore, self-service portals improve application scalability in terms of functionality, as customers can adapt application configurations on themselves according to their changing demands. However, the configurability of current multi-tenant aware applications is rather limited. Solutions implementing variability are mainly developed for a single business case and cannot be directly transferred to other application scenarios.

The goal of this thesis is to provide a generic framework for handling application variability, automating configuration and reconfiguration processes essential for self-service portals, while exploiting the advantages of multi-tenancy. A promising solution to achieve this goal is the application of *software product line* methods. In software product line research, *feature models* are in wide use to express variability of software intense systems on an abstract level, as features are a common notion in software engineering and prominent in matching customer requirements against product functionality.

This thesis introduces a framework for feature-based configuration management of reconfigurable cloud applications. The contribution is three-fold. First, a development strategy for flexible multi-tenant aware applications is proposed, capable of integrating customer configurations at application runtime. Second, a generic method for defining concern-specific configuration perspectives is contributed. Perspectives can be tailored for certain application scopes and facilitate the handling of numerous configuration options. Third, a novel method is proposed to model and automate structured configuration processes that adapt to varying stakeholders and reduce configuration redundancies. Therefore, configuration processes are modeled as workflows and adapted by applying rewrite rules triggered by stakeholder events. The applicability of the proposed concepts is evaluated in different case studies in the industrial and academic context.

Summarizing, the introduced framework for feature-based configuration management is a foundation for automating configuration and reconfiguration processes of multi-tenant aware cloud applications, while enabling application scalability in terms of functionality.

d

Acknowledgements

First of all, I like to thank my supervisor, Uwe Aßmann, for his invaluable feedback, encouragement and support of my thesis over the past years.

I also thank my colleagues in the Software Technology Group at the Technische Universität Dresden, who have taken a great part of this Ph.D. journey with me. Special thanks to Christoff Bürger, Sebastian Götz, Sven Karol, Georg Püschel, Jan Reimann, and Christoph Seidl for reviewing chapters of my thesis and providing valuable feedback. I thank Katrin Heber, who took many time-consuming administrative tasks off my hands.

Additionally, I give thanks to my managers at SAP, Ralf Ackermann, Michael Ameling, Steffen Göbel, and especially Uwe Kubach, without which this collaborative project could not have been established. I thank my colleagues at SAP lab for many inspiring conversations, especially in the INDENICA project context. In particular, thank you, Andreas Rummler for regularly discussing my research topic.

Furthermore, I like to thank the fellow researchers, I met at conferences and in workshops for their valuable feedback and inspiring talks on my and related research topics. Special thanks to Vander Alves for the good and intense discussions. I thank the people I collaborated with during the last years, especially Malte Lochau for the fruitful cooperation and manifold support. In addition, thank you, Stephan Mennicke, Dirk Muthig, and Karsten Saller.

In addition, I thank the students I supervised, Mabel Joselin Brun Chaperon, David Gollasch, Xi Luo, and Tim Winkelmann, who contributed to this work by means of their final theses. I like to thank the teams of Misys IBS and Achievo, were I learned how to apply model-driven software development in practice.

Special thanks to my friends, who supported me during the last years by spending quality time, and my family for their advice and support of my academic education. Most of all, I thank my husband Olaf for being endlessly patient, loving and kind.

I further acknowledge the financial support of my doctoral project #080949335 provided by the European Social Fund, Federal State of Saxony and SAP.

Europäische Union

Europa fördert Sachsen.
ESF
Europäischer Sozialfonds

SACHSEN

f

Publications

This doctoral thesis is based on the following peer-reviewed publications ordered by date, starting with the most recent.

[1] Dirk Muthig and **Julia Schroeter**. A Framework for Role-Based Feature Management in Software Product Line Organizations. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13 (1), New York, NY, USA, August 2013. ACM Press.

[2] **Julia Schroeter**, Malte Lochau, and Tim Winkelmann. Multi-Perspectives on Feature Models. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages & Systems*, MODELS '12, pages 252–268. Springer Berlin Heidelberg, October 2012.

[3] **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*, SPLC '12 (2), pages 171–178, New York, NY, USA, September 2012. ACM Press.

[4] **Julia Schroeter**, Malte Lochau, and Tim Winkelmann. Conper: Consistent Perspectives on Feature Models. In *Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications*, ECMFA '12, pages 55–58. Technical University of Denmark DTU, July 2012.

[5] **Julia Schroeter**, Sebastian Cech, Sebastian Götz, Claas Wilke, and Uwe Aßmann. Towards Modeling a Variable Architecture for Multi-Tenant SaaS-Applications. In *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 111–120, New York, NY, USA, January 2012. ACM Press.

[6] **Julia Schroeter**. Towards Generating Multi-Tenant Applications. In *Pre-Proceedings of the 4th Summer School on Generative and Transformational Techniques in Software Engineering and 4th Software Language Engineering 2011 – Students' Workshop*, GTTSE/SLE '11, July 2011.

Further publications related to the thesis.

[7] Malte Lochau, Stephan Mennicke, **Julia Schroeter**, and Tim Winkelmann. Extended Version of Automated Verification of Feature Model Configuration Processes Based on Workflow Petri Nets. Technical report, Technische Universität Braunschweig, January 2013.

[8] **Julia Schroeter**, Malte Lochau, and Tim Winkelmann. Extended Version of Multi-Perspectives on Feature Models. Technical report, Technische Universität Dresden, December 2011.

Further peer-reviewed publications indirectly related to this thesis.

[9] Uwe Aßmann, Andreas Bartho, Christoff Bürger, Sebastian Cech, Birgit Demuth, Florian Heidenreich, Jendrik Johannes, Sven Karol, Jan Polowinski, Jan Reimann, **Julia Schroeter**, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke. DropsBox: The Dresden Open Software Toolbox. - Domain-Specific Modelling Tools beyond Metamodels and Transformations. *Journal of Software and Systems Modeling (SoSyM)*, pages 1–37, November 2012.

[10] Karsten Saller, Sebastian Oster, Andy Schürr, **Julia Schroeter**, and Malte Lochau. Reducing Feature Models to Improve Runtime Adaptivity on Resource Limited Devices. In *Proceedings of the 16th International Software Product Line Conference*, SPLC '12 (2), pages 135–142, New York, NY, USA, September 2012. ACM Press.

[11] Claas Wilke, Andreas Bartho, **Julia Schroeter**, Sven Karol, and Uwe Aßmann. Elucidative Development for Model-Based Documentation. In *Proceeding of the 50th International Conference on Objects, Models, Components, Patterns*, volume 7304 of *TOOLS '12*, pages 320–335. Springer Berlin Heidelberg, June 2012.

[12] Sebastian Götz, Max Leuthäuser, Christian Piechnick, Jan Reimann, Sebastian Richly, **Julia Schroeter**, Class Wilke, and Uwe Aßmann. Entwicklung Cyber-Physikalischer Systeme am Beispiel des NAO-Roboters. In *Proceedings of Chemnitz Linux-Days 2012*. Universitätsverlag Chemnitz, March 2012.

[13] Sebastian Götz, Max Leuthäuser, Jan Reimann, **Julia Schroeter**, Christian Wende, Claas Wilke, and Uwe Aßmann. NaoText: A Role-Based Language for Collaborative Robot Applications. In *Proceedings of the 1st International Workshop on Software Aspects of Robotic Systems*, ISoLA '11, October 2011.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

ANSI	American National Standards Institute	26
ANTLR	Another Tool for Language Recognition	183
ASP	Application Service Provider	11
AWS	Amazon Web Services	27
BDD	Binary Decision Diagram	46
BPEL	Business Process Execution Language	68
BPMN	Business Process Model and Notation	45
CCM	Cool Component Model	79
CSOP	Constraint Satisfaction Optimization Problem	81
CSP	Constraint Satisfaction Problem	46
CRM	Customer Relationship Management	11
CVL	Common Variability Language	34
CVM	Compositional Variability Management	194
DSL	Domain-Specific Language	180
DSPL	Dynamic Software Product Line	32
EBNF	Extended Backus-Naur Form	183
EIA	Electronic Industries Alliance	50
EMF	Eclipse Modeling Framework	180
EMOF	Essential Meta Object Facility	181
FAMILIAR	Feature Model Script Language for Manipulation and Automatic Reasoning	193
FDL	Feature Description Language	193
FODA	Feature Oriented Domain Analysis	35
GPL	General Purpose Language	180
HUTN	Human-Usable Textual Notation	183
IaaS	Infrastructure as a Service	11
IDC	International Data Corporation	1
IDE	Integrated Development Environment	179
INCITS	International Committee for Information Technology Standards	26
IT	Information Technology	9
JWT	Java Workflow Tooling	215
MDA	Model-Driven Architecture	181
MDSD	Model-Driven Software Development	35
MIT	Massachusetts Institute of Technology	10
MQuAT	Multi-Quality Auto-Tuning	77
MOF	Meta Object Facility	181
NIST	National Institute of Standards And Technology	10
OCL	Object Constraint Language	39
OCR	Object Character Recognition	95
OMG	Object Management Group	181

OVM	Orthogonal Variability Modeling	34
PaaS	Platform as a Service	11
PDF	Portable Document Format	95
PLA	Product Line Architecture	73
Prolog	Programmation en Logique	46
PUMA	Product Line Utilities for Multi-Tenant Aware Applications	6
QCL	Quality Contract Language	79
QoS	Quality of Service	18
RBAC	Role Based Access Control	6
SaaS	Software As A Service	11
SAT	Satisfiability	46
SCArVeS	International Workshop on Services, Clouds and Alternative Design Strategies for Variant-Rich Software Systems	3
SCM	Software Configuration Management	51
SOA	Service-Oriented Architecture	199
SoC	Separation of Concerns	77
SLA	Service Level Agreement	17
SMT	Satisfiability Modulo Theories	46
SPL	Software Product Line	3
SPLC	International Software Product Line Conference	31
SPLOT	Software Product Line Online Tools	202
THEATRE	The Auto-Tuning Runtime Environment	79
TVL	Textual Variability Language	194
UI	User Interface	179
UML	Unified Modeling Language	38
VaMoS	International Workshop on Variability Modelling of Software-Intensive Systems	71
VSL	Variability Specification Language	194
XMI	XML Metadata Interchange	183
XML	Extensible Markup Language	183

The Need for Reconfigurable Cloud Applications and Automated Configuration Management

The great thing is the start – to see an opportunity for service, and to start doing it, even though in the beginning you serve but a single customer – and him for nothing.

— *John Collier*

More and more enterprise applications are provided as services in the cloud, as there is a strong demand for accessing applications everywhere on heterogenous devices. In a recent report, the International Data Corporation (IDC) expected for the next years that applications in the cloud are a booming market [GAB+13]. The IDC market analysis predicts that much more customers use cloud services in the next years, and double their expenses from \$47,4 billions in 2013 up to \$107 billions in 2017. Especially for small and medium-sized companies the trend towards renting services in the cloud is beneficial, as renting a service is cheaper than running an own data center.

However, many enterprise solutions offered by large software vendors are too expensive, and the range of functions of those solutions is too wide for small and medium-sized companies. Thus, providers of enterprise applications address these demands by offering customizable applications. These applications must be highly configurable to meet diverse customer requirements. An example for a configurable cloud application is *SAP Business ByDesign*[1], an application for enterprise resource planning and customer relationship management. This application provides a self-service portal for customers to configure the application and receive a corresponding cost estimate. In the portal, customers choose among diverse functionality by explicitly selecting and deselected application features. For instance, a customer can choose from different marketing and sales functionality and defines a number of employees that will use the application. According to the configuration the estimated costs per month are calculated and presented to the customer. Eventually, a request is sent to the provider for instantiation the application accordingly. The request is processed manually, where an application instance is deployed per requesting customer.

However, the demands of the customers change. Therefore, a cloud application must be scalable in terms of functionality to address the changing requirements of customers. A challenge is to offer self-service portals where customers configure and reconfigure their application subscription without further provider interaction, while their configuration becomes instantly available. Self-service portals, such as the *solution configurator for Business ByDesign*[2], enable customers to

[1]`http://www.sapbydesign.com/`
[2]`https://www.sapconfigurator.com`

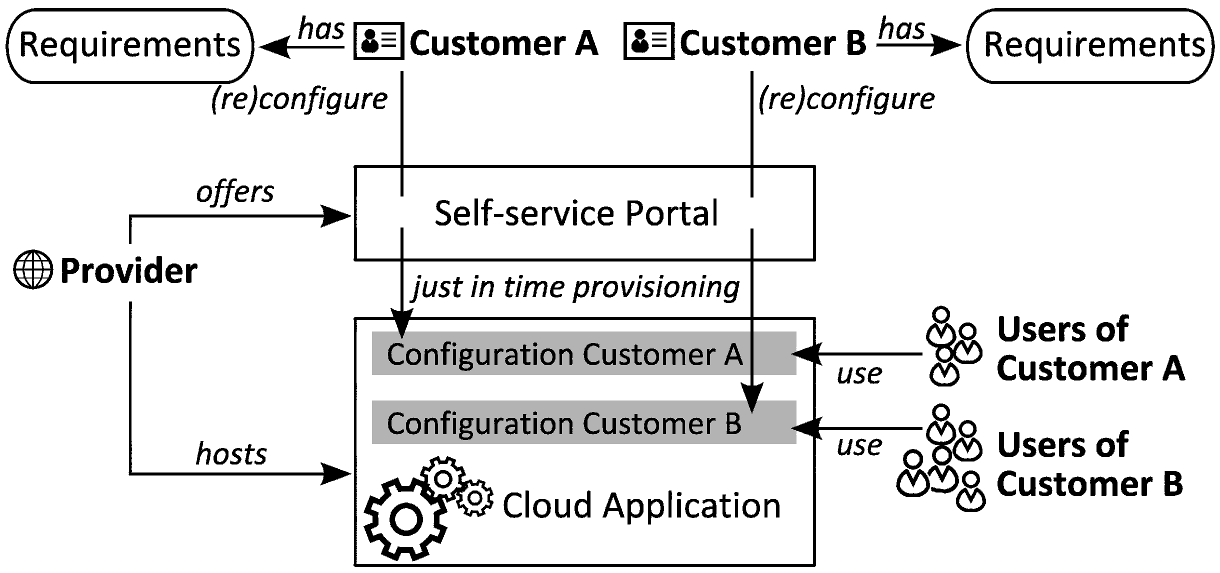
configure applications on their own. However, a derived configuration is sent to a support team of the application provider for instantiating an application manually, which delays application provisioning and produces extra costs.

A study conducted by Hurwitz & Associates reports that manual configuration errors impact the productivity of developers and other people involved in application provisioning, and causes a significant delay in rolling out the application [Hur10]. The study confirms that application downtime is often caused by configuration errors. In addition, the study reveals that downtime of web applications costs companies up to $72,000 per hour. A recent Gartner report confirmed that most of the service outages are caused by configuration and change errors introduced by people and processes [CS10].

> Through 2015, 80% of outages impacting mission-critical services will be caused by people and process issues, and more than 50% of those outages will be caused by change, configuration, release integration and hand-off issues.

As a consequence, configuration management and the provisioning process must be automated to prevent manual errors and to accelerate the instantiation of customer configurations. Furthermore, an application that is available *just-in-time* reduces the decision threshold of customers to try out an application. This further demands for an application architecture that supports the automatic integration of customer configurations.

A *multi-tenant* aware application and platform architecture supports automated provisioning, as multiple customers access a single shared application instance. The following figure exemplifies how to combine a self-service portal and a reconfigurable multi-tenant aware application. In this example, a provider offers a configurable application as a service together with a self-service



Configuration self-service portal and multi-tenant aware application architecture allow for just-in-time provisioning of customizations.

portal for tailoring the application. However, the requirements of customer A and customer B on the application functionality varies. The self-service portal enables customers to independently subscribe for, and unsubscribe from, a service and to tailor the application without further assistance of the provider. Both customers A and B subscribe for a customized version of the cloud application. Moreover, users of customer A access only the tailored application functionality defined by the configuration of customer A, where the same holds for users of customer B. However, current multi-tenant aware applications lack customizability.

Research Objectives

Two central questions are to be answered by this thesis. First, how to develop customizable cloud applications which scale in terms of functionality? Second, how to achieve just-in-time availability of tailored applications?

A structured analysis of existing cloud applications reveals a strong demand for scalable applications, which are configurable and reconfigurable in self-service portals, and provisioned just-in-time, as explained in Chapter 1. Based on these findings, challenges in automating application configuration processes, and the just-in-time provisioning of configured applications are identified. Subsequently, methods from Software Product Line (SPL) engineering are investigated for their applicability to meet the identified challenges. SPL methods are a well-established foundation to cope with variability as explained in Chapter 2. However, the methods are to be extended to explicitly address the challenges in scalable cloud applications.

Three main requirements in applying SPL methods are identified. First, a flexible application architecture supporting just-in-time provisioning of configured and reconfigured applications is required. Second, a self-service portal demands for a method for explicitly tailoring the configuration space prior the configuration process due to explicit concerns. Third, as not all stakeholders of a self-service portal are known beforehand, a structured and adaptive configuration process with reconfiguration support is required. These main requirements are explained in detail in Section 2.8.

The recent workshop International Workshop on Services, Clouds and Alternative Design Strategies for Variant-Rich Software Systems (SCArVeS)[3] reveals that SPL engineering is applicable to cope with variability in cloud applications. This thesis proposes an SPL-based configuration management framework for scalable and reconfigurable cloud applications. The conceptualized framework is implemented prototypically and evaluated empirically.

[3]`http://www.iese.fraunhofer.de/en/events/scarves2012.html`

Contribution

The focus of this work is on expressing variability, automating configuration and reconfiguration processes, while further important issues regarding application and data security, network communication, as well as persistence issues are out of scope of this work.

The contribution of the thesis is a framework for managing configuration and reconfiguration of cloud applications by applying SPL methods. The table on the bottom of this page summarizes the main contributions of this thesis and related publications, while the figure on the next page illustrates these concepts.

Contribution a) is a method for *developing flexible applications* supporting customer constraints in the application architecture. The method extends a self-adaptive application architecture with functional variability and multi-tenancy constraints on accessing functionality. A development strategy is presented in Chapter 4. In SPL engineering, functional variability comprising configuration parameters are uniformly specified on abstract level to define all derivable application configurations while abstracting from implementation details [PBvdL05].

Contribution b) is the concept of *multi-perspectives* for tailoring the configuration space explicitly according to a set of concerns. Views on the configuration parameters separate concerns, while perspectives aggregate multiple views to narrow the configuration space. Hence, perspectives allow for a concise definition of concern-specific pre-configurations. In addition, a perspective allows for expressing customized functionality on an abstract level. For instance, if a customer requests particular customer-specific functionality not available to other customers, the functionality is only visible in the perspective of the requesting customer. If further customers request the same functionality, the availability restriction can be revoked. In addition, explicit information about customized functionality is imported for application maintenance, as customizations are required to function properly after updating the application. Multi-perspectives are introduced and formalized in Chapter 5.

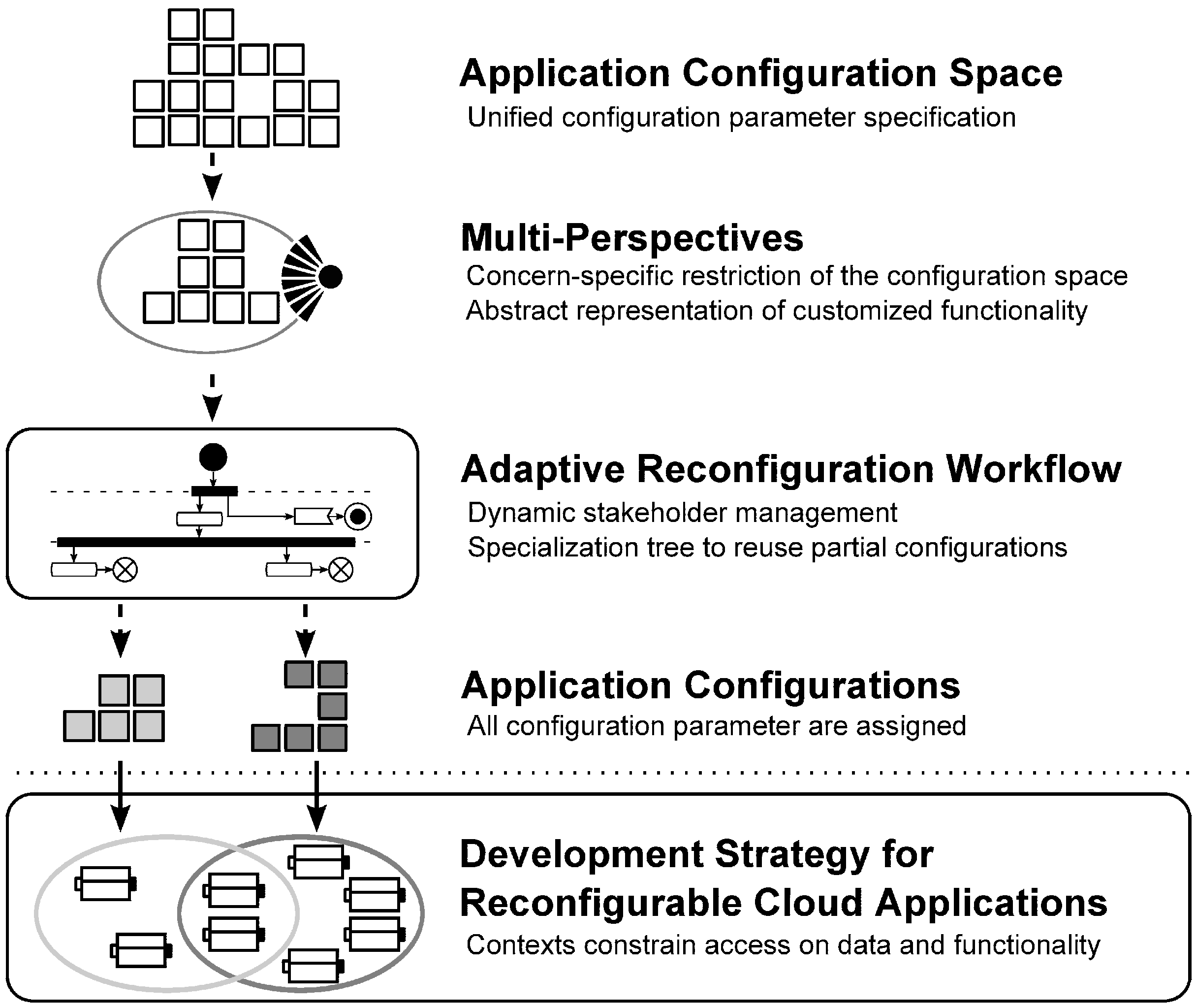
Contribution c) refers to the concept of *adaptive staged reconfiguration workflows* for automating structured, adaptive configuration, and reconfiguration processes. Adaptive staged reconfiguration workflows are presented in Chapter 6 as a special class of staged configuration workflows

Overview of the contributions of this thesis, addressed requirements and related publications.

	Contribution	Explanation	Publications
a)	Flexible application design	Chapter 4	[SCG+12]
b)	Multi-perspectives	Chapter 5	[SLW11, SLW12a, SLW12b]
c)	Adaptive staged configuration workflows	Chapter 6	[Sch11, SMM+12, LMSW13, MS13]
d)	Tool suite PUMA	Chapter 7	`https://github.com/extFM/extFM-Tooling`

introduced by Czarnecki [CHE04]. A staged configuration workflow allows to derive a single variant configuration in multiple steps, where various pre-defined stakeholders are involved in the configuration process. In contrast, in a staged reconfiguration workflow, multiple variant configurations can be derived at the same time, and the workflow can be adapted at runtime to integrate or remove stakeholders supporting dynamic stakeholder management. Hence, partial configurations of particular stakeholders are reused. Furthermore, the workflow supports reconfiguration of partial configurations. Various application configurations, where all configuration parameters are assigned, result from a single workflow. The derived variant configurations correspond to configuration contexts in the multi-tenant aware application instance.

Contribution d) refers to the tool suite PUMA, which is available open source and hosted in the repository `https://github.com/extFM/extFM-Tooling` at Github. PUMA comprises tools and languages to express configuration management related concepts, depicted in the figure above the dotted line and explained in Chapter 7. In particular, reference implementations for the concepts of contributions b) and c) are provided. The concepts of perspectives and an



Application Configuration Space
Unified configuration parameter specification
Multi-Perspectives
Concern-specific restriction of the configuration space
Abstract representation of customized functionality
Adaptive Reconfiguration Workflow
Dynamic stakeholder management
Specialization tree to reuse partial configurations
Application Configurations
All configuration parameter are assigned
Development Strategy for
Reconfigurable Cloud Applications
Contexts constrain access on data and functionality

Configuration management overview of reconfigurable cloud applications efficient consistency check algorithm are implemented in the tool *Conper* presented in Section 7.5. Adaptive staged configuration workflows are implemented in the tool *DyscoGraph* presented in Section 7.6. The concepts of multi-perspectives and adaptive staged configuration workflows are evaluated empirically on case studies in this chapter.

A detailed overview of the concepts of the configuration management framework proposed in this thesis is given in Chapter 3, and explained by example, while the Chapters 1 and 2 explain the background of cloud applications and SPL engineering.

Outline

The outline of this thesis is as follows. In Chapter 1, the state of the art in cloud applications and the context of cloud computing is explained. Subsequently, Chapter 2 explains concepts and methods of SPL engineering, that are related to reconfigurable cloud applications. In addition, open challenges are discussed in combining SPL engineering and cloud applications. Requirements for automating the configuration of cloud applications are identified as well.

Chapter 3 introduces the conceptual configuration management framework to automate the configuration of cloud applications based on SPL methods meeting the identified requirements. Chapter 4 proposes a development method for flexible cloud applications comprising multi-tenancy and a component-based self-adaptive application architecture. In addition, Chapter 5 introduces multi-perspectives and provides a formalization of views and consistent perspectives on feature models. Chapter 6 explains how to automate the configuration and reconfiguration of cloud applications by extending staged configuration concepts. In this chapter, adaptive staged reconfiguration workflows are introduced and explained by example. Adaptive staged reconfiguration workflows combine Role Based Access Control (RBAC), staged configuration concepts, and graph rewrite rules to automate the configuration of cloud applications while reducing redundancies of configuration decisions.

Furthermore, the tool suite Product Line Utilities for Multi-Tenant Aware Applications (PUMA) is introduced in Chapter 7. The tool suite comprises reference implementations for multi-perspectives and adaptive staged reconfiguration workflows developed in this work to evaluate these concepts for applicability. Finally, Chapter 8 summarizes this work and gives an outlook for future research regarding the SPL oriented development of cloud applications.

Part I.

Context and Preliminaries

1. State of the Art of Cloud Applications

Computing may someday be organized as a public utility just as the telephone system is a public utility. . . . The computer utility could become the basis of a new and important industry.

— John McCarthy, 1961

Cloud computing and cloud applications become more and more important in industry. A representative study commissioned by KPMG and Bitkom reports that 37% of German companies applied cloud computing in the year 2012, and further 29% plan and discuss its utilization [KPMG13]. This chapter provides an overview of cloud computing in general, while explaining the state of the art in cloud applications and identifying specific research challenges for reconfigurable cloud applications.

The terms *cloud* and *cloud computing* are often used nowadays with different meanings in mind. This chapter clarifies their meaning and explains characteristics of cloud applications combining desktop application functionality with the flexibility of a web application. Cloud-specific technologies and architectural paradigms are explained that build the basis for cloud applications. In addition, cloud technologies imply new business models by providing resources and applications as services on the Internet. In turn, business models and corresponding business concerns drive the design of cloud applications. For instance, a large group of customers is reached by providing configurable applications varying in function and price. Even more customers are attracted by providing scalable applications that scale fast in terms of functionality according to current customer demands. Hence, providing applications with scalable functionality demands for a reconfigurable application design and a concise variation management which is focus of this thesis.

1.1. Origin of Cloud Terminology

The term *cloud* originates from the long term use of a cloud-like shape as a metaphor for the Internet to abstract from network devices and connections in presentations and publications. Thus cloud is defined as follows.

Definition 1.1 (Cloud). *Cloud is a general term for providing on-demand services on the Internet while abstracting from particular computing resources and network connections.*

Actually, the idea of shared Information Technology (IT) services is not new. In 1961, John McCarthy described the idea of providing computing resources as services in a speech given at the Massachusetts Institute of Technology (MIT). He named the concept *utility computing* and explained it as follows [BKNT11].

> If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility. [...] The computer utility could become the basis of a new and important industry.

The key idea of utility computing is that customers pay only for resources actually used [Par66]. Thus, utility computing is the predecessor of modern business models based on service provisioning in modern cloud computing. As various services are built on top of each other, cloud computing is often described as a stack.

1.2. The Cloud Computing Stack

In the definition of cloud computing given by National Institute of Standards And Technology (NIST), the cloud computing stack comprises infrastructure, platform and application layers, as depicted in Figure 1.1 [MG11]. The *infrastructure layer* constitutes the basic layer in the cloud computing stack. It comprises physical resources, such as servers, network connections, and further hardware devices. Hardware virtualization technologies, load balancing and frameworks for elasticity and scalability allow for abstracting from particular resources. A distributed server cluster is built by combining multiple server nodes. The *platform layer* is located above the infrastructure providing operating system, middleware and application runtime environment. At this layer, frameworks abstract from system resources providing defined interfaces. Example frameworks at this layer are identity management and persistence. The *application layer* comprises cloud applications that reside partially or completely in the cloud. Cloud applications integrate cloud services to implement application features interacting with system resources via interfaces provided at the platform layer.

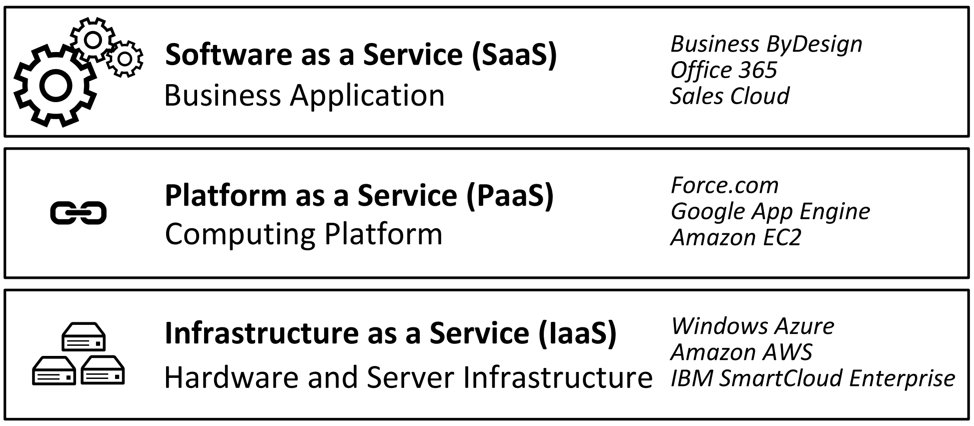

Software as a Service (SaaS)
Business Application
Business ByDesign
Office 365
Sales Cloud
Platform as a Service (PaaS)
Computing Platform
Force.com
Google App Engine
Amazon EC2
Infrastructure as a Service (IaaS)
Hardware and Server Infrastructure
Windows Azure
Amazon AWS
IBM SmartCloud Enterprise

Figure 1.1 The cloud computing stack, related business models, and example services.

For offering resources and services on each layer of the cloud computing stack, new business models are defined.

1.3. Recent Business Models for Cloud Services

Based on a *pay-as-you go* subscription basis, customers are only charged for the amount and time utilizing resources. Customers allocate resources on-demand instead of running their own data centers, shifting the operation risk to the provider. Hence, customers are not responsible for installing, hosting, or maintaining cloud services [BKNT11]. Table 1.1 summarizes the most common business models comprising Software As A Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [Jam12].

How these business models are related to the layers of the cloud computing stack explained in Section 1.2 is depicted in Figure 1.1. The SaaS business model refers to the application level, implying that business applications are provisioned over the Internet as services [Ma07]. SaaS application providers are the successors of Application Service Providers (ASPs) [BLB$^+$00], where the SaaS business model is based on the ASP business model. A SaaS application consists of various, loosely coupled services and components. On-demand SaaS applications are less expensive than on-premise solutions as not the application itself is sold, but rather a license. Such applications reach a larger market segment than on-premise solutions as customers do not have to bear the risk of hosting the application in house. Another advantage from a customer's point of view is that full costs of using an application can be calculated in advance due to a fixed price model, where a monthly fee is paid per desktop license, for instance. Furthermore, providers have lower maintenance cost as SaaS applications are not installed in customer data centers, but on provider servers [Wat05]. Thus, more and more providers of business applications migrate their standard software into the cloud. For instance, Microsoft's office suite comprising spreadsheet and word processors is available on-demand as *Office 365*[1] providing further social network integration. A further example of a SaaS application is *SAP Business ByDesign*[2], a

[1]`http://office.microsoft.com`
[2]`http://www.sapconfigurator.com`

Table 1.1 Comparison of common business models in the cloud adopted from [Jam12].

Business Model	**Service Provider**	**Service Customer**	**Provisioning**
Software as a Service (SaaS)	Application provider	Tenants and application users	Business application
Platform as a Service (PaaS)	Platform or infrastructure provider	Application developer	Middleware, operating system, runtime environment, frameworks
Infrastructure as a Service (IaaS)	Infrastructure provider	System developer	Computing time, storage

Customer Relationship Management (CRM) application for small and medium size companies. Some providers of SaaS applications offer underlying frameworks as PaaS services. For instance, Salesforce provides the CRM application *Sales Cloud*[3] as a SaaS application and the underlying platform *Force.com*[4] as PaaS.

Marc Benioff, founder of Salesforce, coined the term PaaS for the business model of providing platform services. PaaS abstracts from the physical layers and provides middleware, application containers and platform services to application developers at the platform level. Further services offered at this level are, for instance, identity management, persistence, load balancing and performance isolation. Thus, application developers do not need to take care of installation and maintenance of the middleware and platform services. Due to information hiding, developers access provided middleware frameworks via defined interfaces. Further examples of PaaS offerings are *Google App Engine*[5] and *Amazon Elastic Compute Cloud (Amazon EC2)*[6].

The IaaS business model refers to the infrastructure level of the cloud computing stack, providing fundamental services such as storage and computing capacities to system developers that have full control of the installation of operating system, middleware and applications. Examples for IaaS offerings are *Microsoft Windows Azure*[7], *Amazon Web Services*[8], and *IBM SmartCloud Enterprise*[9]. Further *X as a Service* business models have been proposed, but are not discussed here as they are out of scope of this work. This work focusses on the provisioning of SaaS applications and their dependencies on platform and infrastructure resources. Depending on the domain and purpose of a SaaS application, various deployment models are to be distinguished that are further explained in the following.

1.4. Deployment Models of Cloud Services

Public, private, hybrid, and community cloud are to be distinguished according to the location of the cloud infrastructure and access restrictions on the provisioned services. Each deployment model aims at different concerns and has different access scopes, as depicted in Table 1.2 [MG11].

1.4.1. Public Cloud

Services in a *public cloud* are available to various customers. Using a public cloud is convenient for highly available and distributed web applications that perform computationally intensive calculations and therefore have varying workload, such as event web sites or applications with complex mathematical calculations. A public cloud infrastructure is often spread over multiple

[3]`http://www.salesforce.com/sales-Cloud`
[4]`http://www.force.com`
[5]`https://appengine.google.com`
[6]`http://aws.amazon.com/ec2`
[7]`http://www.windowsazure.com`
[8]`http://aws.amazon.com`
[9]`https://www.ibm.com/de/smartcloud`

Table 1.2 Cloud deployment models.

Deployment Model	Deployment Location	Accessibility
Public cloud	World wide	Services publicly available to independent customer
Private cloud	Organization wide	Inside an organization
Hybrid cloud	Combining public and private cloud	Critical parts processed inside an organization, others in a shared public cloud infrastructure
Community cloud	Multiple organizations	Multiple authorized organizations share artifacts of common interest

countries. Thus, it cannot currently be guaranteed that data in a public cloud does not leave a certain country. Hence, sensitive data, where legal restrictions necessitate that the data does not leave a particular country, cannot be processed in a public cloud [MH10].

1.4.2. Private Cloud

A *private cloud* differs from a public cloud mainly in the access restriction on resources and in the location of the infrastructure. The infrastructure is hosted in-house of an organization and is only accessed inside the organization, where infrastructure and platforms to realize load balancing are similar to the public cloud. Thus, the organization guarantees that data is processed in a particular country. However, the data center hosting the private cloud is probably not as scalable to the number of requests as a public cloud as there are less computer nodes available. A private cloud is more cost intensive as an organization cannot benefit from sharing resources with other organizations compared to a public cloud.

1.4.3. Hybrid Cloud

A *hybrid cloud* combines advantages of a public and a private cloud. Infrastructure processing sensitive data is hosted by the organization itself and protected against access from outside, whereas public cloud services are integrated to benefit from scalability and elasticity of the public cloud infrastructure on high load peaks. Particular cloud services are further explained in Section 1.5.3.

1.4.4. Community Cloud

In a *community cloud* the infrastructure is hosted by potentially multiple organizations, and is only accessed by authorized organizations being members of that particular community. A community cloud aims at sharing resources, as well as community-specific data. For instance, flight information is offered to different airlines in a community cloud and any participating airline provides its own data and consumes data of other members.

As SaaS applications aim at serving various different customers, they are usually deployed in a public or community cloud. Such a publicly available SaaS application demands for a sophisticated customer management especially if it is configurable and multi-tenant aware, as explained in Section 1.5.6. In contrast, SaaS applications deployed in a hybrid and private cloud are available to a single customer, thus having less strict requirements on customer management.

1.5. Characteristics of Cloud Computing and Cloud Applications

Cloud computing is a paradigm for providing computing resources as services over the Internet consumable on heterogenous devices. New web technologies allow shifting complex computerized calculations into a server cluster [HM05]. The NIST defines cloud computing as follows.

Definition 1.2 (Cloud Computing [MG11]). *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.*

Cloud computing is applicable for complex calculations in a server grid and platforms with anticipated and unanticipated workload peaks [FE10].

The main characteristics of cloud computing are explained in the following, which are scalability, elasticity, resource sharing and resource provisioning as a service.

1.5.1. Scalability and Elasticity

Scalability in the cloud refers to the ability of a system to handle growing amounts of workload and to improve data throughput on-demand. From a business view, it is an important property of cloud services and is depicted by *Scale fast or fail fast.* Successful cloud services start by serving a small amount of users. On increasing demand, the services scale to handle up to hundreds of thousands of users simultaneously, and scale down on decreasing demand. Scalability is realized technically on infrastructure level, where horizontal and vertical scaling are to be distinguished. *Horizontal scaling* is also known as *scaling out* which implies that more nodes are added to the server grid if currently allocated resources are not sufficient and nodes are removed if no longer needed. Virtualization technologies are applied to abstract from potentially heterogeneous resources to enable horizontal scalability. In contrast, *vertical scaling*, also called *scaling up*, denotes that a single server node is enhanced with resources, such as main memory [JOP11].

Beside scalability in terms of users concurrently accessing a service, *functional scalability* is important for cloud applications. Especially, enterprise applications with small and medium size companies need to be scalable in terms of functionality to always satisfy the demands of their customers. However, companies grow and shrink according to the current market situation and their functional requirements on enterprise applications scale accordingly. Most enterprise applications are provided as an all-in-one solution which does not completely cover customer requirements. Applications that are configurable can only be configured once at design time but are not to be changed during runtime. Most of the current available enterprise cloud applications are not variable at runtime regarding functionality and quality. However, cloud applications could reach a larger market segment and create a unique selling point opposite of conventional applications by providing scalability in terms of functionality and quality.

Elasticity is the capability of a system to allocate resources according to requests and service demands. The system scales transparently conveying the illusion of unlimited resources. The capacity adjusts over time allowing for handling rarely occurring peak periods with high server loads. For instance, high workload leads to an allocation of many resources, where low workload leads to a release of these resources making them available for other services. This is achieved by offering transparent load balancing [JOP11].

From a customer's point of view, cloud computing offers unlimited storage and computing resources. Customers can flexibly allocate and release resources depending on their varying business demands [AFG$^+$09]. Cloud computing is beneficial for providers as resources are shared among customer instances, thus enabling a cost efficient utilization of resources.

1.5.2. Resource Sharing

Resource sharing plays a central role in cloud computing enabling economics of scale. Customers share hardware and software resources at all levels of the cloud computing stack explained in Section 1.2, where the particular resource location is hidden from the customers [MG11]. Different kinds of resource sharing are to be distinguished according to the resource types hardware, application instance and data as summarized in Table 1.3. *Virtualization* is based on technologies at infrastructure level enabling sharing of server hardware and network resources among system users. At platform level, middleware components, application frameworks and containers are shared between different customers. Depending on the cloud deployment model explained in Section 1.4, infrastructure and platform resources are shared inside a company in a private cloud scenario, and together with several organizations otherwise.

Table 1.3 Overview of levels of resource sharing classified by the type of the shared resources.

Classification	**Hardware and Platform**	**Application Instance**	**Data**
Virtualization	x	-	-
Multi-tenancy	x	x	-
Knowledge Sharing	-	-	x
Community Multi-tenancy	x	x	x

Virtualization is the foundation for resource sharing at business application level. At this level, either single application functionality or a whole application instance is shared among different customers. An architectural paradigm to realize resource sharing in cloud applications is *multi-tenancy*, which is explained in detail in Section 1.5.5.

Furthermore, data can be shared between various customers of the same community. The sharing of data is called *knowledge sharing*. Combining data sharing with sharing of the application instance and hardware resources leads to *community multi-tenancy* according to the community cloud deployment model, as explained in Section 1.4.4.

1.5.3. Cloud Services

In the cloud, resources are provisioned as *services* and, thus, shared between service consumers. They are consumed by end-users on heterogenous devices, such as laptops, tablet computers, smartphones and smart televisions, as depicted in Figure 1.2. Cloud services comprise any kind of hardware and software resource. For instance, hardware is offered in terms of storage, computation, and network connections, while software is offered as small apps and web-based enterprise applications. Data is also offered as a service, comprising community knowledge and collected big data. An analyst from IDC defines cloud services as follows.

Definition 1.3 (Cloud Service [IDC09]). *Cloud services are consumer and business products, services and solutions delivered and consumed in real-time over the Internet.*

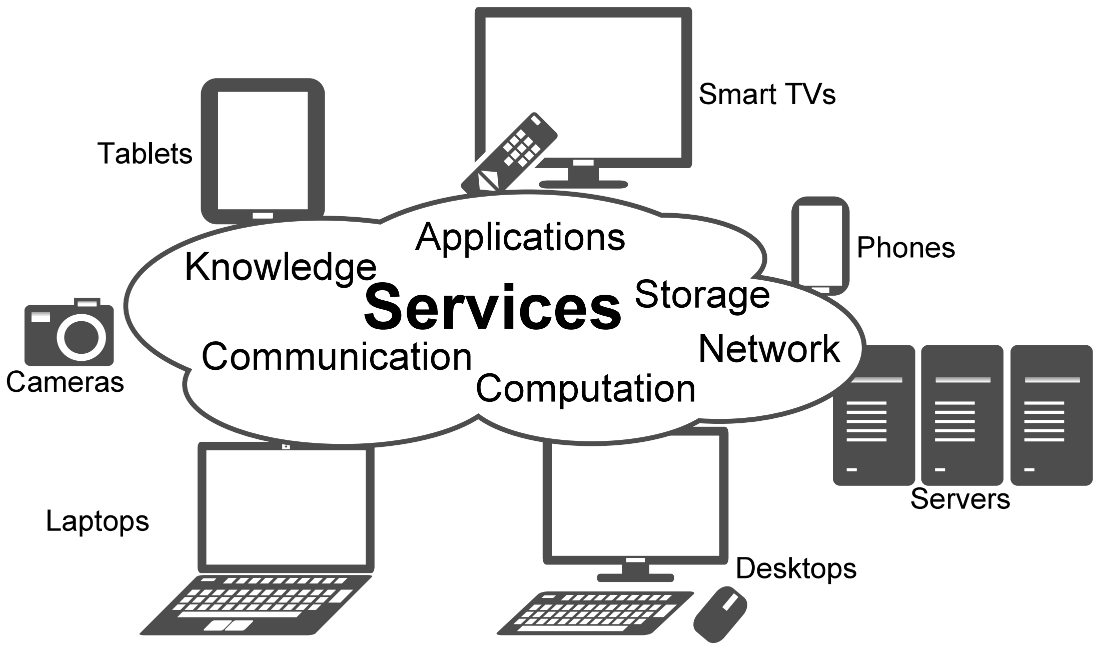


Figure 1.2 Various types of services are provided in the cloud consumable on heterogenous devices.

Examples of cloud services are applications, knowledge, communication platforms, computation power, network infrastructure, and storage capacity, to name but a few. A service in the cloud is available almost in real time over the Internet, often easily accessible by customers without detailed IT knowledge, where customers are companies and end-users. Cloud services are highly scalable in amount and time and associated to infrastructure, platform and application resources related to the layers of the cloud computing stack explained in Section 1.2. The billing of the services is based on the resource consumption enabling new business models, as discussed in Section 1.3. Cloud offerings enables customers to subscribe and unsubscribe on a self-service base without further assistance of the provider.

Self-Service Portals

Cloud providers offer online self-service portals available 24 hours a day, where customers can sign up for a service that is instantly available without any delay in deploying the system. Customers that are supposed to serve themselves, better understand the value of the product and are more willing to test a service as the entry barrier is low [CMRD13]. Examples for SaaS applications providing self-service portals are the collaboration application *37signals*[10] and the CRM application *Zoho*[11].

An advantage of cloud applications combined with a SaaS business model could be to offer functional scalability by means of self-service portals to the customers. Thus, customers are enabled to extend or decrease their service subscription according to their current demands. A requirement for this approach is an architecture capable of provisioning the reconfigured application just-in-time.

A challenge in providing a self-service portal in practice is the compliance of the available application with country-specific law and legal restrictions.

Legal Restrictions

In many cases, not all services are offered in all countries due to country law. Furthermore, customers may only use cloud services if providers guarantee that customer data is neither stored, nor processed in countries with different applicable laws. For instance, the European data protection directive is valid Europe wide, but interpreted differently in each European country [95/46/EC]. Thus, a service agreement between provider and customer has to be signed. Such a contract is called Service Level Agreement (SLA) defining the provided function volume, non-functional qualities that must be met by the service, as well as support and responsibilities of the contractors [Ver99]. SLAs can be defined as follows.

Definition 1.4 (Service Level Agreement). *A Service Level Agreement is a contract between provider and customer that defines guaranteed quality levels of a service and support modalities.*

[10]`http://37signals.com`
[11]`http://www.zoho.com`

Within the private sector, SLAs are signed to guarantee a certain service quality level, to define contract conditions, and to specify support modalities to satisfy customer needs. SLAs are not agreements about costs, but describe measurable quality attributes of the application and are commonly referred to as Quality of Service (QoS) aspects [Ver99, KL03]. Examples of these attributes are availability, reliability, performance and security levels. Furthermore, an SLA explicitly specifies exclusion of service liability. The contract contains an agreement of how to deal with operational problems covering, for instance escalation strategies, help desk provisioning, hot-line, and agreed severity levels. Terms of use are a relevant legal aspect of a service agreement varying between customers. Beside the content of an SLA, further aspects may be variable in a configurable cloud application, as explained in the next section.

Automating the compliance with SLAs enables applications to scale not only in functionality but also in quality. Such a scalable application is variable with respect to different configuration parameters, which are explained next.

1.5.4. Configuration Capabilities

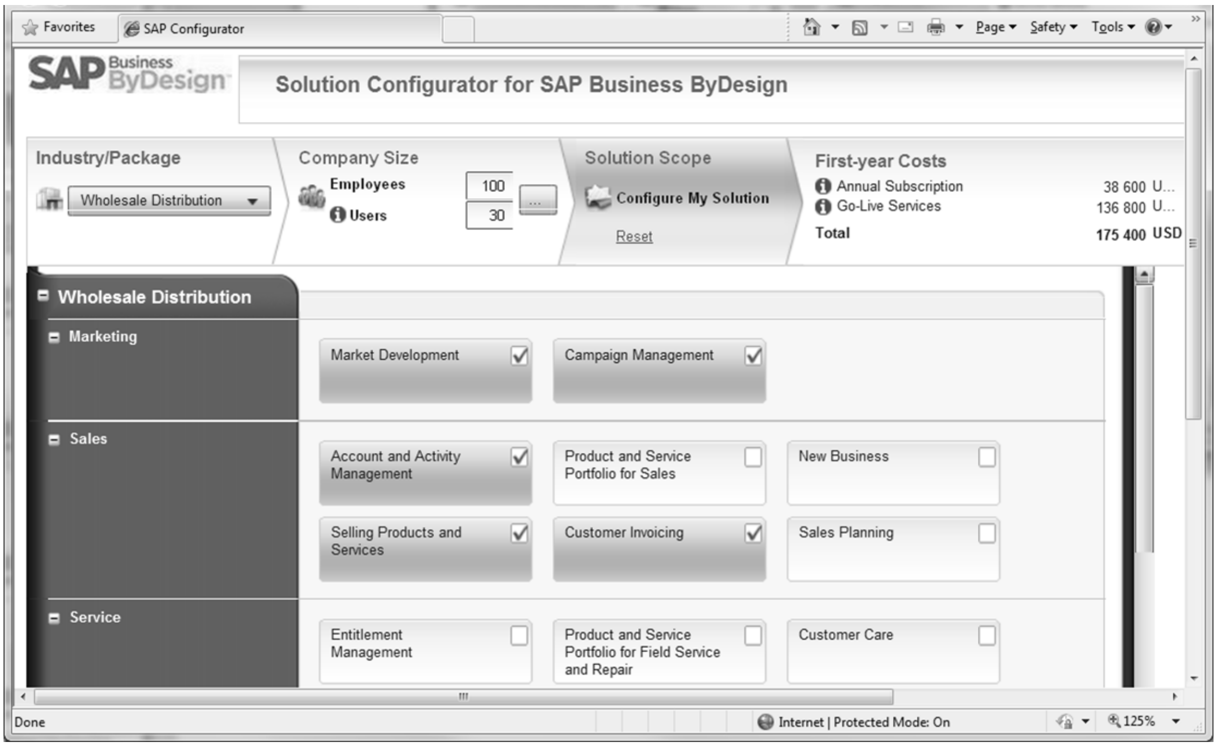
Cloud applications are configurable to address varying customer requirements. An example of a configurable cloud application is *SAP Business ByDesign*[12] offering various configuration options as a self-service to customers. Business ByDesign is an information system targeting small and medium size organizations in different domains. The application is highly configurable comprising various enterprise management components, such as executive management support to identify organization goals, financial management components for controlling, CRM and human resource management to administer customers and staff, as well as project management, and supply chain management, to name but a few.

A customer can choose from different application functionality and configure the number of users, as exemplified in the screenshot in Figure 1.3. Business ByDesign is provided as a SaaS application and the price for the SaaS offering is calculated-based on the customer's feature selection. The application features are grouped in a configuration wizard according to their category. For common use cases, a feature pre-selection is also available. Additionally, if a customer selects a feature that has further dependencies to other features, a menu appears notifying the customer that further features need to be selected or deselected.

In general, variability of SaaS applications can be classified in different categories. Various categories of variability are identified in cloud applications as adopted from [SZG$^+$08, Nit09] and shown in Table 1.4. Depending on the application domain, the degree of configurability varies, and thus not all variability categories listed are configurable in an application. Most of the currently available applications are rarely configurable as enabling configuration options causes further costs.

Besides configurability, multi-tenancy is an important characteristic of cloud applications and is explained in the next section.

[12] `http://www.sapconfigurator.com`



Solution Configurator for SAP Business ByDesign

Industry/Package: Wholesale Distribution

Company Size: Employees 100, Users 30

Solution Scope: Configure My Solution, Reset

First-year Costs

Annual Subscription	38 600 U...
Go-Live Services	136 800 U...
Total	175 400 USD

Wholesale Distribution

- Marketing
  - Market Development ✓
  - Campaign Management ✓
- Sales
  - Account and Activity Management ✓
  - Product and Service Portfolio for Sales ☐
  - New Business ☐
  - Selling Products and Services ✓
  - Customer Invoicing ✓
  - Sales Planning ☐
- Service
  - Entitlement Management ☐
  - Product and Service Portfolio for Field Service and Repair ☐
  - Customer Care ☐

Figure 1.3 The cloud application Business ByDesign is configurable in a self-service portal.

1.5.5. Multi-Tenancy

Multi-tenancy is said to be the key success factor for sustainable cloud applications due to resource sharing capabilities, where configurable and multi-tenant aware applications address even a larger profiting market segment [Aik11]. Integrating various specifications given in literature, multi-tenancy is defined as follows [Aik11, CCW06, BZ10].

Definition 1.5 (Multi-Tenancy). *Multi-tenancy is an architecture paradigm for platforms and applications serving multiple customers at the same time while guaranteeing isolated system access. Customers are referred to as tenants in this context and are logically separated, but physically integrated in a multi-tenant environment.*

Hence, multi-tenancy comprises two main concepts, (i) the sharing of system resources and data between application users, and (ii) restricted user access on resources, application functionality and data with respect to an affiliated tenant. Not only hardware resources, but also application functionality and potentially data is shared among the tenants, as explained in Section 1.5.2.

Tenants share the same application instance, where performance, control flow and data flow of the users of these tenants are isolated [CCW06]. Thus, a multi-tenant aware application is a special form of a multi-user web application. In multi-user and in multi-tenant applications, control and data flows between users are separated. This is achieved by a session management

Table 1.4 Configuration options of cloud applications.

Configurability	**Description**
User interface	The user interfaces of an application vary according to the expert level of a user. Moreover, different human machine interactions are available, such as textual input and voice input. Furthermore, the branding for different companies changes the look and feel of the user interface. Country-specific internationalization support changes the displayed language and currency.
Features	Applications vary in the amount and type of functionality. For instance, the amount of functionality in a freely available version of a business application is limited, whereas the commercial application is not.
Business processes	Business processes, rules and tasks implemented in an application vary depending on customers. Besides the results of a process can be the same while process steps vary.
Quality	The offered QoS comprising security, performance and availability may vary. Service quality is contracted in an SLA.
Data management	The persistence of data varies and data objects may have different attributes in different applications. In a particular context, the precisely measured data value is needed, whereas in another context a coarse grained value abstracted in an equivalence class is sufficient.
Access control	The access control policies are configurable. They specify who is allowed to access and interact with the system.
Usage context	The ways of interacting with a system may vary. A cloud application can be accessed via a smart-phone application, a web site, and a client application on a desktop.
Reports	The scope of reports varies in an application according to the user creating a report. The content of a report depends further on security policies and relevant data.

to handle the data and control flow of different users accessing the application concurrently by enabling stateful communication. Furthermore, load balancing methods are applied to guarantee that users of each tenant do not influence their application performance [CA82].

The difference between multi-user and multi-tenancy is in the data access. In multi-user applications, a user accesses restricted user-specific data, as well as shared application data available to the public. In contrast, a further permission layer is inserted in between these two in a multi-tenant application. The access on data is therefore restricted on three levels in a multi-tenant aware application, as depicted in Figure 1.4.

Data is partitioned into user-specific data, shared restricted tenant-specific data and publicly available shared data. Hence, tenant-specific data is access restricted to users of a particular tenant, while this data is shared among all users of the tenant. For multi-tenant aware applications

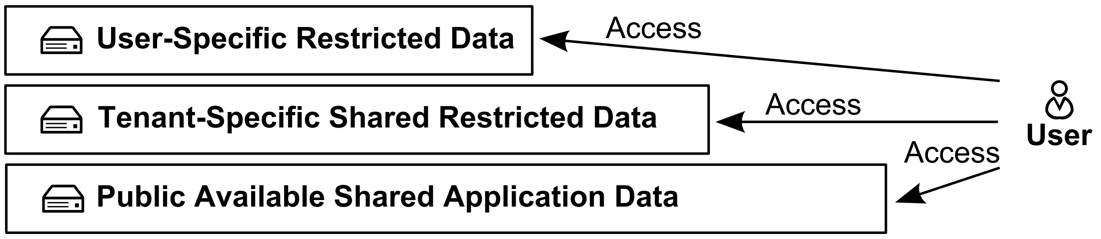

User-Specific Restricted Data
Tenant-Specific Shared Restricted Data
Public Available Shared Application Data
Access
Access
Access
User

Figure 1.4 Three-layer data access restriction in multi-tenant aware applications.

users are identified at application login and associated with a tenant to enable data access restrictions. Additionally, depending on the application type the amount of restricted and shared data varies.

Sometimes multi-tenant applications are confused with multi-instance applications. Figures 1.5 and 1.6 visualize the difference between single-instance multi-tenancy and multi-instance applications. Common to both scenarios is the sharing of hardware among customers realized by a virtualization layer on top of the physical infrastructure. In multi-instance scenarios as displayed in Figure 1.5 a virtual machine and operating system is deployed per customer and thus, the application instances of the customers A, B, and C are separated as well. Hence, resources between the customers are only shared on the hardware and platform level. The applications in such scenarios equal common on-demand applications and are not multi-tenant aware.

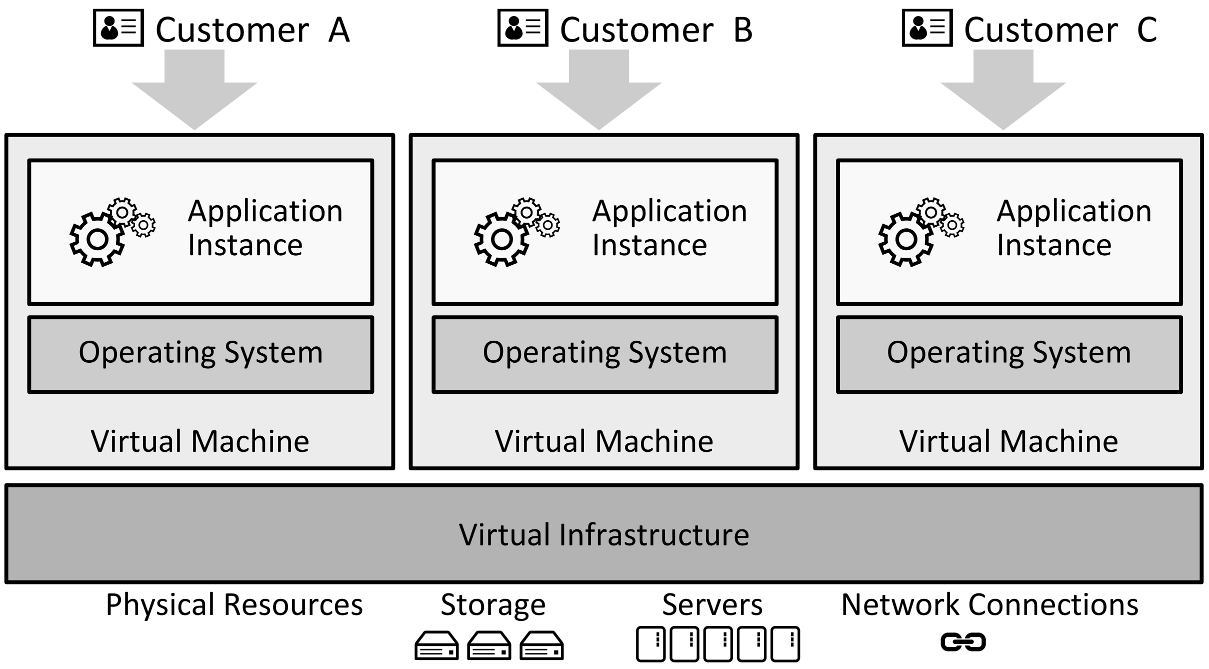


Figure 1.5 Multi-instance application deployment.

A multi-instance approach combines the advantages of cloud storage and computing with conventional on-demand applications. The approach does not scale as for every new tenant, a dedicated virtual machine must be deployed. Furthermore, maintenance costs are high, as every application instance must be maintained separately [CC06].

Multi-tenancy enables scalable and efficient applications and is exemplified in Figure 1.6. Efficiency means resources are allocated for the customers according to their demand and the presence of further tenants is transparent for each customer. A multi-tenant application is scalable in terms of the number of concurrent accesses to an application. For example, the single software instance depicted in the figure serves all three customers `A`, `B`, and `C` and their users. The application is efficient, as for customer `A` accessing the application instance, the concurrent utilizations of customers `B` and `C` are transparent. Moreover, the application scales with respect to the number of served customers.

To achieve scalability and efficiency, appropriate methods are implemented in the underlying platform. Applications with one database per tenant and a shared database for all tenants are to be further distinguished. An application with a shared database has the highest efficiency and is referred to as pure multi-tenancy [CCW06]. However, optimization methods for enhancing scalability and resource efficiency of multi-tenant applications are out of scope of this work. Furthermore, customers may be concerned about security and service availability of a configurable multi-tenant application. These properties are relevant for operating an application, but are considered out of scope of this work. This thesis rather focuses on application scalability in terms of functionality and quality.

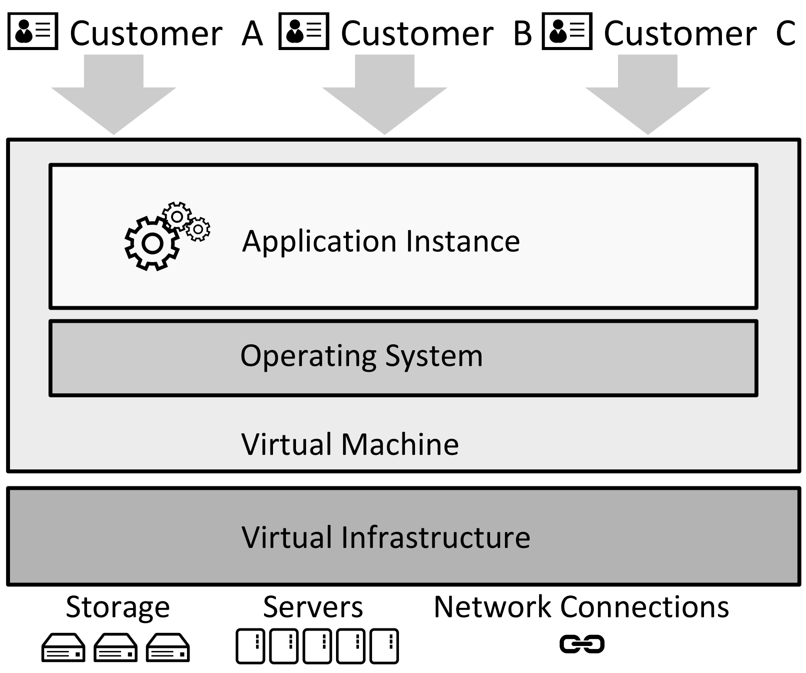


Figure 1.6 Multi-tenant single-instance application deployment.

1.5.6. Configurable Multi-Tenant Aware Applications

In general, different customers have different application requirements, as discussed in Section 1.5.4 and exemplified in Figure 1.7. In the visualized scenario, customers `A`, `B`, and `C` have their own requirements and thus varying configurations incorporated in the single application instance as an extension to the multi-tenant application depicted in Figure 1.6. An analysis of applications available on the market reveals that configuration options of current multi-tenant aware applications are rather limited [Cha13]. It is an open challenge to provide highly configurable multi-tenant aware SaaS applications as they require for a comprehensive management of configurations for a large number of diverse customers.

Due to resource sharing, configuring the application on source-code level is complex, error-prone and does not scale [BZ10]. Hence, the focus of this work is to apply configuration management on an abstract level with automated evaluation of configuration options that can be easily integrated in a self-service portal. In configuration management, various stakeholders with varying authority are to be considered as their configuration decisions influence each other.

1.6. Stakeholders Participating in the Configuration Process

Generally, the set of stakeholders comprises organizations, particular groups and individuals. They are separated into two categories in terms of business of a company according to the *ISO 10006:2003* standard for quality management in projects [ISO 10006]. *Internal* stakeholders

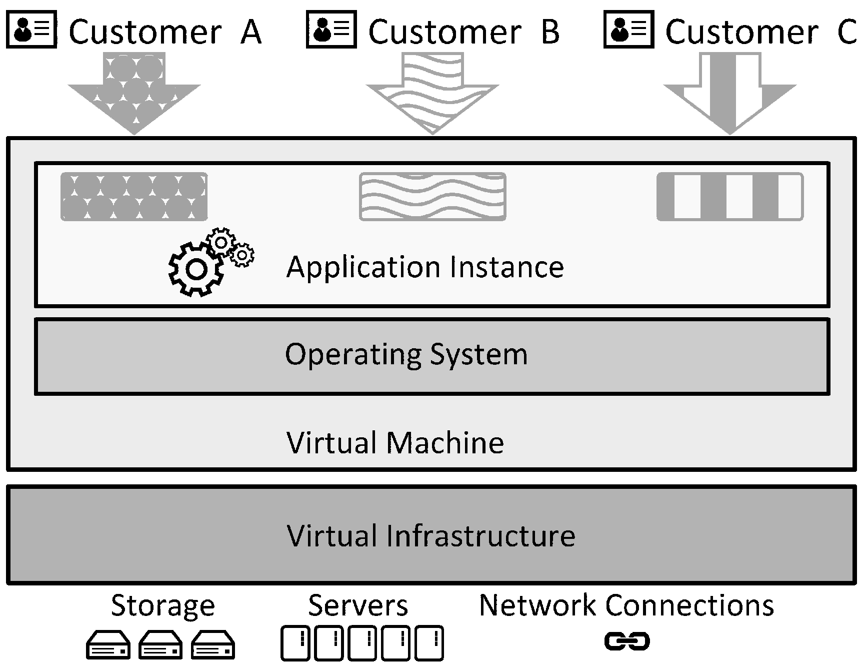


Figure 1.7 Configurable multi-tenant aware single-instance application.

are directly involved in a business inside the company, such as employees, and management. In contrast, *external* stakeholders are affected by business decisions of the company, such as customers, suppliers, and shareholders. Adopted from the *ISO 10006:2003* standard [ISO 10006] a stakeholder is defined as follows.

Definition 1.6 (Stakeholder [ISO 10006]). *A stakeholder is an individual, group or organization having an interest in a particular project or are involved in the project.*

In general, multiple different stakeholders participating in the configuration process of multi-tenant aware SaaS applications can be identified [AFG$^+$09]. The stakeholders have varying configuration responsibilities that further may have interdependencies. Stakeholders actually involved in the configuration depend on the domain of the SaaS offering and according to the application context.

Figure 1.8 exemplifies common stakeholders. In this example, a *resource provider* is offering infrastructure and platform services, and an *application provider* offers application functionality. Customers that subscribe for the SaaS offering are therefore referred to as *tenants*. *Users* access the application and use the provided application functionality. Users are employees or further customers of a tenant. In other scenarios the set of involved stakeholders may differ. For instance, particular providers may be further distinguished, such as infrastructure provider and platform provider. Furthermore, a platform provider may have different experts for parts of the platform, such as a database expert and a security expert. In a different scenario a multi-layered tenant structure may be defined, such as the SaaS application is customized and rented to tenants, which incorporate own customers as their tenants that have further customers, and so forth.

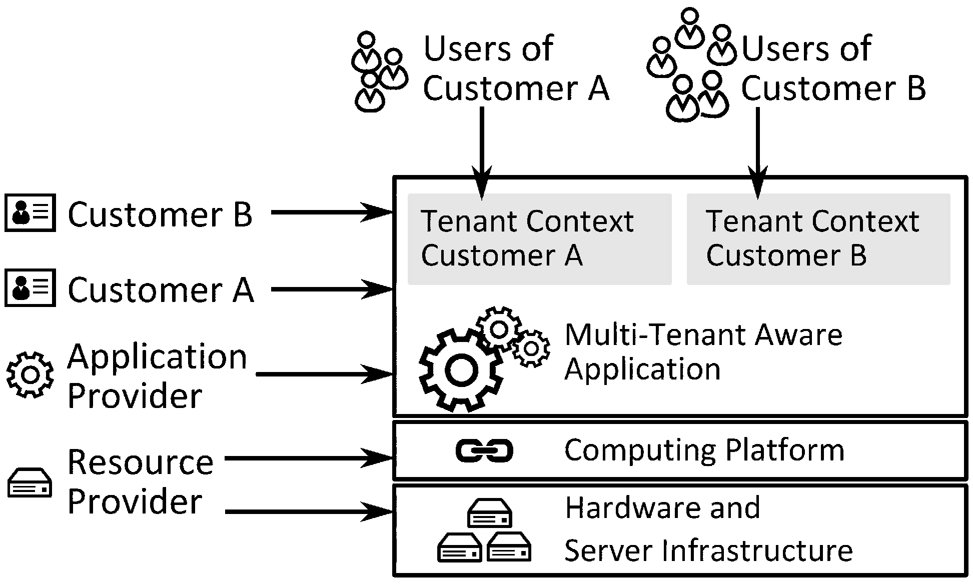

Users of Customer A
Users of Customer B
Customer B
Customer A
Application Provider
Resource Provider
Tenant Context Customer A
Tenant Context Customer B
Multi-Tenant Aware Application
Computing Platform
Hardware and Server Infrastructure

Figure 1.8 Common stakeholders involved in the configuration of a cloud application.

Particular stakeholders of a cloud application are often not known beforehand. For instance, customers and their users sign up for the service during application runtime, where the application provider is already known during setup of the application. Thus, responsibilities and configuration opportunities of stakeholders are identifiable by stakeholder types. For instance, the configuration possibilities of all customers are equal, and identifiable by the type *tenant*. Hence, concrete customers `A` and `B` are identified as *tenants* during application runtime.

Configuration decisions of stakeholders potentially influence each other. For instance, if a resource provider chooses a particular application container, only compatible application functionality is available to a tenant. Furthermore, between tenant and provider a *tenancy contract* is signed in the context of an SLA to specify offered functionality, as well as the QoS level that should be fulfilled. However, configuration decisions of some stakeholders are independent. For instance, all tenants have the same configuration opportunities, while concrete tenants are mutually independent from other tenants in conducting configuration decisions. Such independent tenant configurations are integrated on architectural level and multiple tenants share the same hardware and software resources. Here, the integration on architectural level is considered out of scope of the configuration process.

However, some stakeholders have more influence in the configuration process than others and each stakeholder is restricted to particular configuration decisions. This can be achieved by applying access control mechanisms.

1.7. Access Control in Shared Cloud Environments

In multi-user environments, especially relevant in shared cloud environments, operations on a system need to be restricted to prevent unauthorized actions and misuse. Mechanisms of access control ensure this and are defined as follows [Ben06].

Definition 1.7 (Access control [Ben06]). *Access control aims at limiting the access on system resources, services and information to authorized entities only, whereas an entity may be a person or a computer process.*

In Computer security, access control covers *authorization* to specify access rights, *identification and authentication* to verify the identity of a subject, *access approval* to grant or reject access during operation, and *auditing* and logging to control performed actions and to identify security violations. A *subject* is an actor performing an operation on the system, whereas a subject may be a person or a computer process. The operation is performed on an *object* in the system being a hardware or software resource [SV01]. In modern multi-user systems, RBAC is the prevailing paradigm for restricting the access on system resources as it is the best representation of processes from the real world [FKC07].

1.7.1. Role Based Access Control

Since its introduction in 1992, RBAC became a common access control mechanism in business applications. System access is restricted via roles reflecting job functions. The generic concept of roles abstracts from individuals and increases durability since job functions in a company do not change as often as individuals.

Initially, the formal model of RBAC was introduced by Ferraiolo and Kuhn in 1992 [FK92] to administer security in large networks. Sandhu et al. proposed a software framework for RBAC in 1996 [SCFY96]. Both models were incorporated in 2000 by the NIST to eventually create the unified NIST RBAC model [SFK00]. Subsequently, the NIST standard was adopted by the International Committee for Information Technology Standards (INCITS)[13] of American National Standards Institute (ANSI) and released as the standard ANSI INCITS 359-2004 [INCITS 359] in 2004. The standardization emphasizes the practical relevance of RBAC to manage user rights in enterprise software. For instance, RBAC is applied in the enterprise application *SAP R3* [SAP09].

Due to the evolution of RBAC, four levels are distinguished as shown in the schema in Figure 1.9, which is adopted from [SFK00]. Each level comprises different functionality, where $RBAC_0$ contains basic concepts, $RBAC_1$ and $RBAC_2$ are two separate extensions of the basic concepts, and $RBAC_3$ comprises all concepts of the other three levels. The basic model $RBAC_0$ embodies the fundamental concepts of roles, permissions, objects, subjects and their relations.

Figure 1.10 depicts the main concepts of RBAC. A *role* represents a job function in an organization owning a set of permissions. A *subject* is an active entity that performs operations, and may be a human user of the system or a computer process. An *object* is a passive entity used or consumed during operation execution. It is further categorized into time, information, resource, and processor entities. A *permission* represents an authorized interaction between a subject and an object. Furthermore, a hierarchy among roles in RBAC is additionally defined as a partial order relationship in $RBAC_1$.

The role hierarchy may be assumed as multiple inheritance relation, where child roles inherit permissions from parent roles. The *role hierarchy* defines an inheritance relation as known from

[13]www.incits.org

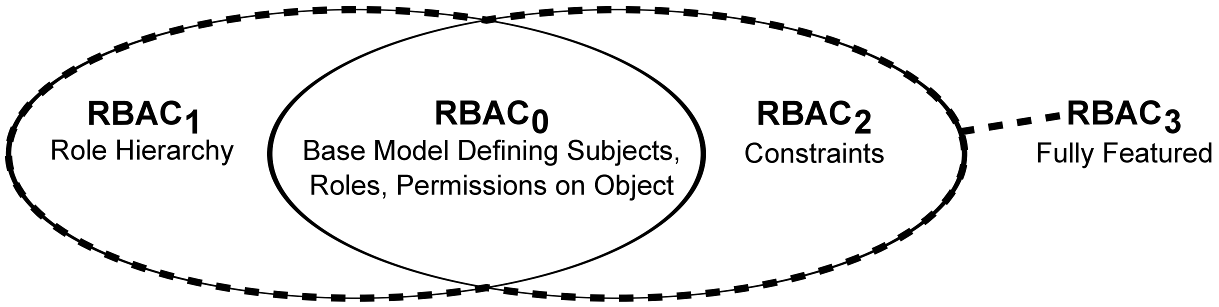


Figure 1.9 The four levels of RBAC displayed as Venn diagram.

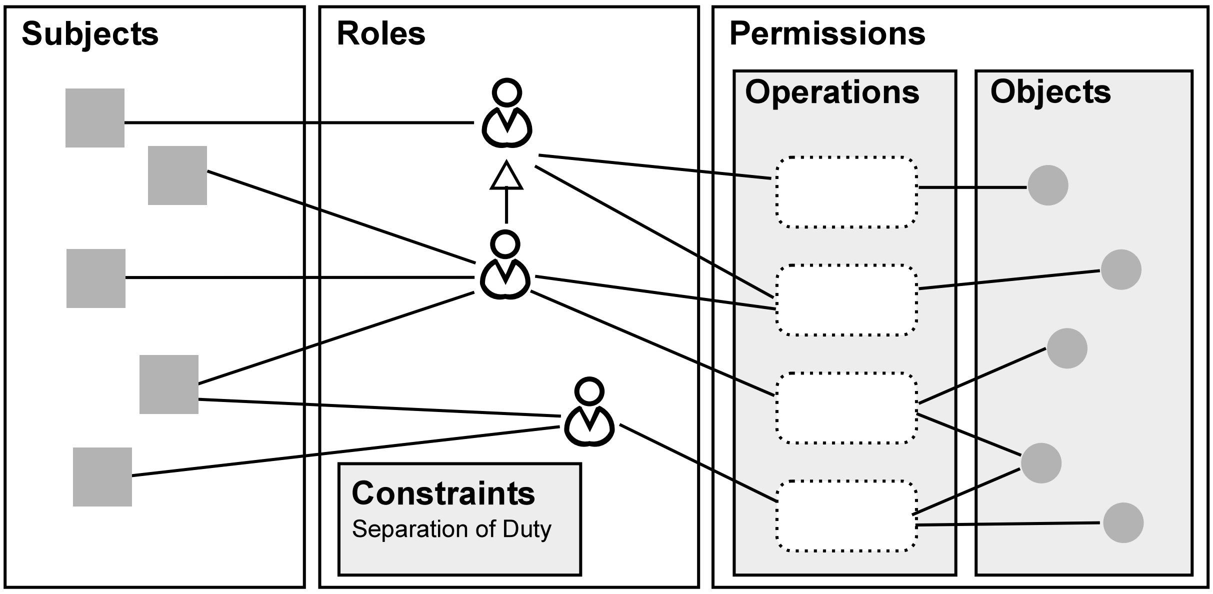


Figure 1.10 Schematic representation of RBAC concepts.

object-oriented modeling [RBP+91]. It reflects a relationship among roles specifying that a role inherits permissions from other roles. The role hierarchy does not have circular relationships and the association between roles is transitive. Moreover, *constraints* are added in $RBAC_2$ as another extension to RBAC_0 to separate duties. Finally $RBAC_3$ as a top group includes the concepts of the other three levels [SFK00].

The combination of RBAC and self-service portals enables access control of different stakeholders in particular cloud services. For instance, the online marketing company HubSpot Inc.[14] applies RBAC to offer end users direct access to the management console of Amazon Web Services (AWS)[15], as well as to the Cloud Control Panel of Rackspace Inc.[16].

> Early on, portals didn't give you a lot of controls – they allowed everybody to access everything. Now it's possible to offer adequate access without giving everybody the keys to the kingdom.

said Jim O'Neill, chief information officer at Cambridge about self-service portals applying RBAC [Par13].

1.8. Challenges in Providing Reconfigurable Cloud Applications

One of the benefits of cloud computing is scalability, as explained in Section 1.5.1, where service subscriptions are easily changed, and resources are flexibly scaled up and down according to

[14]http://www.hubspot.com
[15]http://aws.amazon.com
[16]http://www.rackspace.com

customer demands. Especially enterprise applications require scalability on application level with respect to application functionality. Customers, such as small and medium-sized companies, require the ability to change their configuration according to their current business situation. Smaller companies choose a limited range of functionality with the ability to scale up by adding more functions later on. In contrast, if a company requires less functionality due to business reasons, the application must scale down.

Moreover, a self-service portal enables customers to configure and reconfigure an application on themselves without the need for interacting with the provider. Thus, customers are served 24 hours a day, as explained in Section 1.5.3. A configured and reconfigured application must be available to the customer just-in-time without a manual deployment step conducted by the provider. Thus, automated provisioning of tailored cloud applications demands for an application architecture, capable of instantiating a configuration just-in-time and supporting reconfiguration if configuration parameters change. As explained in Section 1.5.5, a configurable multi-tenant aware application architecture is suitable, where new and changed configurations can instantly be integrated in a running application instance without further manual interaction.

To analyze, if SaaS applications deployed on public cloud infrastructures are multi-tenant aware, customizable, and support just-in-time provisioning, a structured empirical analysis of cloud applications available on the market was conducted [Cha13]. The analysis shows that only 20% of the observed 85 SaaS applications are multi-tenant aware, and these applications offer severely limited configuration options to the customers. The functionality of the applications is customizable by integrating plug-ins on source code level. None of the observed applications enables customers to configure application functionality directly on an abstract level as a self-service. Additionally, most of the configurability and customizability solutions implemented in state of the art applications are not reusable in other application scenarios as they are realized ad hoc for a particular purpose. Hence, stakeholders involved in the configuration process are also fixed and implemented in a pragmatic way.

The lack of configurability can be ascribed to the absence of (i) consistent variability and configuration management and (ii) a flexible multi-tenant aware architecture being capable of adapting to different tenant configurations while software and hardware resources are shared. A challenge is to provide a generally applicable approach to automate the configuration of various applications supporting different numbers and different types of stakeholders concisely.

Configurable SaaS applications with varying integrated tenant configurations require for unified variability definition of application functionality and configuration parameters, as discussed in Section 1.5.6. A tenancy contract between application provider and tenant defines general terms, such as the duration of the tenancy. In addition, properties on how to provision application support are provided. Moreover, technical conditions, such as the provisioned application functionality are specified. A tenancy contract further defines QoS guarantees, such as adhering levels of availability, security, and geographical or legal restrictions of the application. Tenants have varying requirements on functionality and QoS, the variability of both must be handled. Another challenge is in the provision of a multi-tenant aware architecture that supports variability among different tenant configurations, while sharing hardware and software resources among tenants of the same application instance.

The identified challenges can be addressed by applying methods from SPL engineering.

1.9. Benefits for Cloud Computing by Applying Software Product Line Engineering

SPL engineering aims at developing various similar products at the efficiency of a single product by reusing established software components. In the embedded systems domain, SPL engineering is already well accepted and widely applied to develop various customized products by reusing single artifacts. Prominent examples are the automotive industry in general, as well as the highly variable Linux kernel [TH02, SSSPS07]. Applying SPL engineering methods in information systems, and especially in the cloud computing domain, is a recent trend in software development [MA02, Mie10].

Well accepted techniques from SPL engineering are of interest for different areas of cloud computing as the recent SCArVeS workshop[17] reveals. Generally, SPL engineering offers various tangible and intangible benefits to an organization. Compared to traditional single system development, measurable benefits are higher product quality, faster time-to-market, higher productivity, and cost savings. Additionally, not directly measurable intangible benefits are customer and professional satisfaction [Coh03].

Benefits from a system architect point of view are improved maintainability of an application and its variations. Variability is expressed explicitly on different levels of detail and the dependencies between configuration parameters are modeled uniformly.

From a customer point of view, the application of SPL leads to customizable cloud applications at lower service costs. Configuration processes are simplified and feasible by customers in self-service portals. Hence, customized applications are available just-in-time without manual provider interaction. Additionally, SPL methods support the provision of customizable cloud applications scalable in terms of functionality according to customer demands. Furthermore, as various software components are reused, SPL engineering increases reliability of the applications functionality and correctness.

In the next chapter, SPL engineering is introduced with special focus on methods that address the challenges in providing reconfigurable cloud applications discussed in the previous section.

1.10. Summary

This chapter classifies the scope of this thesis by explaining the context of cloud computing. Therefore, an overview of the state of the art in current cloud applications available on the market is provided. Common terminologies applied in cloud computing are introduced and conventional definitions are given. Different cloud-specific business models are discussed that are

[17] `http://www.iese.fraunhofer.de/en/events/scarves2012.html`

closely related to the cloud computing stack defined by NIST. The business model SaaS is often used to refer to cloud applications in related work. Hence, the terms SaaS application and cloud application are often used interchangeably as these applications are provided as services.

Customer orientation plays a central role in provisioning cloud applications as services. Especially enterprise customers demand for applications tailored to their business. Therefore, various configuration options of applications are discussed. However, configurable cloud applications have specific demands on separating the stakeholders involved in the provisioning process. Typical stakeholders involved in the provisioning of configurable cloud applications are identified and RBAC as a common concept for restricting access on resources in shared and distributed environments is explained. Furthermore, the characteristics of multi-tenancy as a common architectural paradigm for shared cloud applications are observed and the characteristics of configurable multi-tenant aware applications are observed.

Main characteristics of applications in the cloud are discussed and narrowed to concepts related to this thesis. Service and customer orientation, availability on demand, scalability, just-in-time provisioning, and resource sharing are identified as important properties of cloud applications relevant in this thesis. However, further important characteristics regarding security algorithms, application performance, and network access are out of scope of this thesis.

Additionally, benefits of providing self-service portals for signing up for a service are explained. An overview of deployment models for cloud applications is given to point out that especially public available applications demand for a self-service configuration portal, where customers can configure a service without further interaction of the application provider. However, various legal restrictions are discussed especially applying for self-service portals.

Further challenges in automating the configuration and provisioning of customized applications are identified and benefits on addressing the identified challenges by applying SPL engineering are highlighted. The application of SPL engineering on applications in the cloud is being explicitly observed in the next chapters.

2. Software Product Lines – Foundations and Related Work

Any customer can have a car painted any colour that he wants so long as it is black.

— *Henry Ford, 1909*

Product line engineering enables reuse in a predictable way and allows for managing variation and maintaining variants efficiently [WL99, PBvdL05]. The concepts of first building a reuse infrastructure to derive products at a large scale are transferred from the manufacturing domain to software development introducing the term Software Product Line. Hence, SPL engineering aims at creating customer-specific product variants from a common reusable set of core assets with the efficiency of mass production [CN01]. Especially in the embedded systems domain, SPL engineering is a prevailing discipline, as product line concepts known from manufacturing are homogenously applicable for reusable software and hardware assets.

Research has been conducted in the area of SPLs for over 20 years and it is practically applied in several companies [KCH$^+$90, LSR07]. For instance, the Bosch Group applies SPL engineering to create a product family of washing machines. Further companies, such as Ericsson AXE and Boing, successfully apply SPL engineering and are therefore listed on the *SPLC Hall of Fame*[1]. However, the application of SPLs is not only limited to embedded systems.

Many approaches presented at the annual International Software Product Line Conference (SPLC)[2] reveal that software-intense systems benefit from SPL engineering. In addition, SPLs are also applicable to information systems available on the Internet as a special form of software-intense systems [BFK$^+$99]. As most cloud applications are information systems based on specific cloud technology, methods from SPL engineering build a good foundation for configuration management of those applications. Hence, this chapter explains general concepts of SPL engineering with special focus on methods related to the challenges of reconfigurable cloud applications identified in the previous chapter.

2.1. Mass Customization and Product Lines

The origin of SPL engineering in the manufacturing domain is referred to as *mass customization*. *Mass customization* and *product line* are common notions used in the manufacturing domain expressing that individually adapted products are created with the efficiency of mass products [Pil06]. A definition given by Davis in 1987 defines mass customization as follows.

[1]`http://splc.net/fame.html`
[2]`http://splc.net`

Definition 2.1 (Mass customization [Dav87]). *Mass customization is the large-scale production of goods tailored to individual customers' needs.*

Hence, an assembly line to produce cars, whose components are individually chosen, is considered as a form of mass customization. Individually developed products cover unique customer requirements. However, these products are more expensive than standard products. Therefore, paradigms of mass customization and product line engineering from the manufacturing domain are transferred to the software development process to create software families and reduce production costs [WL99].

2.2. Software Product Line Engineering

In 1968, McIlroy proposed to adopt mass customization techniques to software engineering to create reusable software components [McI68]. As a consequence, component-based *software family engineering* emerged. Software family engineering assumes that there exists more commonality than variability in a family of software systems [Par76]. This variability is expressed in terms of features. *Feature* is a general notion in software engineering reflecting a stakeholder requirement and user visible functionality of a product. Various meanings of the term feature exist [CHS08, AK09]. A feature is defined as follows.

Definition 2.2 (Feature [CE00]). *A feature is a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept.*

The term Software Product Line was coined to describe a software family and is defined in terms of features as follows.

Definition 2.3 (Software product line [CN01]). *A Software Product Line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment [...] and that are developed from a common set of core assets in a prescribed way.*

The terms mass-customized software, software family, software factory and SPL are often used synonymously for individual products derivable in a mass-efficient way with a strong focus on reuse [WL99, CE00, PBvdL05]. SPL engineering is a process to create software infrastructures and architecture to derive similar products of the same domain. These products have more in common than they vary. The knowledge about variability and the explicit specification of variability is essential in SPLs. According to Bosch et al. [BFG$^+$02], variability in an SPL occurs in two dimensions. Variability in *space* corresponds to the variant space that defines all possible configuration variants described statically by a variability model. In contrast, variability in *time* refers to changes either in the variant space or in a single variant configuration over time in order to adapt to new or changed requirements. Changes in derived variant configurations are considered as *reconfiguration* in terms of a Dynamic Software Product Line (DSPL), as explained in Section 2.6. In contrast to evolutionary changes, the configuration space remains unchanged.

Changes in the variant space are evolutionary changes on core assets, such as modifying, adding or removing features from the SPL. Such evolutionary changes altering the variant space are out of scope of this work.

In SPL engineering, creating reusable assets and defining an SPL infrastructure is separated from the derivation of particular products by means of domain and application engineering, as explained in the following.

2.2.1. Domain Engineering and Application Engineering

The software family is established in terms of all derivable product configurations in *domain engineering*, whereas individual variants are derived according to a variant configuration in *application engineering* [CE00]. Domain engineering focuses on identifying and describing commonality and variability among products of the same domain. Hence, the variant space is defined in domain engineering and a reusable SPL infrastructure for deriving products is developed. The results of the domain engineering process are a constraint model describing the full product variant space and reusable software artifacts [GFd98]. In contrast, application engineering aims at deriving valid configurations that satisfy the domain constraint model. Application engineering utilizes the infrastructure to derive variants. Thus, the result of the application engineering process is a complete variant configuration and the corresponding product [PBvdL05]. Figure 2.1 depicts the relation between domain engineering and application engineering and the comprising SPL processes and their resulting artifacts as adopted from [PBvdL05].

Neighbors describes an early approach towards domain engineering on building reusable software assets of the same domain [Nei80]. The approach is implemented in a tool called *Draco*. In Draco, knowledge for constructing software is organized in various reusable domains and variability is classified in terms of generalization as domain independent, domain dependent, and application

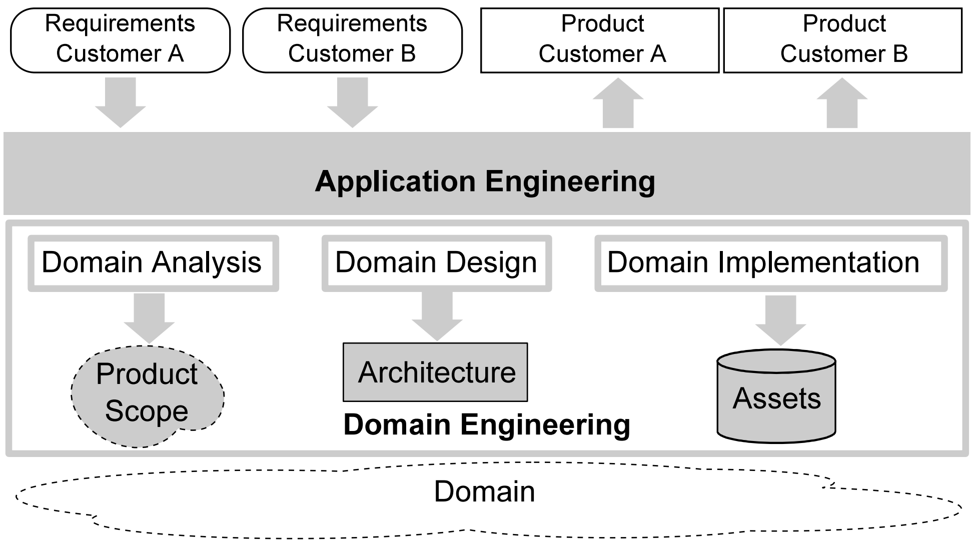


Figure 2.1 Domain engineering and application engineering.

dependent. In general, domain engineering means to design for reuse and is a process to create software artifacts, specify variability and commonality, as well as constraints valid for artifacts in the domain [CE00]. The domain engineering process contains the sub processes of domain analysis, domain design and domain implementation. *Domain analysis* comprises planning, identification and scoping of a particular domain. This activity explicitly specifies products that will be part of the product line and which are not. Furthermore, variable and common functionality of these products are defined. Variability modeling is part of the domain analysis to explicitly specify commonality and variability among product variants of the same product family. The resulting variability model defines the configuration space of the product family [WL99]. Additionally, the *domain design* process results in a domain architecture reflecting product family requirements. *Domain implementation* is a process to set up a variability infrastructure and implement variable assets.

Application engineering is the process to create a single product utilizing the SPL infrastructure and reusing domain assets. The product complies to the product family specification created during domain engineering and is developed cost-efficiently in a way similar to a supply chain.

In practice, the processes of domain engineering and application engineering cannot be separated strictly, as product enhancements are requested frequently by customers and a changeable infrastructure is essential to react on market demands. Thus, reactive and agile SPL engineering emerged to handle such changes [DPAG11], but are out of scope of this work.

2.2.2. Problem and Solution Space

Orthogonally to domain and application engineering, SPL concepts are further split up into problem and solution space [CE00]. In the *problem space*, concepts independent from a particular implementation technology are defined. Application requirements and reusable artifacts are specified. Functional and behavioral requirements on an application are described on an abstract level. Variability among functional and behavioral application requirements is modeled formally by means of a variability model. Common approaches of variability models are Orthogonal Variability Modeling (OVM) [PBvdL05], feature models [KCH$^+$90], decision models [SJ04], Common Variability Language (CVL) [HWC12], and propositional formulae [Bat05].

In the *solution space*, technology dependent parts for realizing applications are specified. For instance, these parts comprise design models, source code, and documentation. In addition, concrete implementations of reusable and variable artifacts are specified, as well as a variable reference architecture combining these software assets. A reference architecture prescribes commonalities and variabilities among all variant configurations of the product family by means of a predominant architectural style and design rules [Bos00].

A prominent approach to define a variability model in the problem space is feature modeling. The applicability of feature models to represent variability of SaaS applications is stated in literature [WZ11]. The notion of features and dependencies among features is intuitive for customers. This notion is therefore applicable in configuration self-service portals where customers configure applications themselves, as discussed in Section 1.5.3.

2.3. Feature Modeling

In the early 1990s, the Feature Oriented Domain Analysis (FODA) study introduced feature modeling as a requirements engineering method to specify an application domain and to describe variability and commonality in products of a company's product portfolio in terms of its features [KCH+90]. Ever since, feature models are widely used in SPL engineering. For instance, in generative programming [CE00], feature-oriented programming [Bat04], delta-oriented programming [SBB+10] and software factories combined with Model-Driven Software Development (MDSD) [GS03].

Depending on the application context, several extensions to the original FODA feature model language have been proposed to enhance the expressiveness of feature models [BSRC10]. In the original FODA case study, the notion *feature diagram* refers to the graphical representation of a feature model in a tree-like hierarchical structure with a single root node representing the application [KCH+90]. A schematic feature diagram in the notation introduced by Czarnecki and Eisenecker [CE00] is shown in Figure 2.2. This representation is commonly used in literature to visualize the relation between features. Therefore, a feature model is sometimes referred to as *feature tree*. In this work, the notion of *feature model* is used to comprise feature diagram and feature tree. A feature model is defined as adopted from [CE00] as follows.

Definition 2.4 (Feature Model). *A feature model defines meaningful feature combinations and represents the configuration aspect of the reusable software.*

A feature model formally describes the relations between features as a result of the domain engineering phase [LKL02]. Each node in the hierarchical structure represents a feature of the application. As an example, the *Business ByDesign* application offered by SAP, and introduced in Section 1.5.4, can be represented as a feature model. Hence, each functional unit of the application is interpreted as a feature.

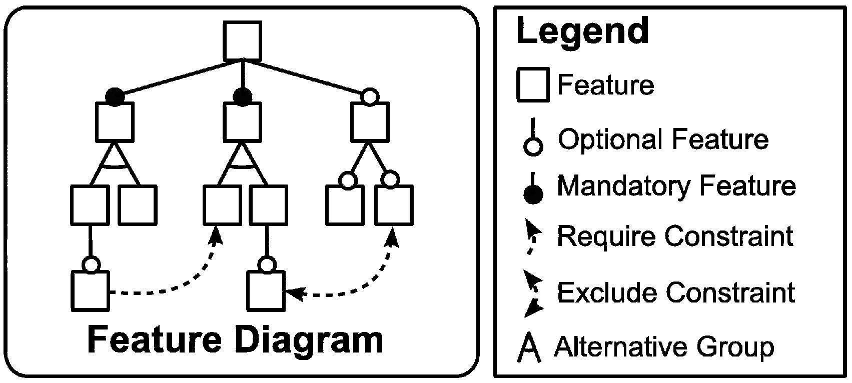


Figure 2.2 Representation of a feature model in the notation of a feature diagram (adopted from [CE00]).

2.3.1. Illustrative Example of a Feature Model

The *SAP Business ByDesign* application is chosen to illustrate the concepts of feature models. By analyzing the configuration wizard provided on the web[3], a feature model with 78 features and 23 cross-tree constraints on those features is extracted manually. These features represent variable functionality of a *Business ByDesign* application. An excerpt of the feature model containing 19 features and four cross-tree constraints is applied in the following to explain feature model concepts and depicted in Figure 2.3. The complete feature model is included in the Appendix in Section B.2.

The root feature named `Business ByDesign` represents the variable application. The relationship between child features and their parent is a decomposition relation constituting *imply* constraints. Thus, the presence of a child feature in a configuration requires the presence of its parent.

In this example, the root feature `Business ByDesign` has five direct child features, which are `Marketing`, `Purchasing`, `Supply Chain Setup Management`, `Project Management`, and `Product Development`. Additional *require* and *exclude* cross-tree constraints constitute an implication graph with further semantics [CW07]. For instance, the feature `Campaign Management` requires the feature `Market Development`, while the feature `Basic Project Planning` excludes

[3]https://www.sapconfigurator.com

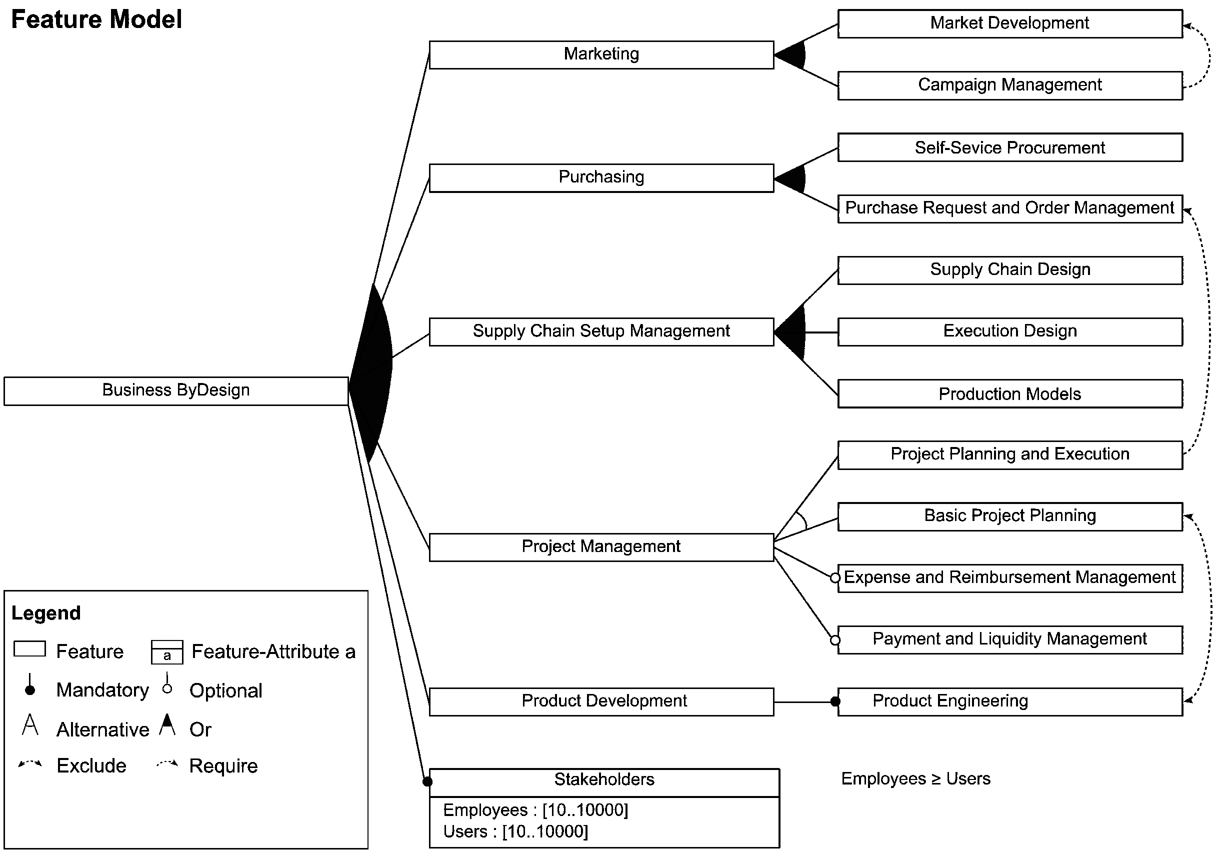


Figure 2.3 An excerpt of the feature model representing the Business ByDesign application.

the feature `Product Engineering`. Concepts that cannot be expressed by an implication graph are parent-child relations, group relations, and complex cross-tree constraints written in propositional logic.

According to the FODA definition, child features of a parent feature can either be solitary or grouped. A *solitary* relationship refers to the child's dependence on its parent feature, being either optional or mandatory. Hence, an *optional* child may be included, whereas a *mandatory* child must also be included in a configuration if a parent feature is included. In the example in Figure 2.3, the feature `Payment and Liquidity Management` is an optional feature, whereas `Product Engineering` is a mandatory feature. The *grouped* relationship is defined between multiple child features and a single parent feature. In addition to the parent child relation, feature groups express dependencies among sibling features. In the initial FODA study, only *alternative* relationships between features in a group were introduced. An alternative relationship expresses that one of the grouped features must be chosen, but not both at the same time. According to the *Business ByDesign* example, the features `Project Planning and Execution` and `Basic Project Planning` are contained in an *alternative* group.

Later on, *or* feature groups are introduced, defining that at least one feature of the group must be contained in a configuration if the parent is included [CE00]. For instance, the features `Market Development` and `Campaign Management` are contained in an *or* group. In the FODA study, attributes that further classify features, such as cost and a description, are already mentioned [KCH+90].

Industrial size feature models can contain up to thousands of features [STB+04, LP07, BSL+10] and thus contain rather complex dependencies and constraints. The described feature model concepts are not sufficient in all application areas. Features are only selectable or deselectable in an application. They cannot model enumerated application qualities and numerical configuration parameters. The modeling of numerical configuration parameters is required, to express variable SLAs of cloud applications, for instance, as explained in Section 1.5.3.

Various extensions to FODA feature models have been proposed in literature to address the reuirements of the different application domains. In the following, extensions relevant for expressing variability knowledge of cloud applications are explained.

2.3.2. Feature Model Extensions

Several extensions to the described feature model concepts exist [BSRC10]. To model configuration knowledge of cloud applications, extending feature models with group cardinality, attributes with discrete domains, and complex constraints, is promising.

Group-cardinality

To generalize the relationship of grouped and solitary features without distinguishing between alternative and or feature groups, Riebisch introduced the concept of *group cardinality* inspired

Table 2.1 Unified modeling of feature decomposition relations applying group cardinality (adopted from [RBSP02]).

Decomposition Relation	Group Cardinality	Number of Features in Group
Optional Feature	0..1	1
Mandatory Feature	1..1	1
Or Group	1..n	n
Alternative Group	1..1	n
Optional Group	0..n	n
Mandatory Group	n..n	n

by Unified Modeling Language (UML) multiplicities [RBSP02]. The group cardinality is specified by its multiplicity comprising a lower and an upper bound defining how many elements of the group must be at least and at most contained in a variant configuration. Table 2.1 depicts how the decomposition relations *optional*, *mandatory*, *or*, and *alternative* are expressed using group-cardinality. The optional solitary feature relation is a special case of an optional group relation, where the same holds for the mandatory relation, accordingly. Group cardinality have the advantage that solitary and grouped features are modeled in the same way. Hence, formalism and tools do not need to explicitly cope with special cases as solitary and grouped features are unified.

Attributes over Finite Domains

Attributes are assumed as scalar variables with a particular data type equal to a discrete or continuous domain [SRP03, BTRC05]. To model the non-functional configurability of SaaS applications, features are enhanced by sets of attributes. Different notions of attributes are proposed. In general, an *attribute* is a property with a name and a value further characterizing a feature [CBUE02, BTRC05]. A predefined domain is assigned to an attribute specifying potential attribute values and various attributes share the same domain. For instance, a feature `User Interface` has an attribute `Background Color` with the domain values `light blue, dark blue, and green` of the domain `Color`.

The cost of a feature is another attribute relevant for features used in pricing of SaaS applications. Furthermore, quality attributes, e.g., reliability and availability, in SaaS applications are associated with SLAs, as explained in Section 1.5.3. The values of such attributes are classifiable by enumerations when using symbolic values. For instance, availability may be classified in `low, medium, high, very high`. An enumeration allows for effective analysis of constraints on attribute values [MS98].

Domains are distinguishable in discrete and continuous domains. A *discrete domain* may be represented by integer numbers, whereas a *continuous domain* is represented by real numbers. In the *Business ByDesign* example in Figure 2.3, the mandatory feature `Stakeholders` contains two attributes `Employees` and `Users`. Both attributes can assign a value between 10 and 10000.

In the figure, a notation for attributes is used similar to the notation introduced by Benavides [BTRC05]. Features with attributes are represented similar to classes in UML class diagrams. In this notation, the rectangle used to represent a feature is split to contain an area for listing attributes and their domain.

Feature models comprising attributes over discrete domains are reducible to feature models without attributes. For instance, the attribute `User` in the *Business ByDesign* example can be transformed into a feature with an *alternative* group comprising all potential domain values represented as child features. Hence, tools and formalism applicable on feature models without attributes are reusable for such attributed feature models. In contrast, attributes over continuous domains are more expressive. Thus, feature models with attributes over continuous domains are not reducible to feature models only comprising features. Additionally, a feature model contains further complex cross-tree constraints among features, attributes, and their domain values. As constraints on infinite continuous domains are in general undecidable, only attributes with finite domains are considered in this work.

Complex Feature Model Constraints

A feature model constrains the relations between features, attributes and domain values. In general, constraints describe complex combinatorial problems declaratively. A constraint is defined as follows as adopted from [Coh90].

Definition 2.5 (Constraint [Coh90]). *A constraint declaratively specifies a relation between logical variables of different domains.*

As such, a constraint defines allowed value combinations for these variables. Feature model constraints typically describe static dependencies between features, attributes and domain values. A logical expression is referred to as constraint. An expression is unary, binary, and n-ary, depending on the number of contained variables [RN09]. A *unary* expression restricts the value of a single variable, e.g., $x > 2$. A *binary* expression defines a dependency between the values of two variables, e.g., $x \neq y$. A *n-ary* expression defines dependencies between three or more variables, e.g., $\max(a_1, a_2, \ldots, a_k)$. The combination of such expressions forms complex constraints. For instance, complex constraints on features and attributes based on Object Constraint Language (OCL) have been introduced in [SRP03]. Various types of constraints exist regarding their purpose and application context [Coh90]. Some examples of constraint types are shown in the following.

- propositional logic, e.g., $\vee, \wedge, \rightarrow, /$
- relational operators, e.g., equality $=$ and inequalities $\neq, \leq, \geq, >, <$

- arithmetic constraints define a relation R between two expressions e in the form $e_1 R e_2$, where $R \in \{=, \neq, \leq, \geq, >, <\}$ and at least one expression contains operators $o \in \{+, -, *, /\}$
- conditional, e.g., if e_1 then e_2 else e_3
- combinatorial, e.g., min, max, sum

The discussed constraints are *absolute* constraints defining that a violation of one constraint prohibits a valid solution. If it is important for some reason to not only find a valid, but an optimal solution, *preference* constraints are used to specify preferred solutions. Preference constraints are encoded in cost functions, which are applied to find an optimal solution regarding costs, but are out of scope of this work.

In general, a feature model defines hierarchical constraints, group constraints and cross-tree constraints, which can be classified into the discussed constraint types. Considering the tree structure of the feature model, hierarchical constraints are implies relations between child and parent features. Group constraints are assumable as arithmetic constraints. Cross-tree constraints comprise the binary *require* and *exclude* relations between features as originally proposed in FODA and further complex attribute constraints.

In the *Business By Design* example a require constraint is defined between the feature `Campaign Management` and `Market Development` indicated by the directed dashed line in Figure 2.3. A further exclude constraint is defined between the features `Basic Project Planning` and `Product Engineering`. As attribute constraints unary and binary relational constraints between attributes and domain values are to be considered. In a unary constraint, an attribute a is put in relation with a literal value $v \in D$ contained in the attribute domain D such as $a \,\mathrm{rel}\, v$ with rel $\in \{=, \neq, \leq, \geq, >, <\}$ for numerical Integer domains and rel $\in \{=, \neq\}$ otherwise. For instance, be an attribute *color* defined over the domain $LightColors = \{yellow, orange, white\}$. A unary constraint could be defined as *color* $\neq$ *yellow* specifying that the attribute should not be yellow. An example for a constraint on an attribute *memberCount* defined over the Integer domain $Members = \{1, \ldots, 100\}$ can be specified as *memberCount* > 50. Additionally, binary constraints are defined similarly except that, instead of a literal value, a second attribute is inserted in the constraint and the domains of both attributes are comparable. For instance, the constraint *color* $\neq$ *background* expresses that attribute *color* should not have the same value as attribute *background*. In the *Business ByDesign* example, a relational cross tree constraint is defined on the `User` and `Employee` attributes. This constraint ensures that the number of employees must be greater or equal to the number of users.

A feature model describes the variability of products of an application domain in terms of constraints among features. The specification of a variability model in the domain engineering phase and the derivation of configurations in the application engineering phase is distinguished, according to the definition in Section 2.2.1.

2.4. Derivation of Variant Configurations

Based on the constraints specified by a feature model, valid variant configurations are derivable in a specialization process. Specialization is defined as follows.

Definition 2.6 (Specialization [CHE04]). *Specialization is a transformation process with the goal to bind variability specified by a feature model. Input and output of the transformation is a feature model, such that the set of the configurations denoted by the resulting feature model is a subset of the configurations denoted by the former feature model.*

A stakeholder specifies a variant configuration in a specialization process by selecting and deselecting features from the feature model obeying variability constraints [CE00]. If the feature model includes attributes, then assigning attribute values is part of the process as well. The result of a specialization process is a complete configuration where all variability is bound. The feature model of the *Business ByDesign* example shown in Figure 2.3 defines various complete configurations as explained in the following.

2.4.1. Illustrative Example of a Variant Configuration

A specialization process leads to a valid complete variant configuration. Based on the *Business ByDesign* feature model explained in Section 2.3.1, a possible complete variant configuration is depicted in Figure 2.4. Red crosses and light gray feature names mark deselected features, while check marks highlight selected features in the complete configuration. Hence, in the example complete configuration, 9 features are deselected and 10 features are selected. Selected features are highlighted by a check mark in the figure, while deselected features are greyed out and marked with a cross. Furthermore, the two attributes have assigned values.

The exemplified complete configuration comprises the root feature `Business ByDesign`, as well as the further selected `Marketing` feature together with its child features `Market Development` and `Campaign Management`. The root feature generally represents the application itself and must always be selected in a valid complete configuration. Additionally, the `Supply Chain Setup Management` feature and its child feature `Supply Chain Design` are included in the variant configuration. The feature `Project Management` and its child features `Basic Project Planning` and `Payment and Liquidity Management` are further included together with the attributed feature `Stakeholders`. Furthermore, the two attributes of the feature `Stakeholders` are set. In this example, the value of the `Employees` attribute is 100, and the value of the `Users` attribute is 15. The functionality represented by the selected features is available in the tailored application instance.

In contrast, the deselected features are grayed out in the figure and are not available in a derived application configuration. Deselected features are `Purchasing` and its child features `Self-Service Procurement` and `Purchase Request and Order Management`, as well as `Execution Design`, `Production Models`, `Project Planning and Executions`, `Expense and`

`Reimbursement Management`, and the feature `Product Development` together with its child feature `Product Engineering`.

A complete configuration can be derived in a single step or in multiple steps. The incremental process of deriving variant configurations in multiple steps is operationally described by means of *staged configuration* [CHE04, CHH09, RSPA11a]. Stages can be associated with different phases of a configuration process and with different stakeholders.

2.4.2. Staged Configuration

The key concept of *staged configuration* is to derive a complete variant configuration in multiple ordered specialization steps. Hence, the chronology of configuration decisions is stipulated. Each stage in a staged configuration process consists of a set of specialization steps, where in each specialization step a single configuration decision is conducted. Staged configuration is defined as follows.

Definition 2.7 (Staged Configuration [CHE05b]). *Staged configuration is a step-wise specialization process to transform a feature model until a complete variant configuration is reached.*

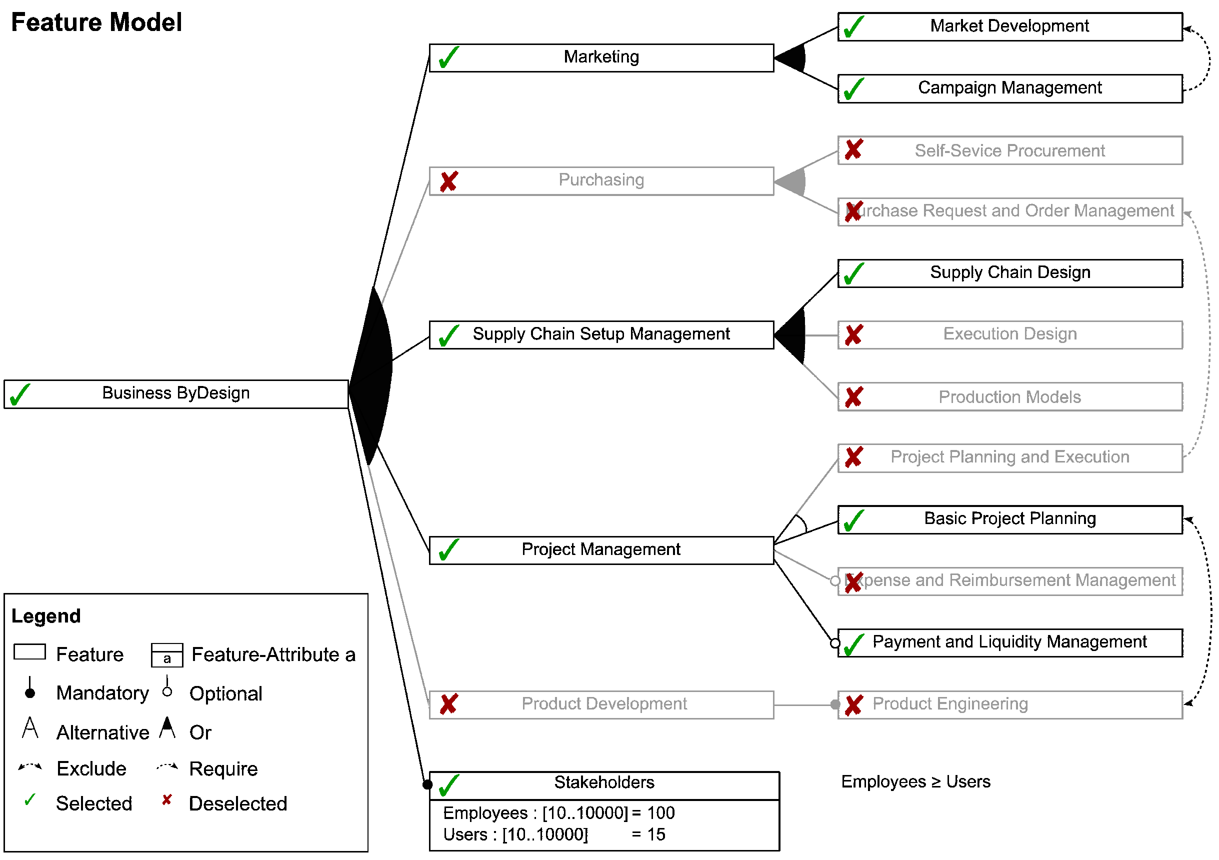


Figure 2.4 A complete configuration of the Business ByDesign feature model excerpt.

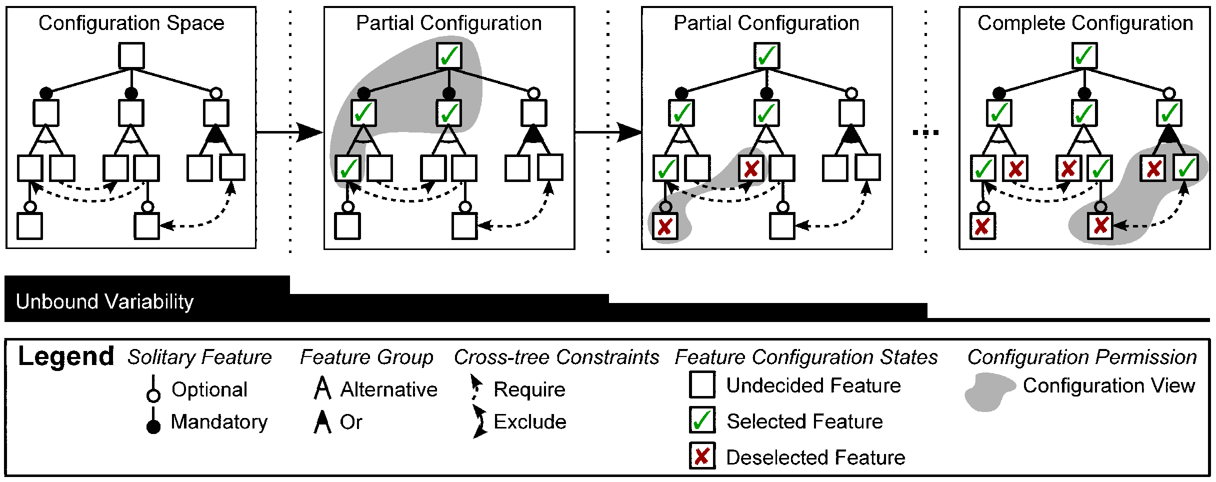


Figure 2.5 Configuration decisions made in a staged configuration process step-wise decrease the amount of unbound variability.

Figure 2.5 illustrates a staged configuration process. The feature model on the left side is the input for the staged configuration process and describes the complete configuration space. This feature model, where no variability is bound, is input for the first stage. The output of each stage is a transformed feature model with reduced variability. Each transformed feature model must obey feature model constraints and is input for the subsequent stage. In each stage, configuration decisions are conducted in multiple specialization steps. Allowed configuration decisions of a stage are specified in a *configuration view.*

A configuration view shows a stakeholder-specific part of the feature model relevant for conducting configuration decisions. In the figure, configuration views are highlighted with a gray area around configurable features. By conducting configuration decisions, the degree of the unbound variability decreases, until in the final stage n, a complete configuration is reached where all variability is bound. Configuration decisions conducted in a stage cannot be reverted in subsequent stages. The amount of variability bound per stage depends on the assigned views and differs therefore. The stages 1 and 2 that are not final as they result in *partial configurations*, where variability is still left.

According to the *Business ByDesign* example in Figure 2.3, different views can be defined comprising the configuration operations of differing features. Figure 2.6 shows two sample views for two different stakeholders conducting configuration operations.

One view is entitled `view 1`, where belonging features are highlighted with a dark gray background, while features belonging to the second view entitled `view 2` are highlighted with a light gray background. Hence, a resulting staged configuration process comprises two ordered stages, with a view assigned to each stage. Assuming that configuration operations are conducted first in `view 1` followed by operations in `view 2`, the result of the first stage is a partial configuration with variability left, as exemplified in Figure 2.6, while the result of the second stage is a complete configuration similar to the configuration in Figure 2.4. Staged configuration processes are modeled and automated by workflows.

Staged Configuration Processes and Workflows

Staged configuration is formally modeled as process and automated by a workflow [MCdO07, Hub12]. As such, a process is defined as follows.

Definition 2.8 (Process [WfMC99]). *A process is a formalized view of a business process, represented as a coordinated (parallel and/or serial) set of process activities that are connected in order to achieve a common goal.*

Workflow modeling aims at automating the execution of formally defined business processes and is defined as follows.

Definition 2.9 (Workflow [WfMC99]). *A workflow is the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules.*

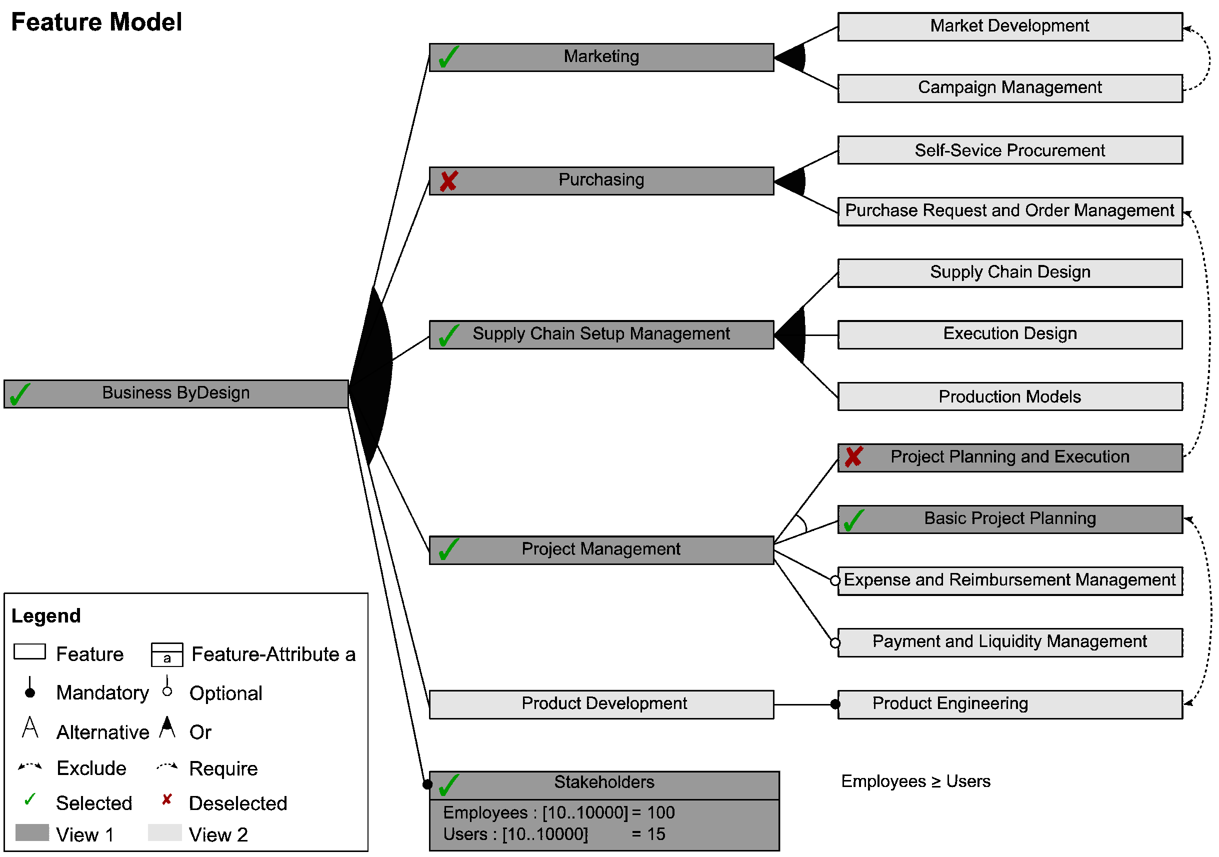


Figure 2.6 Two views on the Business ByDesign feature model applied in a staged configuration process.

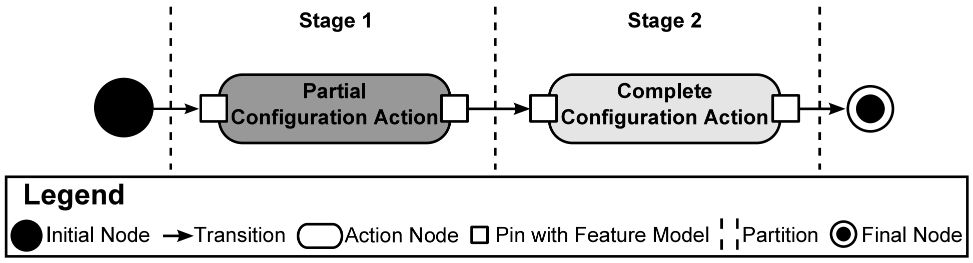


Figure 2.7 An example staged configuration workflow comprising two stages in UML activity diagram notation.

In literature, staged configuration processes based on extended feature models are formalized and automated as workflows [CHE05b, MCdO07, KOD10, MSDLM11, HCH09, AHH11]. Different languages exist to define a workflow, such as Business Process Model and Notation (BPMN)[4] and UML activity diagrams [DtH01]. A benefit of UML activity diagrams is that they provide a graphical notation for specifying workflows, as well as execution semantics [Fah08].

Figure 2.7 depicts a UML activity representing the sample staged configuration workflow for the *Business ByDesign* example explained before. The views depicted in Figure 2.6 can be mapped to stages. The view *view* 1 is dedicated to `stage 1`, while `view 2` is dedicated to `stage 2`, highlighted by the same background color. Hence, the staged configuration workflow comprises two stages, separated by swim lanes. Each stage contains a *configuration action* that applies the associated view for conducting configuration operations. The purpose of each action is to reduce the variability of the feature model. The feature model is input and output of these actions, as depicted by *action pins*. The result of `stage 1` is a partial configuration whereas the result of `stage 2` must be a complete configuration where all variability is bound.

By applying feature models in an SPL-based configuration process, the satisfiability of feature model constraints is to be guaranteed. A feature model is satisfiable if at least one valid variant configuration can be derived. Satisfiability of a feature model can be checked by applying constraint programming, as discussed in the next section.

2.5. Feature Model Satisfiability

Regarding the concepts of a feature model, constraint programming methods are applicable to analyze the satisfiability of a feature model. *Constraint programming* describes complex combinatorial problems declaratively and is applicable to check the satisfiability of a feature model [BRCT05]. In a constraint program, relations between variables are stated as constraints and evaluated by a solver. Constraint programs find assignments of variables that fulfill the specified constraints [RN09]. The satisfiability of a feature model is analyzed by translating feature model concepts into adequate logic representations. Depending on the type of vari-

[4]`http://www.omg.org/spec/BPMN/2.0/`

ables used to express the feature model constraints, different solvers are used to check the satisfiability [SDM+11].

Typical constraint program representations of relations in a feature model are Programmation en Logique (Prolog), Binary Decision Diagram (BDD), Constraint Satisfaction Problem (CSP), and Satisfiability Modulo Theories (SMT). Applying the respective solvers to the constraint program verifies the satisfiability of the translated feature model. Basic feature models containing features and cross-tree constraints expressed in propositional logic are represented as a set of Boolean variables and associated constraints [Man02, Bat05]. For such feature models with Boolean domains, Satisfiability (SAT) and BDD solvers are applicable, as well as Prolog [MWC09, BTRC05, ZYZJ08, Seg08, Bat05].

A common approach to checking the satisfiability of feature models with discrete attribute domains is by translating the feature model into a CSP and using CSP solvers to reason on the model [BRCT05, WDS09, MSDLM11, KOD10]. Additionally, feature models with attributes over infinite discrete and continuous domains are checked by applying SMT solver [XHSC12].

For feature models containing attributes over finite domains, as applied in this work, a CSP solver is suitable. Therefore, the following section explains in detail how to apply a CSP solver to check the satisfiability of extended feature models. The application of other solvers is out of scope of this work.

A CSP defines a set of variables and a set of constraints, where each variable has a domain containing its possible values [Tsa93]. A CSP is solved by identifying a consistent assignment of variables that satisfies all constraints. Checking the satisfiability of a feature model is a CSP, where a valid variable assignment equals a valid variant configuration of the feature model. If at least one valid variable assignment exists, the feature model is satisfiable.

A basic feature model containing binary imply and exclude cross-tree constraints is a Boolean CSP [BRCT05]. When transforming an attributed feature model into a CSP, attributes are considered as variables with finite domains and associated with a certain feature. The worst case runtime complexity of solving finite-domain CSPs is exponential with respect to the number of variables. In most cases, CSP solvers are much faster as they comprise special search algorithms. Thus, CSP solvers scale for large feature models as shown in literature [MSD+12].

Applying a CSP solver allows reasoning on feature models. For instance, the number of derivable variant configurations can be computed, as well as core features included in all variant configurations and dead features contained in none of the derivable variant configurations [TBD+08]. A CSP solver checks a given partial configuration and determines whether feature model constraints remain satisfiable. Based on a conflict in a partial configuration, a CSP is applied to suggest error resolving sets of features that need to be selected or deselected to solve the conflict [WSB+08].

2.6. Dynamic Software Product Lines

A Dynamic Software Product Line (DSPL) is a special kind of SPL, where the configuration space describes the states of an adaptive reconfigurable application [HHPS08]. In contrast to statically configured applications of an SPL, the configuration of a deployed application is changed dynamically during application runtime in a DSPL. At a point in time, a complete configuration represents the current state of the application. If requirements change during application runtime, the deployed application is *reconfigured*. In a DSPL reconfiguration is defined as follows.

Definition 2.10 (Reconfiguration [GH04]). *Reconfiguration in a DSPL is the process of switching between configurations of an already deployed application during runtime caused by changed requirements.*

In DSPLs several processes that transform a feature model are to be distinguished. Specialization processes occur in SPLs, as explained in Section 2.4. Additionally, in DSPLs further reconfiguration and generalization processes occur. Figure 2.8 classifies these processes. The *feature model* on the left side defines the *configuration space* by constraining feature combinations. Based on this model, *product variants* are derived in a *specialization* process, as explained in Section 2.4. A *complete configuration* as visualized on the right side satisfies all feature model constraints and represents a valid product variant of the SPL. The specialization process may be conducted in a single step, where a complete configuration is directly derived from the feature model, and in multiple steps forming a staged process with intermediate configurations.

Various staged configuration approaches propose to derive a product in multiple steps, as discussed in Section 2.4.2. The result of each stage is a *partial configuration* that does not contradict feature model constraints and includes still unbound variability. Each subsequent stage further specializes the partial configuration until all variability is bound in the final stage resulting in a

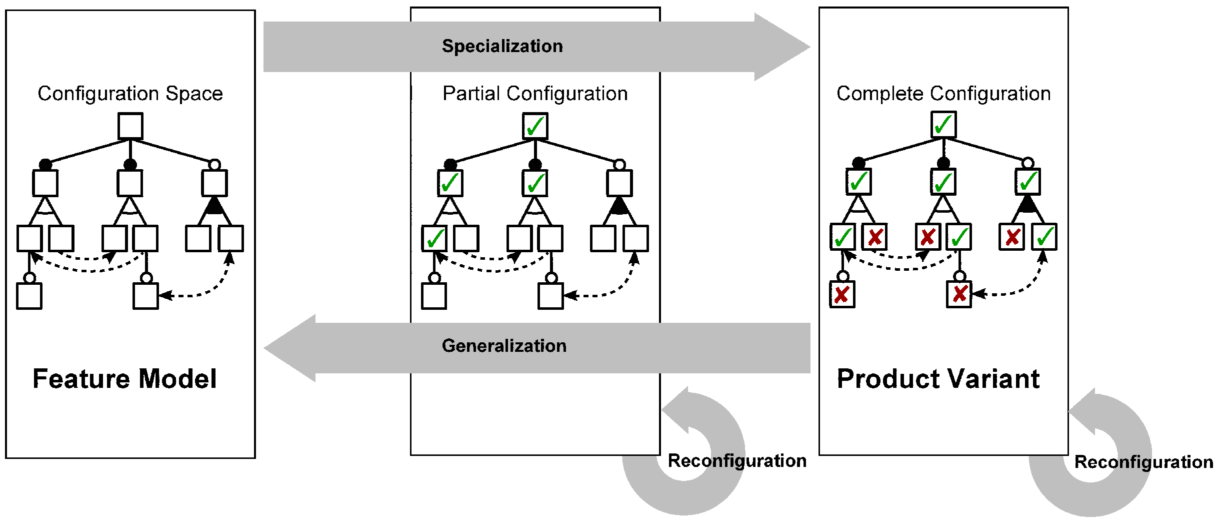


Figure 2.8 Classification of feature model transformation processes.

complete configuration. In contrast, the inverse process of specialization is *generalization* and is specified as follows.

Definition 2.11 (Generalization). *Generalization is a transformational process with the goal to release bound variability specified by a feature model. Input and output of the transformation is a feature model, such that the set of the configurations denoted by the resulting feature model is a superset of the configurations denoted by the former feature model.*

In a generalization process bound variability is released and thereby the number of derivable variants is increased. Generalization occurs in DSPLs as previous configuration decisions are revoked. Additionally, *reconfiguration* occurs in complete variant configurations by switching from one configuration to another or in partial configurations where changes are propagated to subsequent stages. Reconfigurations of complete variant configurations is a central concept in DSPLs, as explained before.

In general, DSPL approaches are applied in various areas where reconfiguration of an application is required during runtime, such as self-adaptive systems and ubiquitous computing [BSBG08, LK06]. As reported recently, cloud applications are a further application area for DSPLs [BGP12].

2.6.1. Reconfigurable Cloud Applications are Dynamic Software Product Lines

Software information systems also benefit from SPL and DSPL engineering in terms of an efficient variability management [BFK$^+$99]. Most applications provided in the cloud are information systems. A configurable cloud application can be regarded as a product family in SPL engineering where each customer configuration is presumed as a product variant. The customization of a cloud application is assumed as a DSPL as requirements of customers may change and thus, their tenant configurations are to be reconfigured accordingly during application runtime. Thus, the domain and application engineering phases of an SPL as visualized in Figure 2.1 are adopted to the configuration of cloud applications as shown in Figure 2.9. Domain engineering in an SPL equals a *SaaS engineering* phase, where the configuration options and the shared artifacts of a cloud application are specified. The configuration options of the SaaS application in the cloud equal the variability in a product portfolio. Therefore, the *SaaS analysis* phase is performed to define the configuration scope of the application in terms of features. Furthermore, in the *SaaS design* phase, the application architecture is defined and the multi-tenant aware shared infrastructure is specified. Commonly, a configurable SaaS application will be deployed in a public or community cloud. In the *SaaS implementation* phase, shared software and hardware assets are specified and implemented. Additionally, in the implementation phase, shared assets are implemented.

The application engineering process of an SPL, in which a particular application is derived per customer, equals the *SaaS customization* process of a multi-tenant aware cloud application. In the SaaS customization process the requirements of a customer are evaluated and a suitable tenant configuration is created. Such a tenant configuration is not an application on its own, but integrated into the multi-tenant aware single-instance SaaS application.

2.6.2. Cloud Applications and Software Product Lines

As discussed previously reconfigurable cloud applications can be perceived as a specific type of SPL. However, their concepts differ. Aligning different types of cloud applications and SPLs leads to the comparison in Table 2.2. The concepts of SPLs, DSPLs, and cloud applications are classified in terms of the number of customers, related variant configurations and the number of deployed product instances. In an SPL, each customer has one configuration and a corresponding product instance. A configurable multi-instance SaaS application equals an SPL, as per customer one configuration is derived and a corresponding product instance is deployed per

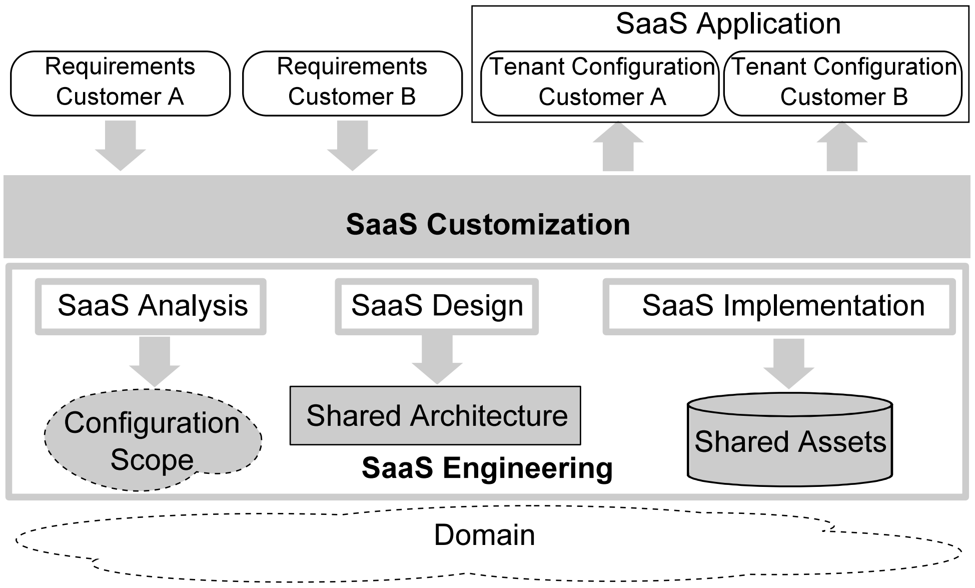


Figure 2.9 Applying processes from Software Product Line engineering on the configuration of a cloud application.

Table 2.2 Classification of cloud and SPL concepts regarding customer configuration and product instance multiplicities.

Category	Customer	Configuration	Product instance
Software product line	1	1	1
Configurable multi-instance SaaS application	1	1	1
Multi-tenant aware SaaS application	m	1	1 (shared by all customers)
Dynamic software product line	1	n	1
Reconfigurable multi-instance SaaS application	1	n	1
Reconfigurable multi-tenant aware SaaS application	m	n	1 (shared by all customers)

customer. However, a multi-tenant aware SaaS application differs, because multiple customers share the same configuration and the same product instance. In a DSPL, each customer has multiple configurations of the same product instance, where one configuration is active at a time. Additionally, a reconfigurable multi-instance SaaS application equals a DSPL, as per customer multiple configurations are derived, but only one at a time is active in the related product instance. Moreover, in a reconfigurable multi-tenant aware SaaS application, multiple customers have a configuration at a time that is integrated into a shared single product instance. Similar to a DSPL, configurations change according to changing customer requirements.

Changes in a variant configuration are a potential source of error. To prevent errors in configurations by managing and explicitly tracing configuration information, the discipline of *configuration management* emerged. Generally, configuration management copes with configurability and changes in configurations of a product or a system.

2.7. Configuration Management in Software Product Lines

Configuration management is a discipline dealing with changes and evolution of products and systems. The system engineering process defines the demand for tracing configuration information of configurable items in a system and the ability to guarantee consistency of a product's properties with respect to product requirements and is defined by ANSI and Electronic Industries Alliance (EIA) in the standard ANSI/EIA-649-B as follows [EIA-649-B].

Definition 2.12 (Configuration Management [EIA-649-B]). *Configuration Management is a technical and management process applying appropriate resources, processes, and tools to establish and maintain consistency between the product requirements, the product, and associated variant configuration information.*

In other words, configuration management aims at planning, controlling and documenting changes to assure the quality of products. According to the ANSI/EIA-649-B standard main activities of this process are

- *configuration management planning* for planning (i) what configuration items to control, (ii) when to control a variant configuration, and (iii) how to change a variant configuration,
- *configuration identification* for specifying controlled functional and physical configuration properties,
- *configuration control* monitoring what changes are made to a configuration when and how,
- *configuration status accounting* determining and reporting the configuration status of any configuration item, and
- *configuration verification and audit* guaranteeing that configuration items are configured correctly and the complete configuration is consistent with respect to the specification.

Generally, a variant configuration is allowed to change after reaching a baseline. A baseline is a product version that reaches major product requirements. The intention of configuration management is to keep variable system artifacts manageable throughout their lifecycle, starting with their design, implementation, test, build, release and finally maintenance. Hence, this process is applicable to various domains ranging from industrial engineering to software engineering.

In software engineering, Software Configuration Management (SCM) is a discipline of planning and managing evolutionary changes of a software system and its parts [IEEE 828-2012]. SCM deals with dependencies and relations among variable software parts and defines the *variant space* to describe variation in space, while further coping with changes of a product in terms of product versions considering variation in time by means of a *version space* [CW98]. Hence, SCM has a strong focus on the implementation phase and is therefore often equated with collaborative version control of source code artifacts [Bab86]. However, version control and traceability of source code artifact changes over a period of time are not considered in this thesis.

An important aspect of configuration management is the management of variation in time to maintain the integrity of configuration changes in a variable product. Hence, configuration management is an important multi-dimensional management process in SPL engineering coping with variations in time and space [NCB+13].

In SPL engineering, the variant space is modeled explicitly, as explained in Section 2.2.1, and thus a part of SCM. SCM in SPL engineering does not only manage the variation of a single product, but with the variation of all derivable products and therefore shared artifacts [YR06, Tha12]. In addition, configuration options of shared core product line assets as part of the domain engineering are managed, and on the other hand the compliance of derived products with respect to specifications is managed in the application engineering.

In SPL engineering, a product changes in time, either caused by evolutionary changes of core assets implying the variant space to change accordingly or by reconfiguration in a DSPL leaving the variant space constant.

This thesis assumes a cloud application as a DSPL without changing variant space, as explained in Section 2.6.1. As discussed in Section 1.9, the application of SPL methods on cloud applications is beneficial to cope with variation in application functionality.

Main configuration management activities listed above can be automated or at least supported by automation for cloud applications by applying SPL methods as further explained in the next chapters. For instance, functional variation of a cloud application is described in a unified feature model, as explained in Section 2.3 in the problem space. However, open research questions in applying SPL methods on cloud applications remain. In the following, requirements for applying methods from SPL engineering for automating configuration management of cloud applications are summarized.

2.8. Requirements for Automated Configuration Management of Cloud Applications

As explained in Section 1.5.1 cloud applications demand for functional scalability with respect to functionality and quality. Hence, cloud applications must be reconfigurable. They demand for automated configuration management to ensure the correctness of configurations and reconfigurations, as well as to omit manual interaction in providing applications. Hence, providing self-service portals enables various stakeholders to conduct configuration operations on their own. The result of the configuration decisions can the be available just-in-time by automating configuration management, as motivated in Section 1.8.

Methods from SPL engineering combined with a multi-tenant application architecture are applicable to automate configuration management. However, different requirements for a concise configuration management approach applying such methods are identified in the previous chapters and summarized in the following.

Requirement 1 (Flexible cloud application architecture). *Reconfigurable cloud applications that are scalable in terms of functionality and quality require a flexible software architecture that supports reconfiguration at runtime, as well as just-in-time provisioning of tailored applications.*

A flexible software architecture supports variability in two dimensions, (i) among space with respect to customer configurations, and (ii) among time regarding changes in each single customer configuration. Furthermore, mechanisms are to be provided to restrict the user access and gain access to different functionalities and resources according to the specified requirements. In addition, customer configurations may change and trigger adaptation of the application instance at runtime.

Feature models are suitable to express the parameterized variability of cloud applications as identified in Section 1.8. Quality constraints, such as performance, and the server location can be expressed by attributes and related value domains. However, a unified model describing variability and meaningful configuration variations may be rather complex and extensive for a SaaS application. For instance, the feature model of the *Business ByDesign* application, depicted in Figure 2.3, contains 78 features and 23 cross-tree constraints. The feature model restricts configuration operations by introducing further, not always obvious, dependencies between features.

Requirement 2 (Consistent configuration space tailoring). *Prior to conducting particular manual configuration operations, a consistent and automated tailoring of the configuration space is required.*

Business reasons, such as selling features in packages induce the feature model to be meaningfully partitioned while respecting feature dependencies. Furthermore, due to legal restrictions, as discussed in Section 1.5.3, a large amount of functionality may not be available in a certain country. Hence, the configuration space must be tailored efficiently by obeying feature model constraints prior to conducting manual configuration decisions.

Requirement 3 (Adaptive staged reconfiguration processes). *Various different stakeholders are to be flexibly integrated in the configuration of SaaS applications, where the configuration options of stakeholders are restricted properly. Configurations conducted by specific stakeholders are reusable as pre-configurations for further stakeholders, while reconfiguration decisions of partial configurations are propagated to depending stages.*

Varying stakeholders are involved in configuring SaaS applications with differing configuration permissions, as discussed in Section 1.8. However, in an open cloud-environment, not all stakeholders are known beforehand. For instance, how many customers and which customers will subscribe for a SaaS application cannot be predicted. Thus, the configuration process must be flexibly modifiable and scalable to handle stakeholders dynamically. Stakeholders, such as providers, create pre-configurations independently reused by several other stakeholders, such as tenants. Therefore, a mechanism to consistently create and handle such pre-configurations in the configuration process is required. In general, stakeholders are only allowed to conduct configuration decisions that lead to valid variant configurations. Thus, the configuration process must not lead to deadlocks.

Configuration decisions conducted during the configuration process are evaluated to ensure the derivation of valid complete configurations. As stakeholder objectives may change, for instance, if a tenant decides to rent different functionality, the tenant's partial configuration is changed and subsequent configurations, such as user configurations are reconfigured. Hence, a scalable configuration and reconfiguration process that adapts automatically to the stakeholders participating in the configuration process of an application is required. Beside the identified requirements, further demands covering a scalable and elastic cloud infrastructure, the implementation of the user interface of a self-service portal, as well as a multi-tenant aware persistence service, and network communication. These aspects are important for the provisioning of enterprise applications, but are out of scope of this work.

2.9. Summary

This chapter explains fundamental concepts of SPL engineering and mass customization and provides an overview of their applicability to cloud computing. The common SPL engineering phases of domain and application engineering are described and how they are transferable to cloud applications. From the variety of SPL approaches proposed in literature, selected concepts related to the configuration of cloud applications are presented. An overview of variability modeling technique applied in SPL engineering is given. Especially, feature modeling as a common technique is discussed. To model the customizability of applications in the cloud, particular extensions to the initially proposed FODA feature models are explained. These extensions comprise group-cardinality and attributes with finite domains.

Moreover, the derivation of variant configurations based on feature models is presented and how the variant configuration can be conducted by multiple participating stakeholders in a staged configuration process. Each stage results in a partial configuration of the feature model, which needs to be checked for satisfying feature model constraints. Different methods to ensure feature model satisfiability are discussed, while for extended feature models with attributes, applying a CSP solver is promising. Thus, the translation of extended feature models into CSP is explained. Furthermore, relations between applications in the cloud and DSPLs are established. A central concept of DSPLs is reconfiguration. A DSPL in general expresses the states of an application and reconfiguration is applied to switch from one state to another. This is similar in an SPL for configurable multi-tenant aware applications Moreover, the SPL describes all possible configurations of all tenants, as well as all possible reconfigurations of a single tenant.

This chapter further introduces the discipline of configuration management focussing on the integrity of configuration changes by planning, controlling, and documenting configuration changes. Thus, configuration management is an important process in SPL engineering. However, open research questions in applying methods from SPL engineering for automating the configuration management of reconfigurable cloud applications remain. These questions lead to the requirements identified in Section 2.8. Hence, a configuration management framework for cloud applications is presented in the following chapters to address the requirements.

Part II.

Configuration Management Based on Feature Models

3. Configuration Management Framework for Reconfigurable Cloud Applications

The hardest single part of building a software system is deciding precisely what to build.

— *Frederic Brooks, 1987*

Reconfigurable cloud applications require configuration management to cope with variation. The previous chapter revealed the applicability of SPL methods to model variation and to prevent errors in automating configuration and reconfiguration processes. For instance, *feature models* concisely specify variability and commonality among application configurations, as shown by the example in Section 2.3.1. In addition, *staged configuration* concepts are appropriate to structure configuration processes and derive valid application configurations accordingly, as exemplified in Section 2.4.1.

However, the requirements identified in Section 2.8 regarding a variable application architecture, a concise definition and reuse of pre-configurations, as well as adaptivity of configuration processes are not yet addressed in SPL engineering. This chapter provides an overview of a feature-based configuration management framework proposed in this thesis that extends SPL methods to address the particular requirements of reconfigurable cloud applications.

The next section introduces a video information system as an example for a reconfigurable cloud application to illustrate the concepts.

3.1. Example of a Video Information System

An example of a component-based video information system is introduced in this section to explain the concepts of the proposed framework for reconfigurable cloud applications. In a hypothetic cloud scenario, a customizable and video portal depicted in Figure 3.1 is offered by an application provider to various customers. The system can be rented and tailored by customers to provide video content to several users. The video portal is multi-tenant aware to serve multiple customers by a single application instance to ease maintenance. A user of the application can access a tailored application to stream videos.

In this scenario, the application comprises several component types visualized in a graphical notation adopted from UML component diagrams. Some component types are mandatory for a video portal, while other types are optional highlighted by different background colors of the component types in the figure. Customers choose different subsets of these types. Mandatory

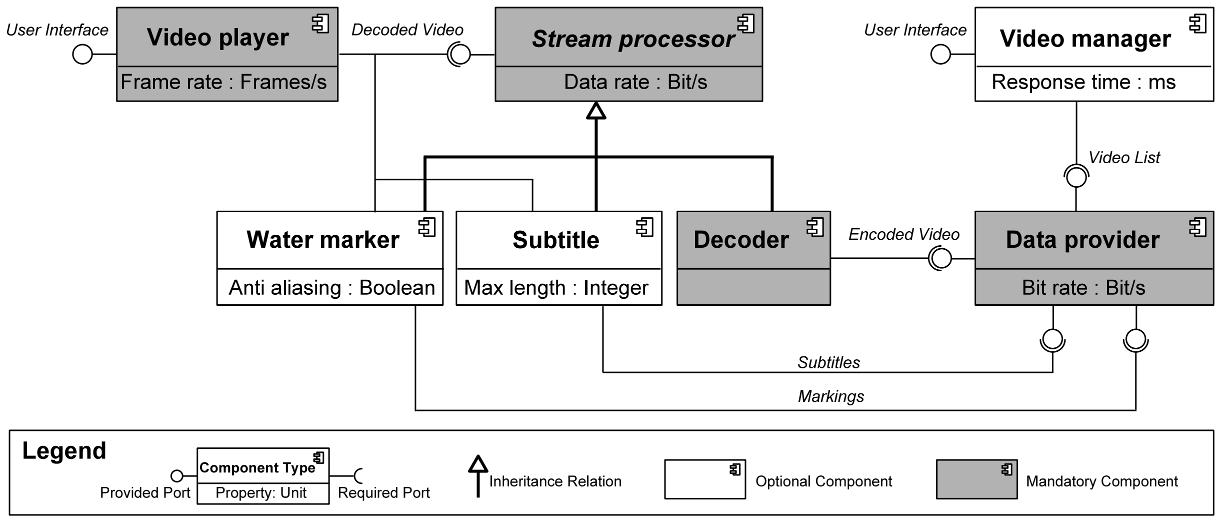


Figure 3.1 A variable video information system modeled as components.

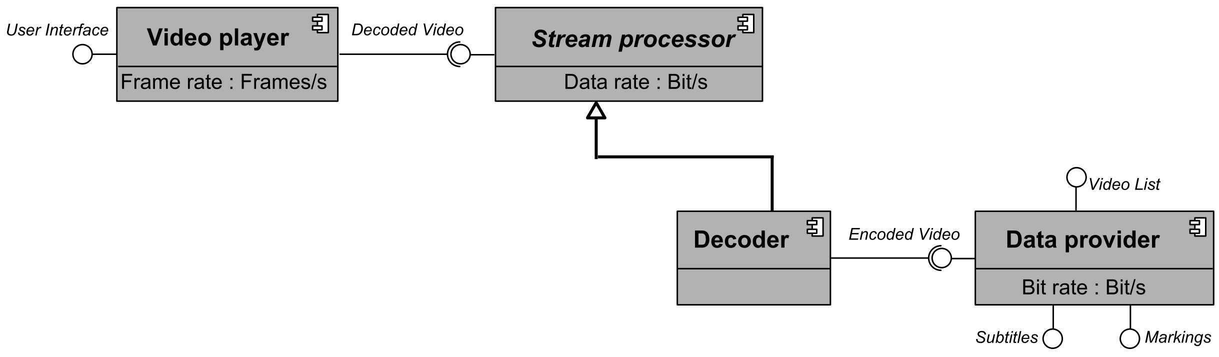


Figure 3.2 Configuration of customer A of the video information system.

component types required in all customer configurations are `Video player`, `Decoder`, `Data provider`, and an abstract `Stream processor`. Hence, these component types contain core functionality of the video information system. In addition, optional component types comprise a `Video manager` offering a user interface to add or remove videos from the library, a `Water marker` component type to add a customer-specific watermark to a video, and a `Subtitle` type to overlay videos with subtitles. In terms of multiplicities, each mandatory component type must be contained in a configuration exactly once, while optional component types can be contained zero or one time. Moreover, the component types `Water marker`, `Subtitle`, and `Decoder` inherit from the mandatory but abstract `Stream processor` component type as shown in the figure. This abstraction enables customers to select an arbitrary combination of the inheriting component types.

Each of the inheriting component types provides a decoded video. However, only the `Decoder` component type decodes an encoded video received from the `Data provider`, while the other two component types require an already decoded video. In a customer configuration, any combination of these three component types is allowed, but the `Data provider` component type must be at least contained as visualized by the gray background of this component type in the figure.

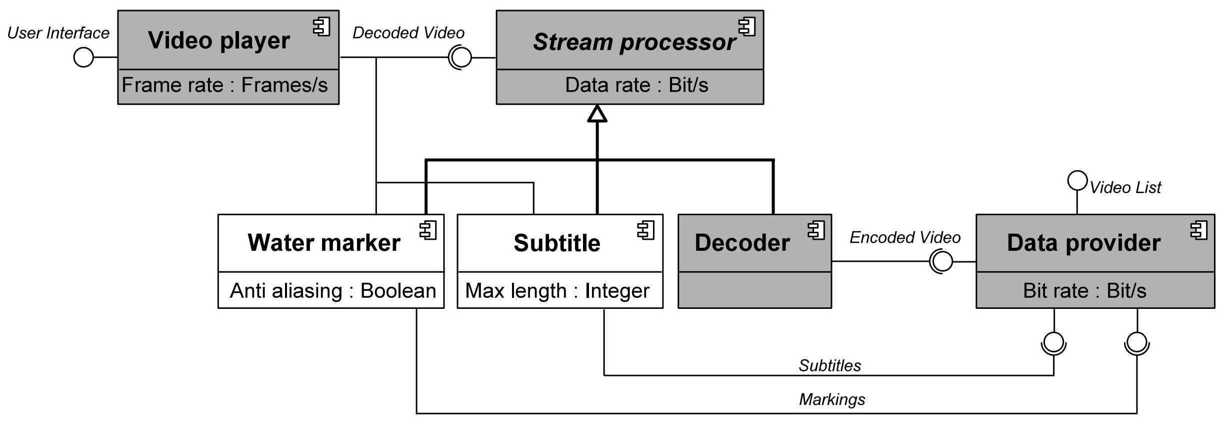


Figure 3.3 Configuration of customer B of the video information system.

Table 3.1 Software component types and their implementations in the video information system example.

Software Component Type	Implementation
Video player	VLC media player, AVS video player
Decoder	Free, Commercial
Data provider	URL, File
Water marker	Transparent, Classic
Subtitle	Single language, Multi language
Video manager	Basic, Standard, Professional

Two customers A and B of the application with different configurations are assumed. The configuration of customer A is depicted in Figure 3.2. This configuration only contains mandatory components according to the specification in Figure 3.1. The configuration of the second customer B is shown in Figure 3.3 and further contains the `Water marker` and `Subtitle` component types.

Moreover, component ports define required and provided functionality of the component types. In addition, each component type defines a set of qualities in terms of component properties characterizing how the provided functionality is fulfilled. For instance, the `Video player` component type has the property `Frame rate` describing how fast a video is being played expressed by the number of frames per second. The `Water marker` component type has a Boolean property `Anti aliasing` to specify whether overlay water marks are smoothed in a decoded and streamed video.

Each component type that is not abstract, has potentially many implementations. The implementations of the component types in this example are listed in Table 3.1. All implementations of a component type provide the same functionality, but differ in their internal implementation and in their provided and required qualities. For instance, a `Video player` component type is implemented by the `VLC media player` and the `AVS video player`. Both implementations play a video stream, but provide different frame rates and video resolutions.

Furthermore, customers may have requirements regarding the quality of the provided application. For instance, customer A requires strong hardware encryption, high application availability, and the application to run only on servers located in the European Union due to legal restrictions. In contrast, customer B does not specify any restriction regarding quality.

Different configuration artifacts to represent these concerns are distinguished in a cloud application. Artifacts developed by applying an SPL-based configuration management approach are explained in the following section.

3.2. Configuration Artifacts of Reconfigurable Cloud Applications

Reconfigurable cloud applications that are scalable in terms of functionality and quality comprise different configuration artifacts on varying levels of abstraction. Figure 3.4 depicts the configuration artifacts of a reconfigurable cloud application developed by applying an SPL-based configuration management in combination with a multi-tenant aware architecture.

In SPL engineering, problem and solution space are distinguished, as explained in Section 2.2.2. Hence, the different configuration artifacts are assigned to these spaces in the figure. Configuration artifacts in the problem space are on a higher level of abstraction than in the solution space.

In the proposed SPL-based approach, the configuration space of a reconfigurable application is described by means of a feature model. Based on this, customers define constraints on the availability of software and hardware components for users in feature model configurations. Hence, configuration artifacts in the problem space are partial and complete configurations of a feature model constraining functionality and quality of a cloud application. In the solution space, artifacts comprise software and hardware components, their instances and quality contracts.

For instance, variability of the video information system example, explained in Section 3.1, can be represented conceptually as a feature model to abstract from component types and implementations. Figure 3.5(a) depicts an example feature model of the video information system. The root feature represents the application, while features on the first level represent component types and features on the second level represent variable component instances. Hence, the extended feature model contains 20 features and describes 3456 valid configurations. A configuration of this feature model defines which component types and implementations are available for selection during runtime. For instance, if both child features `VLC media player` and `AVS video player` are selected, the final decision which one to choose is conducted at application runtime by evaluating the user request. If only one of the two implementations is selected, only this implementation is available for users.

Introducing further features and attributes in the feature model enables customers to define quality constraints. As explained before, the quality constraints of customer A regarding the availability and location of servers, as well as the encryption standard can be expressed by attributed

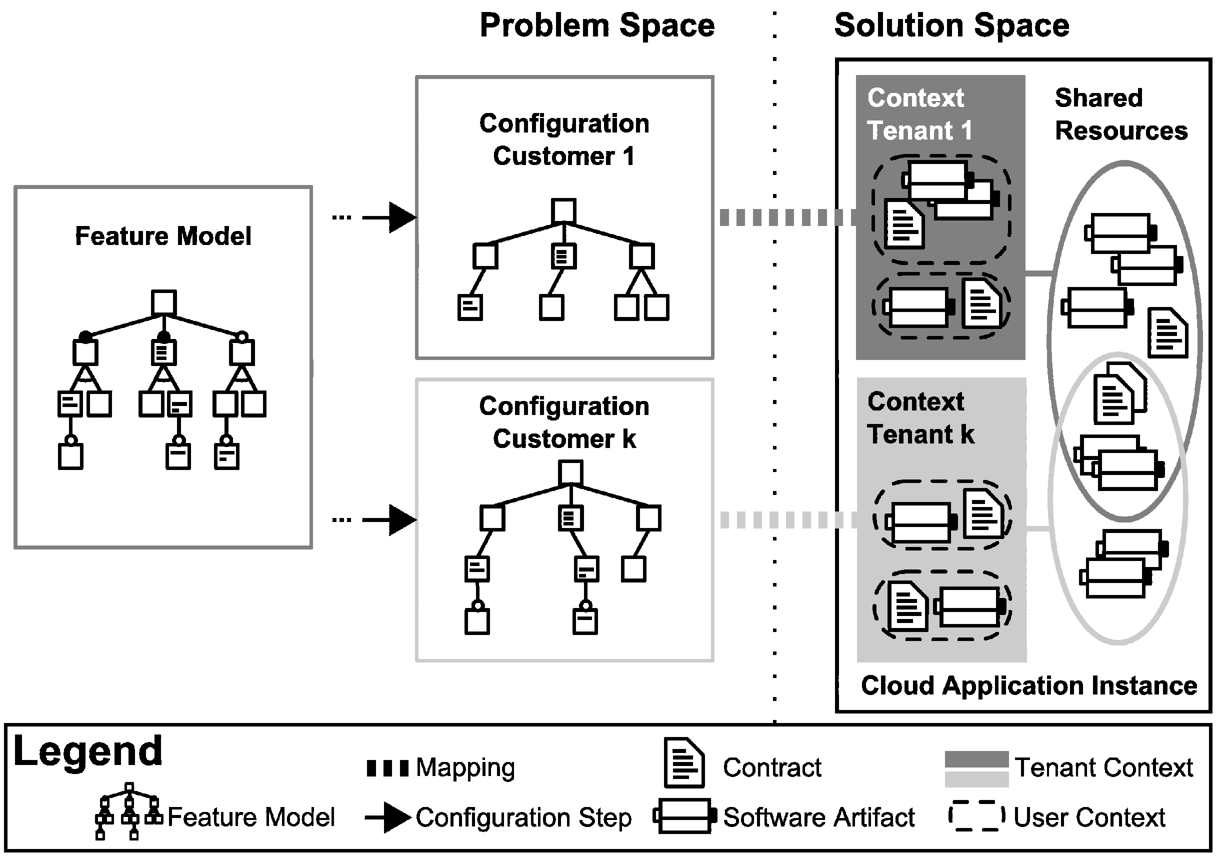
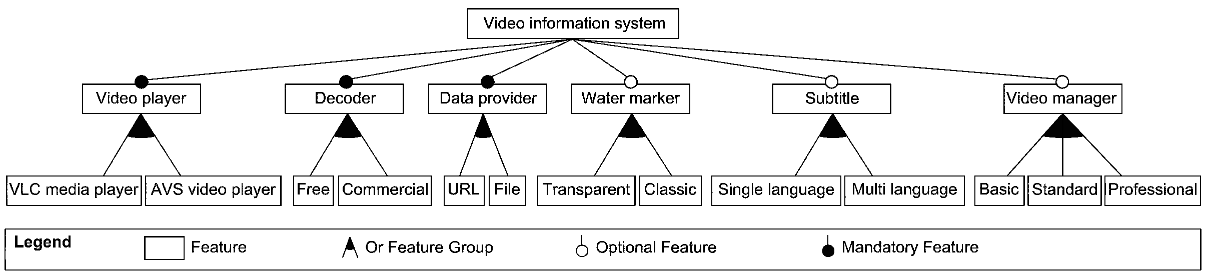


Figure 3.4 Feature configuration artifacts in the problem space are instantiated by software artifacts in the solution space

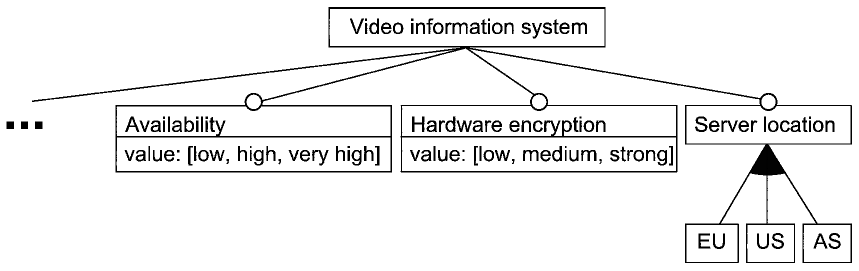
features as visualized in Figure 3.5(b). In this figure, features representing component types and implementations are left out, as indicated by the three dots. However, as the specification of quality constraints is not required, the corresponding features are modeled as optional. To model that only one value has to be selected for availability and encryption, both are modeled as feature attributes.

To constrain the server location, many values can be selected denoted by the *or* feature group containing the *US* to represent the United States, *EU* to represent the European Union, as well as *AS* to represent Asia. This is one possible representation of quality constraints in a feature model. The complete extended feature model is included in Appendix B.3. Although the model only comprises 26 features, 2 attributes and no cross-tree constraints, 387072 different complete configurations are derivable from the feature model.

While customer configurations are logically separated in the problem space, their realizations become physically dependent in the solution space, as they are integrated in a single application instance. The process of instantiating tenant configurations in the solution space from configurations in the problem space can be automated. Hence, a mapping between problem and solution space is specified as model transformation [KKL$^+$98, TCPB07, HW07], but is considered out of scope of this work as it depends on a particular application implementation.



(a) Component types and component implementations are modeled as features.



(b) Quality related requirements are modeled as attributes and features.

Figure 3.5 An example feature model for the video information system.

Reconfiguration occurs if the demands of customers, as well as their users change. Reconfiguration changes both the feature model configuration in the problem space as well as the current tenant context in the solution space. Changes are reported top down, such that reconfiguration in a customer configuration in the problem space triggers changes in the related tenant context in the solution space.

The described configuration artifacts are to be concisely modeled and built in automated configuration management introduced in the next section.

3.3. Feature-Based Configuration Management

Adopting SPL-based configuration management methods to reconfigurable cloud applications allows for automating the provision and reconfiguration of customized applications. Section 2.7 illustrates how configuration management in SPLs copes with the variation of product configurations. The configuration management framework proposed in this thesis manages variation by comprising methods for modeling variability and adaptive reconfiguration processes. However, the main focus of this thesis is on generic configuration management methods in the problem space. A discussion on how to develop flexible application architectures in the solution space is given that suite the configuration management methods applied in the problem space. The division of problem and solution space is taken up in Figure 3.6 to visualize the application of the proposed configuration management concepts.

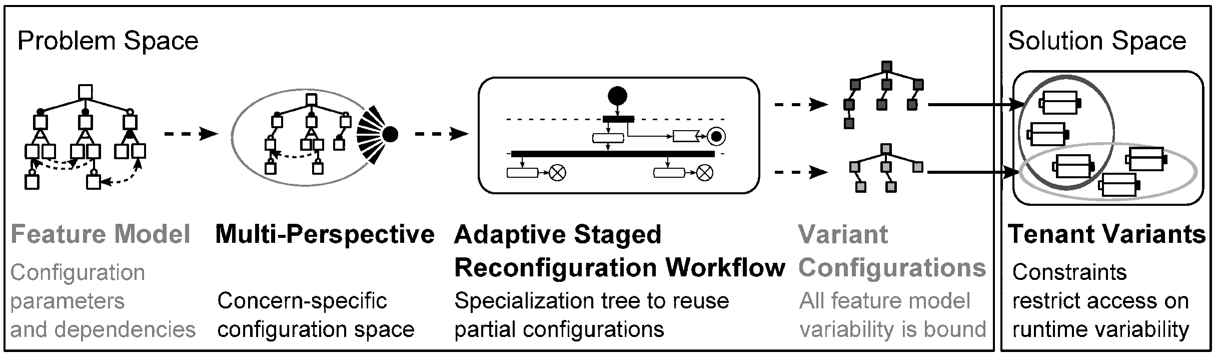

Problem Space
Solution Space
Feature Model
Configuration parameters and dependencies
Multi-Perspective
Concern-specific configuration space
Adaptive Staged Reconfiguration Workflow
Specialization tree to reuse partial configurations
Variant Configurations
All feature model variability is bound
Tenant Variants
Constraints restrict access on runtime variability

Figure 3.6 Concepts and their relation in the configuration management framework for reconfigurable cloud applications.

A feature model defines the configuration space of a reconfigurable cloud application in terms of features, qualities and their dependencies. For instance, the configurability of the video information system introduced in Section 3.1 is expressed by means of a feature model as visualized in Figure 3.5. However, enterprise cloud applications comprise a large number of configuration parameters expressed in a feature model, while common application scenarios often require the same configuration parameters, as explained by Requirement 2 in Section 2.8. Application providers therefore offer pre-configured editions for common scenarios.

Multi-perspectives are introduced to define consistent pre-configurations by comprising various concerns. Hence, they meet Requirement 2. Multi-perspectives allow for grouping features according to specific concerns. In the video information system example, pre-configurations can be defined according to professional and basic functionality, for instance. In this example, the features of the video information system are grouped according to business concerns. Table 3.2 shows the concerns *main*, *professional*, *basic*, and *advanced functionality*, as well as *quality related features*, and *mandatory* and *optional component types* and the assigned features.

In addition, concerns overlap and features are related to various concerns. A multi-perspective combines various of these concerns in a concise pre-configuration applicable for further configuration. The perspective guarantees that the dependencies defined in the feature model are still satisfiable. For instance, selecting the concerns *main functionality* and *professional functionality* listed in the table above leads to a pre-configured feature model containing 15 selectable features, while 48 complete configurations are still derivable. The concepts of multi-perspectives are explained in detail in Chapter 5 and implemented in the tool *Conper* explained in Section 7.5 as part of the *PUMA* tool suite.

A perspective constitutes the basis for further configuration steps leading to complete configurations. The derivation of a variant configuration is conducted in multiple steps by various involved stakeholders. *Staged configuration* concepts based on views and configuration workflows structure configuration processes and prioritize stakeholders of cloud applications. However, in a cloud scenario, not all stakeholders involved in the configuration workflow are known at application design time, and cannot be modeled beforehand according to Requirement 3. For instance, the two customers *A* and *B* are not known during application design time in the video

Table 3.2 Grouping features of the video information system example according to concerns.

Concern	**Related Features**
Main functionality	Video information system, Decoder, Video manager, Data provider, Video player, Water marker, Free codec, URL, Subtitle, Single language subtitle
Professional functionality	AVS video player, Commercial decoder, Transparent water marker, Multi language subtitle, Professional video manager
Basic functionality	VLC media player, File data provider
Advanced functionality	AVS video player, Classic water marker, Standard video manager, Multi language subtitle
Quality related features	Availability, Hardware encryption, Server location, EU, US, AS
Mandatory component types	Video player, Decoder, Data provider
Optional component types	Water marker, Subtitle, Video manager

information system. Both signed up to use the video information system during application runtime. Furthermore, configuration steps with different impact are to be distinguished.

While the configuration operations conducted by customers only influences their users, configuration operations conducted by providers have a global influence. For instance, in the video information system, assuming that the server infrastructure is modeled in the feature model as well, a *resource provider* restricts the availability of server nodes in the European Union due to local maintenance operations. Hence, the feature *European Union* is not available for selection in customer configurations. If the maintenance operations are finished, the *resource provider* enables the availability of server nodes again, which must propagated to all configuration stakeholders conducting downstream configuration operations.

The concept of *adaptive staged reconfiguration workflows* meets Requirement 3 and extends staged configuration. Staged configuration allows to specify and automate the derivation of variant configurations in multiple stages where various stakeholders are involved. These concepts are extended to handle stakeholders dynamically and to model the influence of stakeholder by means of a *specialization tree.* A specialization tree enables reuse of partial stakeholder configurations and automate reconfiguration at different configuration stages. Reconfiguration decisions are propagated along the directed configuration paths of the workflow.

Methods of Role Based Access Control (RBAC) are applied on extended feature models to restrict the access on configuration operations and define configuration views used in the staged configuration workflow. Combining staged configuration with an event-based adaptation engine allows for transforming the configuration workflow during runtime. A sequence of rewrite rules is applied to integrate a stakeholder dynamically in the configuration workflow as not all stakeholders can be defined before executing the workflow. Furthermore, stakeholders can be removed from the configuration workflow during runtime. For instance, customers subscribe and unsubscribe from an application offered as a service. The results of the staged configuration workflow are multiple complete variant configurations where all feature model variability is bound.

In addition, satisfiability of feature models, as well as of partial feature model configurations are checked by translating an extended feature model or a partial configuration into a CSP and executing a CSP solver accordingly. CSP checks are executed on partial feature model configurations during workflow execution to validate conducted configuration operations. The concept of adaptive staged reconfiguration workflows is explained in detail in Chapter 6 and implemented in the tool *DyscoGraph* explained in Section 7.6 as part of the *PUMA* tool suite.

The explained concepts belong to the problem space and abstract from implementation details related to the solution space. However, in the solution space a flexible application architecture is required according to Requirement 1 defined in Section 2.8. The architecture should be capable of integrating customer configurations to restrict the access on system resources and application functionality accordingly. Each complete configuration related to a customer in the problem space corresponds to a *tenant variant* and a set of customer-specific constraints in the solution space. These constraints restrict the access on system resources and application functionality at runtime for the users of a particular customer. By applying a multi-tenant aware application architecture, a complete configuration can be integrated in an already running application. Hence, the configuration becomes available for a customer just-in-time. The same holds for reconfigurations. A development method for a multi-tenant aware architecture based on a self-adaptive architecture is explained in Chapter 4.

Figure 3.7 summarizes the main contributions of the proposed configuration management framework and their foundations. Boxes with a gray background depict common methods and

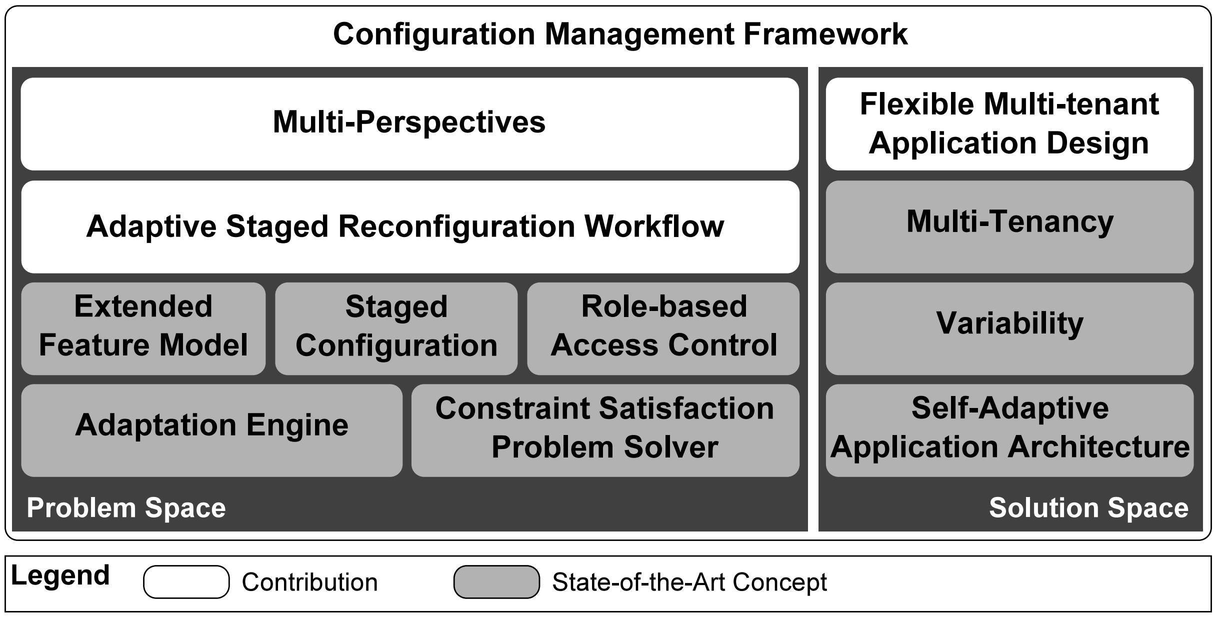

Configuration Management Framework
Multi-Perspectives
Adaptive Staged Reconfiguration Workflow
Extended Feature Model
Staged Configuration
Role-based Access Control
Adaptation Engine
Constraint Satisfaction Problem Solver
Problem Space
Flexible Multi-tenant Application Design
Multi-Tenancy
Variability
Self-Adaptive Application Architecture
Solution Space
Legend
Contribution
State-of-the-Art Concept

Figure 3.7 The conceptual configuration management framework for reconfigurable cloud applications is based on prevailing SPL concepts.

paradigms that build the basis of the configuration management framework. Based on these foundations, new concepts developed in this work are represented by boxes with a white background color. In addition, the introduced concepts support the main configuration management activities explained in Section 2.7.

3.4. Automated Configuration Management Activities

The five main configuration management activities comprise planning, identification, control, status accounting, and verification of configuration changes. Table 3.3 shows how the problem space related concepts of the proposed feature-based framework support the explained configuration management activities.

Summarizing, the proposed configuration management framework allows for a concise variation management in the problem space. Variable concepts are abstracted from implementation

Table 3.3 Automating configuration management activities for reconfigurable applications in the problem space by applying the configuration management framework.

Configuration Management Activity	**Automation**
Configuration management planning to define what, how and when configurations will be controlled	Plan an adaptive staged configuration workflow and configuration relevant business concerns covered in a multi-perspective model.
Configuration identification to specify controlled configuration items	Functional and quality properties of a cloud application are configuration items modeled as features and attributes and their relations in a feature model.
Configuration control monitors what changes are made, when and how	Multi-perspective models and adaptive staged configuration workflows ensure continuous integrity of configuration items. Configuration changes are automatically evaluated against constraints defined in the feature model to identify allowed and disallowed changes.
Configuration status accounting to determine and report the configuration status of any configuration item	The current status of all configuration items is persisted in each partial and complete feature model configuration.
Configuration verification and audit to guarantee that configuration items are configured correctly.	The requirements, dependencies and boundaries of each configuration item are explicitly defined in the feature model. The continuous check of configuration changes based on the feature model definition ensures the correct configuration of each configuration item.

technologies. In addition, tenant constraints in a multi-tenant aware single instance application provide the basis for managing variation in the solution space.

3.5. Demarcation from Related Work

Approaches applying methods from SPL engineering in the area of cloud applications to manage application variability are proposed in literature [Mie10, RA11]. The approaches have similar objectives of handling variability of multi-tenant aware applications, but vary in focus and the proposed realizations. Table 3.4 compares concepts presented in this work with these approaches. A systematically conducted literature study revealed differences in the following criteria chosen for comparison

- (i) the type of variability model applied to specify the application configurability,
- (ii) the support of concern-specific pre-configurations,
- (iii) if a structured configuration process can be modeled,
- (iv) the considered configuration stakeholders,
- (v) if partial configurations can be reused,
- (vi) the support of configuration at runtime,
- (vii) reconfiguration at runtime,
- (vii) multi-tenancy on architecture level,
- (viii) the kind of application architecture applied, and
- (ix) if an implementation of the proposed concepts is provided.

An SPL-based development environment for SaaS applications applying OVM for modeling application variability is proposed by Mietzner [Mie10]. Overlapping concerns on the variability model are not considered in this approach. Structured configuration processes for customers can be defined in the workflow language BPEL. However, only tenants are considered as stakeholders conducting configuration operations in the configuration process. The approach does not support reconfiguration nor the reuse of pre-configurations. The application architecture applied in this approach is service oriented and multi-tenant aware. However, the architecture does not support configuration and reconfiguration at runtime. An implementation of the proposed concepts is provided in the tool suite *Cafe*.

Rühl and colleagues outline an approach for handling the variability and managing configuration of multi-tenant aware SaaS applications [RA11]. This work proposes to apply a proprietary catalog to specify application variability without explicitly modeling overlapping concerns on the

Table 3.4 Comparing SPL-based configuration management approaches for cloud applications.

Characteristic	**Mietzner et al. [Mie10]**	**Rühl et al. [RA11]**	**This thesis**
Variability model	OVM	Proprietary catalog	Extended feature model with attributes over finite domains
Overlapping concerns on the variability model	-	-	+
Structured configuration process	Customization flow defined in Business Process Execution Language (BPEL)	Single step configuration	Adaptive staged configuration workflow
Configuration stakeholder	Tenants only	Tenants only	Various stakeholders
Reuse of partial configurations	-	-	+
Configuration at runtime	-	+	+
Reconfiguration	-	-	+
Multi-tenancy	+	+	+
Implementation	Tool suite *Cafe*	-	Tool suite *PUMA*

variability model. An application is configured by a tenant in a single step without the need for a structured configuration process as there are no other stakeholders considered. Moreover, the reuse of pre-configurations and reconfiguration of the application is out of scope of this work. A variable service-oriented and multi-tenant aware application architecture is assumed without providing more details on the architecture.

Concluding, these approaches focus on the variability management of SaaS applications in the cloud, but apply a different variability model and have another focus than this thesis. Based on methods from SPL engineering, the configuration management framework proposed in this work offers generic methods in the problem space. These methods comprise the definition and execution of structured configuration workflows by means of an *adaptive staged reconfiguration workflows* and the definition and derivation of consistent pre-configurations by means of *multi-perspectives*.

Hence, the focus of this work is on configuration and reconfiguration processes during application runtime. Furthermore, based on a self-adaptive architecture, a development method for multi-tenant aware and reconfigurable cloud applications is proposed that is convenient for feature-based configuration management.

3.6. Summary

The configuration management framework introduced in this chapter allows for realizing cloud applications scalable in terms of functionality and quality. Hence, cloud applications can be configured and reconfigured in a self-service portal without the need for manual interaction while the changes are available just-in-time in the application instance. Well-established methods from SPL engineering are applied in the framework to cope with variability and manage changes to configuration artifacts automatically.

Feature models are common in SPLs to concisely define the configuration space of an application in terms of features and their dependencies. In addition, staged configuration concepts focus on the structured derivation of complete variant configurations where different configuration stakeholders are involved. The framework extends these methods to address the specific requirements for reconfigurable cloud applications summarized in Section 2.8. For instance, the new concept of *multi-perspectives* allows for defining concise concern-specific pre-configurations on feature models prior to a configuration process. In addition, staged configuration is extended to support the reuse of pre-configurations for various dynamic stakeholders and to support reconfiguration and is realized via *adaptive staged configuration workflows*. Moreover, a *development method for flexible application architectures* is proposed enabling the automated integration of configuration changes into the application instance at runtime. The method combines multi-tenancy with a self-adaptive component-based architecture.

Hence, the three main parts of the framework are a *development method for flexible application architectures*, *multi-perspectives*, and *adaptive staged configuration workflows*. While the development method can be assigned to the solution space, multi-perspectives and adaptive staged configuration workflows are related to the problem space. This chapter gives a comprehensive overview of the functional interaction between these parts. How to develop flexible multi-tenant aware architectures is explained in the next chapter, while the problem space related parts are explained in the subsequent chapters.

Summarizing, the configuration management framework proposed in this chapter is a foundation to manage variability of reconfigurable cloud applications, and to support configuration and reconfiguration processes on an abstract level by means of SPL methods.

4. A Flexible Architecture for Reconfigurable Cloud Applications

Stay committed to your decisions, but stay flexible in your approach.

— *Tom Robbins*

Concepts presented in this chapter are published in a paper at the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS) [SCG+12].

This chapter observes main characteristics of reconfigurable cloud applications with respect to their architecture that are relevant for automated feature-based configuration management. Furthermore, architectural concepts are identified that implement these characteristics and a corresponding example architecture for reconfigurable cloud applications is discussed.

As summarized by Requirement 1 in Section 2.8, a reconfigurable cloud application requires a flexible and modular architecture capable of handling variability in various dimensions, (i) among stakeholders sharing resources, and (ii) within a stakeholder configuration due to reconfiguration changes. In addition, different types of stakeholders are identified in the context of reconfigurable cloud applications, as explained in Section 1.6. For instance, *resource providers* offer the infrastructure and computing platform comprising server nodes, network communication, storage, and databases, as infrastructure and platform services. *Application providers* utilize these services to deploy a cloud application on top. *Customers* are interest groups or companies renting tailored application functionality and *users* affiliated to specific customers use the cloud application. These stakeholders have different and even contradictious requirements. A user demands a high quality of the application in terms of availability and performance. In contrast, resource and application providers intent to minimize operating costs by optimizing resource utilization while maximizing profit.

Resource utilization can be optimized by applying a multi-tenant aware architecture, as explained in Section 1.5.5. Furthermore, multi-tenancy allows for just-in-time provisioning of tailored customer configurations and reconfigurations as the application instance is already running and does not need to be set up and deployed explicitly. However, a multi-tenant aware cloud application must not be shut down to integrate new configurations or to reconfigure according to changing demands because the application instance is shared among customers. Hence, an adaptive application architecture is required supporting reconfiguration at application runtime.

In addition, multi-tenancy comprises two main concepts, (i) resource and data sharing among application users, and (ii) user access restriction on resources, functionality, and data with respect to an affiliated tenant company. While resource and data sharing is similar to multi-user applications, the restriction on runtime variable functionality and application data differs. Tenant companies and their users have varying requirements on the functionality of the application and on the infrastructure where their data is processed according to legal restrictions, as explained in Section 1.5.3.

The focus of this work is on architectural concepts for dynamically restricting the user access on application functionality and resources with respect to tenant companies and user requirements. This chapter describes a method for developing reconfigurable cloud applications based on a self-adaptive architecture [SCG$^+$12]. Multi-tenant extensions of the architecture regarding access restrictions on functionality and resources are proposed that are evaluated at runtime to adapt an application automatically to the requirements of a tenant and its users. For a common understanding, concepts of software architectures are explained first.

4.1. Software and Product Line Architecture

In the context of software and system engineering, the notion of *architecture* is ambiguous. An empirical study identified the following key understandings of architecture within software organizations [Smo02]. Hence, architecture is

- a *blueprint* defining the structure of an implemented system,
- *literature* to document software solutions and to function as a reference for future development,
- a *language* for common terminology and understanding of the structure of a system regarding communication between stakeholders, and
- a *decision* about system structure with respect to business concerns.

However, any of these architecture metaphors are relevant to successfully develop software and these metaphors do not occur solely in organizations. They can be considered as important aspects of a system considered during system development. In addition, the understanding of architecture differs between stakeholders and their roles in a company [Zac87]. While data base experts have a data-centric understanding of architecture, the architectural view of an operational board is defined in monetary terms with respect to costs and benefits of architectural decisions. The ISO/IEC/IEEE 42010:2011 standard continues on the different aspects of architecture by providing a definition as follows [ISO 42010].

Definition 4.1 (Architecture [ISO 42010]). Architecture *is a set of (system) fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.*

According to this definition, an architecture captures the essential parts of a system, while a system is build of a variety of resources and entities of different type. Hence, fundamental parts of a system can be physical and structural components, as well as functional and logical elements.

A common understanding is that architecture describes the structure of a system in terms of components and their relationship [SDK$^+$95]. Due to the modular design of components they are reusable in different systems. However, architecture does not define what a component is, rather than the rational of the components and their structure.

For products of the same domain, a so called *reference architecture* is specified to describe the variations between the products and to cope with complexity while reducing the development costs. Adopted from the ISO/IEC/IEEE 42010:2011 standard on software engineering a reference architecture is defined as an architectural framework for a particular application domain serving as a template for instantiating concrete architectures [ISO 42010]. Hence, a reference architecture comprises conventions, principles and practices for the description of architectures established within a particular application domain. A Product Line Architecture (PLA) is a reference architecture implementing variability and commonality to capture the high level design of derivable products of a certain domain [PBvdL05]. To specify variability and commonality in a PLA, often the notation of components is applied, as components enable the modularized reuse of functionality.

4.1.1. Product Line Architecture

A PLA describes variability among products with a strong focus on reuse. In SPL engineering the two phases of domain and application engineering are distinguished, as explained in Section 2.2.1. Following a component-based SPL approach, component engineering is part of the domain engineering phase. Components encapsulate reusable functionality in individual building blocks also called core assets. The way how core assets are composed is defined by a PLA. Component properties, their interfaces and interactions of the application are defined by means of structural, behavioral and functional models in a PLA. Subsequently, components are assembled and integrated in application engineering to constitute a product [WL99].

However, in common SPL approaches, a single product is derived per configuration, as explained in Section 2.6.2. The PLA defines how to compose a stand-alone product instance for a given configuration. In contrast, in an SPL-based reconfigurable multi-tenant aware PLA, multiple configurations are derived but integrated into a single product instance at the same time. Logically independent configurations in the problem space become physically dependent in the application instance in the solution space, as discussed in Section 3.2. These configurations constrain the access of users as each user of the system is affiliated to a tenant.

For cloud applications in general, a commonly applicable reference architecture cannot be defined due to varying business concerns and therefore different applicable cloud concepts, as explained in Chapter 1. Hence, this work does neither propose an enterprise architecture framework nor a complete PLA for reconfigurable applications as both are domain-specific and depend on business decisions that drive the particular application functionality and system structure. However, the focus of this work is on how to realize resource variability at runtime with respect to tenant configurations constraining the access of their users. A solution is proposed retaining tenant and user requirements. The solution is based on a self-adaptive architecture to address the requirements of reconfigurable cloud applications.

In the next section characteristics of reconfigurable cloud applications are described and architectural concepts applicable in a PLA to address them are explained.

4.2. Characteristics of Reconfigurable Cloud Applications

As identified in Chapter 1, a reconfigurable cloud application has the following characteristics. The application design is modular to enable reuse and compose parts according to customer requirements. A reconfigurable cloud application scales in two dimensions. First, in terms of functionality and quality to serve customers according to their functional and quality demands. Second, in terms of concurrent user access and the number of tenants, as explained in Section 1.5.1. Resources of the application and underlying infrastructure are to be shared to save development and maintenance costs, as explained in Section 1.5.2. The application requires a self-service administration portal for configuration and reconfiguration. In addition, configuration and reconfiguration of the application are to be automated to omit manual interaction according to Section 1.5.3.

The characteristics of reconfigurable cloud applications and architectural styles implementing them are summarized in Table 4.1. A *component-based* application design is convenient for developing modular applications accessed over the Internet [ABGK02], and hence, applicable for cloud applications. Variability and thus scalability in terms of functionality and quality is realized by defining *contracts* between components to achieve loose coupling between components.

Adapting and reconfiguring an application during runtime requires a robust architectural style referred to as *self-adaption*. As explained previously, different stakeholders with different objectives are involved in the provision and configuration of a cloud application. The system can be optimized to fulfill the often contradictious stakeholder objectives by applying a self-adaptive approach. Furthermore, resources between customers can be utilized efficiently by applying a *single-instance multi-tenancy* approach. Eventually, customers can be enabled to perform configuration and reconfiguration on their own while the changes become available just-in-time, which can be achieved by applying an SPL-based configuration approach with an automated configuration derivation.

Table 4.1 Characteristics of reconfigurable cloud applications and corresponding architectural concepts.

Application Characteristic	**Architectural Concept**
Modular	Loosely coupled components [GTL00]
Variable in terms of functionality and quality	Components with contracts [BJPW99]
Adaptation and reconfiguration at runtime	Self-adaptation [Lad99]
Optimized resource utilization	Multi-tenancy [CC06]
Customer-specific (re)configuration in a self-service portal and changes available just-in-time	Automated SPL derivation process [WL99]

A multi-tenant aware reconfigurable cloud application copes with variability at runtime in two dimensions, (i) among customers, and (ii) within a single customer configuration due to reconfiguration changes. Furthermore, reconfiguration and adoption to stakeholder requirements is performed during application runtime, as well as integration and decommissioning of customers. Frameworks proposed in literature for dynamically adaptive systems address most of these requirements [GKR+06, MBJ+09]. These frameworks define systems, which adapt automatically to requirement changes or usage contexts at runtime.

An example application architecture comprising the explained characteristics is depicted in Figure 4.1. The block diagram shows the different layers of the architecture. The example architecture contains the three layers *infrastructure*, *platform*, and *application* according to the NIST definition explained in Section 1.2, as well as a *presentation layer* on top and a further cross-cutting *management layer*. The infrastructure layer provides network, servers, and storage as services. These services are a basis for the platform layer which provides foundation services to access the underlying infrastructure services, as well as a multi-tenant aware database to separate tenant-specific data from each other as described in Section 1.5.5. Subsequently, the platform services are consumed by software components on the application layer.

Software components are also multi-tenant aware as their functionality is shared. Furthermore, these components are reconfigurable in terms of component binding as different customers have different application configurations causing varying control flows. In addition, the presentation layer consumes services provided by the application layer to display an application front end to

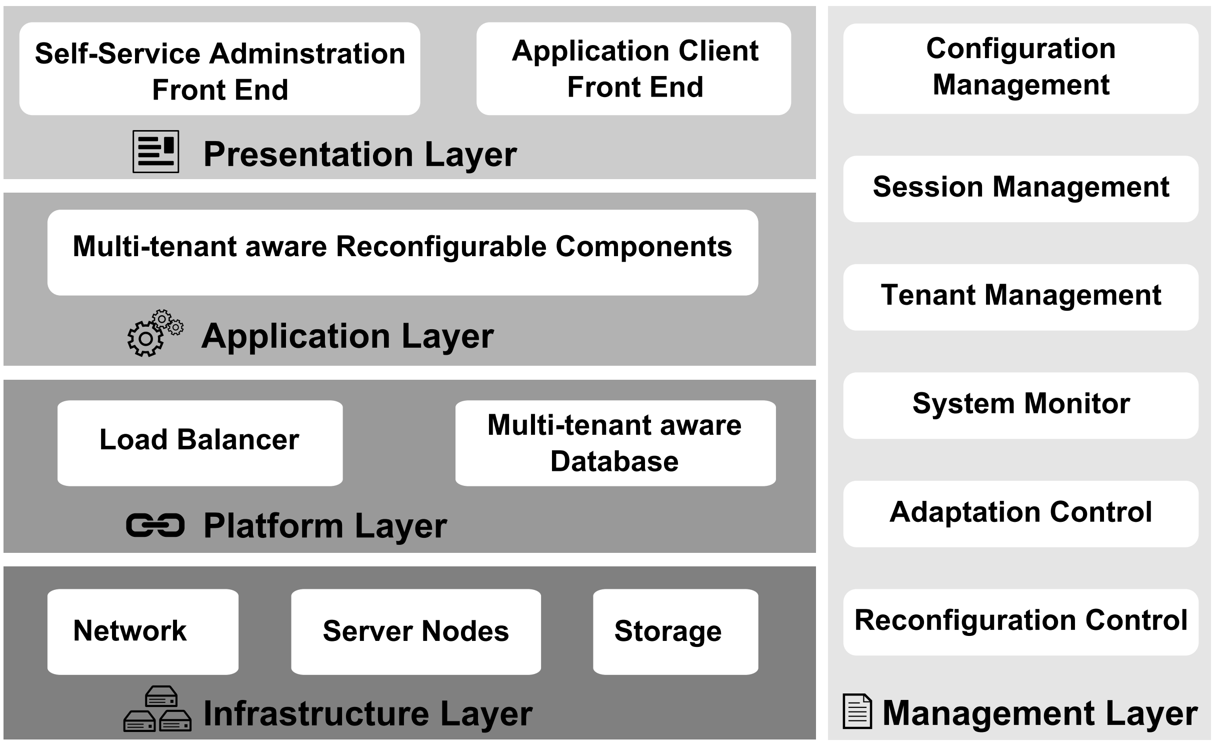


Figure 4.1 Example architecture of multi-tenant aware reconfigurable cloud applications.

the application users and to offer a self-service administration front end to tailor an application. A further management layer cross-cuts all four stacked layers. This layer manages the services of all four layers and comprises configuration management to automatically handle changes, and tenant management to identify and assign system resources to tenants. Furthermore, a system monitor observes the system utilization, and adaptation and reconfiguration of the services are controlled. Session management enables stateful communication to separate control and data flow of users.

This layered architecture reflects the defined characteristics and is the basis for developing a flexible architectures for reconfigurable applications. Furthermore, reconfigurable cloud applications under the following infrastructure conditions are considered. A cloud infrastructure comprises a server cluster with distributed server nodes. Server nodes communicate with each other and if a node fails applications are automatically transferred to another server. Additionally, the server cluster is horizontally scalable and server nodes are dynamically utilized, as explained in Section 1.5.1. Furthermore, resources in the cloud are instrumented to measure their quality at any time.

The components of an application are distributed over multiple physical nodes in the server cluster. In addition, at design time of the application not all customers and their users are known. Hence, it is impossible to model all potential customers and their users in advance.

The next section explains requirements for developing reconfigurable cloud applications capable of restricting the user access according to multi-tenancy constraints.

4.3. Developing a Flexible Architecture for Reconfigurable Cloud Applications

A component-based application architecture with a modular design is a good foundation for PLAs, as discussed previously. Each component has a design time and a runtime representation [Szy98]. A modeling language is appropriate to precisely specify an application at design time, while at runtime these models are applied to optimize stakeholder objectives. Hence, further design time and runtime requirements on an architecture for reconfigurable cloud applications regarding application variability at runtime and constraining the access on resources and functionality are distinguished. The design time requirements are listed below.

- *Modeling Software and Hardware Components* Customers have constraints on functionality, as well as on the underlying hardware resources. To automatically compute these constraints, components must be modeled uniformly in a convenient notation. Hence, a modeling language for specifying reconfigurable cloud applications is required to define software and hardware components and their relations. While software components may depend on software or hardware components, hardware components may only rely on hardware components. The application provider models the application architecture, while the resource provider models the hardware infrastructure, both in the same modeling language.

- *Specification of Qualities* Customers and users have constraints on the quality of software and hardware components. Hence, a language for modeling applications must be capable of expressing qualities and their dependencies.

- *Modeling Tenant Configurations* Each application customer has an own application configuration. Hence, modeling of tenant configurations is required. A tenant configuration comprises a unique identifier for a customer, a set of software and hardware components and customer-specific constraints on application qualities on the level of hardware and software components.

The following runtime requirements are to be addressed by an applicable architecture.

- *Self-Optimizing Runtime Environment* A flexible architecture of reconfigurable cloud applications requires a runtime environment that is self-adaptive. Hence, the runtime environment must reason about infrastructure, software components, and tenant configurations to optimize and adapt itself regarding the software component distribution according to tenant configurations. In addition, the runtime environment must scale to multiple users as a multi-tenant aware application is a special form of a multi-user application, as explained in Section 1.5.5. Hence, in the runtime environment control and data flow between users of the application must be separated and are further restricted by a user's tenant affiliation.

- *Quality Monitoring* The qualities defined at application design time must be monitored to reason on software and hardware components, as well as on tenant configurations. However, only qualities that change during runtime need to be measured periodically.

- *User Management* To balance the objectives of a user and the affiliated tenant, tenant configurations and user requests are to be managed appropriately.

- *Reconfiguration of Tenant Configurations* A cloud application must be scalable in terms of functionality and quality. Customer requirements change and thus their tenant configurations change accordingly.

In the following, concepts of Multi-Quality Auto-Tuning (MQuAT) are explained that address the identified requirements and mark this architecture eligible for implementing reconfigurable multi-tenant aware applications.

4.4. Background on Multi-Quality Auto-Tuning Architecture

MQuAT is a reference architecture for self-adaptive systems. The MQuAT architecture is developed in the *cool software* project[1] as a component-based software architecture for implementing self-adaptive and self-optimizing applications [GWS$^+$10]. Component-based architectures are loosely coupled and support Separation of Concerns (SoC) on implementation level, where each software component encapsulates a set of semantically related functions [McI68]. Thus, MQuAT

[1]`http://www.cool-software.org/`

as a component-based architecture is convenient to build reusable software assets for variant configurations in the solution space of SPLs [CE00].

DSPLs and configurable cloud applications respectively, further demand a reconfigurable architecture, as explained in Section 2.6. MQuAT as an auto-tuning architecture is eligible to implement DSPLs by supporting application reconfiguration at runtime if changes occur during operation. Internal changes in the application instance and external changes in the application environment are therefore considered. For this purpose, MQuAT comprises a control loop to monitor the system, detect changes, decide how to react to the changes, and act by executing the decisions [DDF+06]. Hence, a centralistic monitoring and measuring unit observes the system. In contrast to agent-based systems, where each agent optimizes local knowledge, the optimization unit has a global view on the entire system [GWCA11]. In MQuAT contracts between hardware and software components in terms of qualities are defined.

A further characteristic of the MQuAT architecture is its auto-optimizing capability towards multiple system qualities in terms of objectives. The approach was initially intended for energy auto-tuning, but iteratively extended to support multi-objective optimization. For instance, MQuAT finds the optimal configuration of a system providing best performance while saving at most energy. Objectives are often contrary and different strategies for optimizations are discussed in [Göt13], which are out of scope of this work.

4.4.1. Essential Parts of the Multi-Quality Auto-Tuning Architecture

The MQuAT architecture comprises a *design time part* providing a development method for self-optimizing systems and applications and a *runtime part* for realizing self-optimization during system and application operation, as depicted in Figure 4.2.

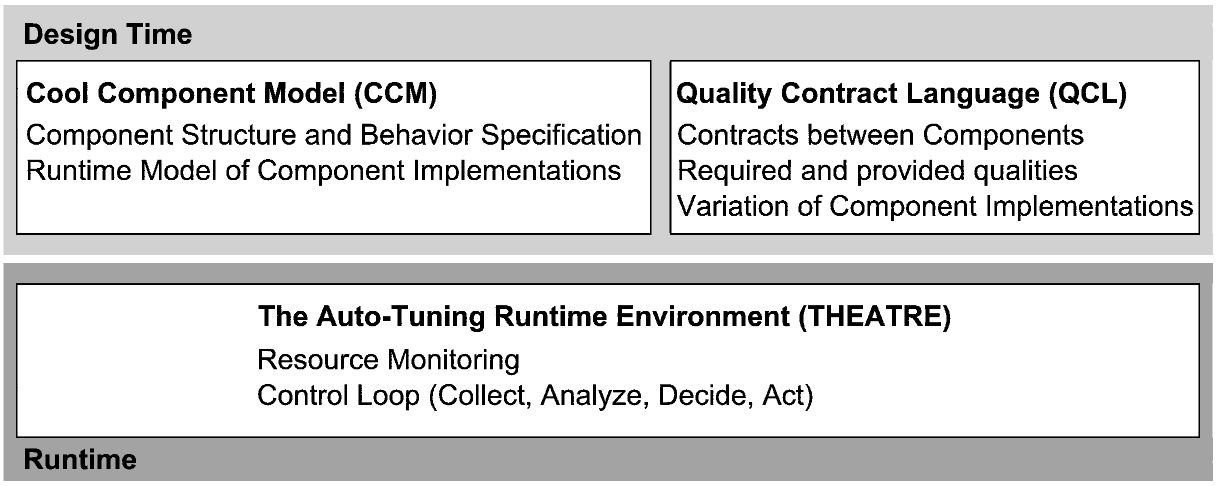

Design Time
Cool Component Model (CCM)
Component Structure and Behavior Specification
Runtime Model of Component Implementations
Quality Contract Language (QCL)
Contracts between Components
Required and provided qualities
Variation of Component Implementations
The Auto-Tuning Runtime Environment (THEATRE)
Resource Monitoring
Control Loop (Collect, Analyze, Decide, Act)
Runtime

Figure 4.2 Overview of the integral parts of the MQuAT architecture (adopted from [GWS+10]).

The design time part comprises a component model to define conceptual properties of components and characteristics of component relations. In MQuAT, the component model contains the Cool Component Model (CCM) to define component types, their properties, and the Quality Contract Language (QCL) to specify relations between components in terms of quality contracts. Hence, the CCM defines component types, their behavioral states, variations among component implementations, and describes the runtime state of a system. The language QCL is a language to define quality contracts between components regarding required and provided non-functional properties. Furthermore, implementations of CCM component types are defined in QCL contracts. Summarizing, the development time part comprises a functional model providing the operation specification and a structural model defining exposed properties of the component.

The runtime part comprises The Auto-Tuning Runtime Environment (THEATRE), a quality-aware runtime environment that implements a reconfiguration control loop and provides a global monitoring unit to observe the system. Pre-conditions specify assumptions that must be fulfilled to execute a method correctly while post-conditions are fulfilled after executing the method. This concept is the foundation for QCL, an expressive contract language for specifying contracts between CCM components with respect to quality attributes. The contracts are evaluated during runtime in THEATRE. If a contract is not meet by the current system configuration, components that fulfill the contract are chosen and reconfiguration is initiated.

A specification of THEATRE and the MQuAT architecture is given in [Göt13]. In the following, design time and runtime parts are explained relevant for reconfigurable cloud applications.

4.4.2. Design Time Declaration of the Cool Component Model and Quality Contracts

The structure and instantiation of IT infrastructures and applications is defined in CCM in a component-based type system. The syntax of CCM is described in a schema following a metamodeling approach to separate the component definition from a specific implementation technology, while CCM models are interpreted at runtime. The CCM metamodel covers various aspects of quality-efficient self-optimizing systems. CCM comprises a *structure model* that defines component types and their interfaces with respect to functionality and quality attributes, and a *variation model* that describes the runtime state of a system.

An excerpt of the structure model is depicted in UML class diagram notation in Figure 4.3. The figure shows supported component types in a structure model, which are *user*, *software* and *hardware* component types. Software and hardware components can be hierarchically structured, as visualized by the composite pattern in the figure. Hence, the component types described in the video information system example in Section 3.1 can be modeled as software component types in a CCM structure model. Furthermore, the structure model describes hardware components. For instance, a CCM structure model typically contains a `Server` component type comprising further hierarchically structured hardware component types such as a `CPU`, `HDD`, and `RAM`. Software components can be organized hierarchically as well. In addition, a component type defines required and provided functionality by means of ports and defines quality attributes by means of component properties, as exemplified in Section 3.1. For instance, a `Data provider` component provides functionality in terms of `Video Lists`, `Encoded Videos`, `Subtitles`, and

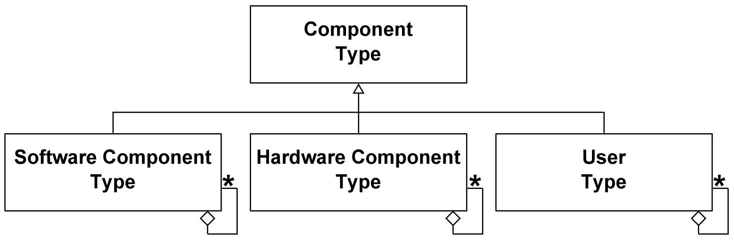


Figure 4.3 Excerpt of the metamodel for CCM structure models.

`Markings`, while specifying the quality `Bit rate` to specify data transmission. The specification of component properties in a component type makes component implementations comparable at runtime. All implementations of the same component type provide the same functionality, but differ in the property values in the provided and required interfaces. Each component type can be instantiated by multiple implementations which are described by means of component instances in QCL contracts. In a QCL contract software implementations and hardware resource instances are defined referring to the corresponding component types in the structure model.

The variation model describes the runtime state of a system by modeling deployed component implementations and their location. A component instance comprises information, where this instance is currently deployed, and offers a value for each quality property defined by the referring component type. Different types of component properties are distinguished, which are *static-instance*, *monitored*, and *calculated*. Static instance properties represent invariants, such that the value of these properties is constant for each component implementation. Monitored properties are measured by instrumenting the system, and a property is calculated, if the value of this property is computed-based on the values of other properties. Variation models and corresponding structure models describe the current state of a system and potential alternatives.

To reason about the optimality of a system, information about how components depend on each other must be specified. In CCM component dependencies are expressed by means of QCL contracts. Component instances have potentially multiple internal behaviorial states described by modes in a QCL contract. For instance, a `CPU` can run in various performance modes with varying resource utilization and thus differing frequency. Hence, modes differ in terms of their required and provided quality properties. A contract is defined per component instance declaring such implementation modes, where modes reflect different levels of user satisfaction. Each mode declares required component types and defines constraints on the minimal and maximal values of quality properties of these component types. In addition, the minimum and maximum values of the provided quality properties are defined.

Listing 4.1 depicts an example contract for the `VLC media player` implementation of a `Video player` component type according to the example in Section 3.1. In this example, the declaration of further hardware component types such as `CPU` and `Network` in a structure model is assumed. The contract defines three modes for the `VLC media player` implementation, which are `fluent`, `hesitent`, and `slideshow`. Each mode declares a provided `frameRate` value for the related quality property defined by the `VideoPlayer` component type. In the mode `fluent`, the minimum provided `frameRate` is defined as 25 Frames/s. Furthermore, requirements on quality properties of other hardware and software component types are defined.

```
contract VLCMediaPlayer implements VideoPlayer {
  mode fluent {
    //required hardware components
    requires resource CPU {
      max cpuLoad = 50 percent
      min frequency = 2 GHz
    }
    requires resource Network {
      min bandwidth = 10 MBit/s
    }
    //required software components
    requires component Decoder {
      min dataRate = 50000 Bit/s
    }
    //provided qualities
    provides min frameRate 25 Frames/s
  }
mode hesitent { ... }
mode slideshow { ... }
}
```

Listing 4.1 Example contract for the *VLC media player* implementation of the component type *video player.*

Regarding the architectural requirements defined in Section 4.3, CCM and QCL fulfill the first two defined design time requirements. Thus, the structure and variation model of CCM enables the precise declaration of software and hardware components in a unified modeling language, while dependencies between qualities of the component types are declared as contracts in QCL. However, the third design time requirement to model and manage tenant configurations is not yet supported in CCM and QCL. Extensions to the architecture are explained in Section 4.5 to address this requirement.

Beside the described design time part containing CCM and QCL, MQuAT comprises a runtime part, which contains the quality-aware runtime environment THEATRE which is described in the following.

4.4.3. Quality-Aware Runtime Environment

As explained before, THEATRE is a quality-aware runtime environment for self-optimizing systems and part of MQuAT. The environment implements a control loop including the phases collect, analyze, decide and act [DDF+06]. In the first phase, information about the current state of the system is collected. In THEATRE hierarchically structured resource managers apply profiling techniques to collect information about component instances. The collected information is analyzed in the second phase, while the results of the analysis are evaluated in the third phase to decide how to proceed and draw up a plan. The analysis in THEATRE is conducted by evaluating QCL contracts and decisions are made by negotiating contracts to find the optimal system configuration and define an adaptation plan. Contract negotiation is implemented as a Constraint Satisfaction Optimization Problem (CSOP). A CSOP generator is part of the global control loop manager to generate the optimization problem. A CSOP solver is applied that solves this optimization problem leading to an target configuration plan by differentiating current and requested system state. The resulting plan is eventually executed in the fourth phase of the control loop. Hence, in the act phase in THEATRE, components are migrated by applying a component-based reconfiguration and adaptation technique [Göt13].

Three layers are distinguished in THEATRE according to the component types *User*, *Hardware*, and *Software* defined in the CCM structure model. Figure 4.4 visualizes the three layers. Both, hardware and software layer contain hierarchically structured component managers. These managers observe the system and provide information about existing components and their properties. *Resource managers* monitor hardware components on the hardware layer and control them. For instance, a resource manager is capable of shutting down a server, while another is responsible for scaling the frequency of a CPU. The topmost manager in the hierarchy of resource managers is called *global resource manager*. On the software layer, *control loop managers* are aware of available software component types, their implementations and their mapping to hardware components. Hence, control loop managers and resource managers exchange their knowledge about the current system. The topmost manager in the hierarchy of control loop managers is called *global control loop manager*. In addition, control loop managers implement the control loop described before.

On the user layer, *context analyzer* collect information about user requests in terms of functionality and quality. For instance, users playing videos from the video information system explained in Section 3.1 request different video resolutions according to their device. A user accessing the system with a smart phone has a limited network bandwidth and requires the video to be played in a smaller resolution than a different user with a desktop computer. Such information about user requests are collected by context analyzer and provided to the global control loop manager to optimize the system accordingly.

Regarding the requirements for an appropriate architecture for reconfigurable cloud applications defined in Section 4.3, the THEATRE runtime environment supports some of the runtime

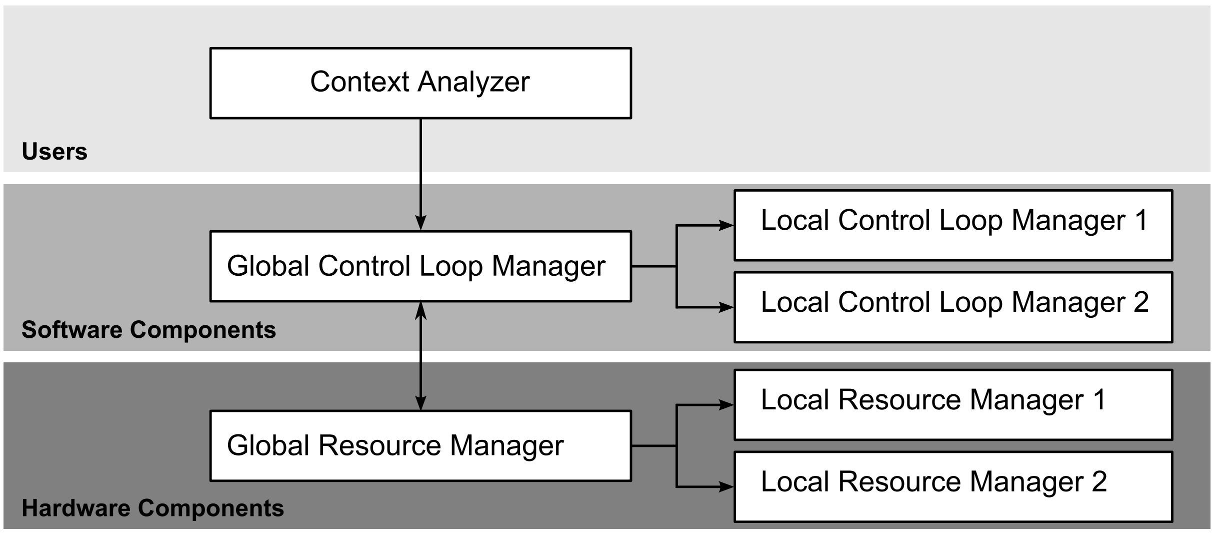


Figure 4.4 Layers of the runtime environment THEATRE and related component managers.

requirements. The first and the second runtime requirement are addressed by providing a self-optimizing runtime environment implementing a control loop and monitoring the qualities of the system components by means of component managers. However, the current implementation of THEATRE is intended for optimizing user requests without constraining tenant requirements.

Summarizing, the MQuAT architecture comprising the design time specifications CCM, QCL, as well as the runtime environment THEATRE addresses four out of seven requirements identified for a flexible architecture for reconfigurable cloud applications. Hence, in the next section extensions to the architecture are proposed to address the remaining requirements with respect to multi-tenant constraints on functionality and resources.

4.5. Architectural Multi-Tenancy Extensions

The current MQuAT architecture is not multi-tenant aware. To support multi-tenancy, customer management and the modeling of customer constraints on functionality and resources are to be integrated. Hence, the structure and variation model definition of CCM are extended with multi-tenancy concepts to model tenant configurations, and the runtime environment THEATRE is extended to manage tenants at runtime. Moreover, multi-tenant-specific platform services are required to identify a user and the affiliated tenant and to compute to which components a user has access. Hence, platform services such as access control and a multi-tenant aware data base are to be realized as mandatory components of an application. In addition, modeling these services as software components enables their reuse in different applications. However, state of the art frameworks are applicable and are therefore not further discussed in this approach.

In the following the required extensions to CCM are explained in detail followed by a discussion of the extensions to the runtime environment THEATRE and a proposal how to express customer constraints regarding functionality and quality [SCG$^+$12].

4.5.1. Multi-Tenant Extensions of the Cool Component Model

A customer defines constraints on component types and instances in terms of functionality and qualities in a reconfigurable cloud application. To model this information in CCM, structure and variation model need to be extended.

The extended CCM structure model specification to model customer configurations is depicted in Figure 4.5. Introduced metaclasses and relations to support multi-tenancy are highlighted with a gray background while metaclasses of the original structure model are white. A customer is represented by the metaclass `Tenant` defining a unique identifier of the corresponding customer, while the metaclass `Tenant Configuration` models a particular customer configuration. Customers can have multiple configurations expressed by the relation with a many multiplicity between both metaclasses. Customer constraints regarding application functionality and qualities are represented by the metaclass `Tenant Constraint` contained by a `Tenant Configuration`. A reconfigurable multi-tenant aware application is modeled by the metaclass `Application`

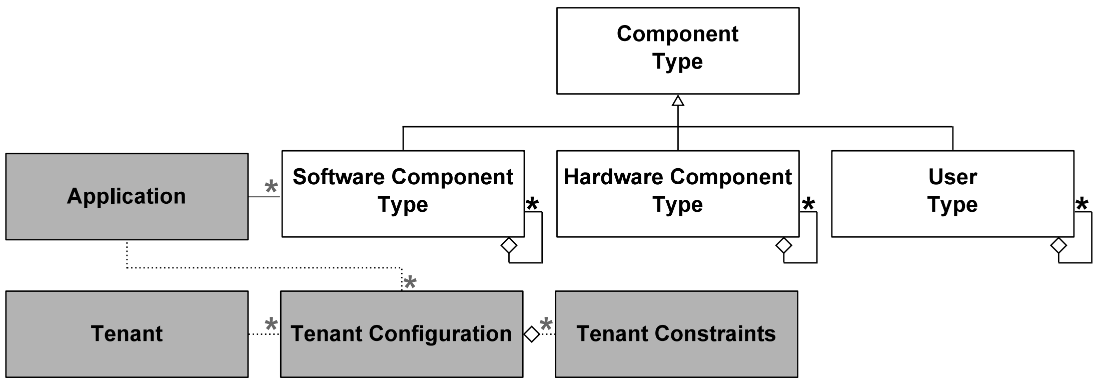


Figure 4.5 Multi-tenant extensions of the cool component model.

which refers to multiple mandatory and optional `Software Component Types`. The concept that multiple customers share the same application instance is expressed by the relation between `Application` and multiple `Tenant Configurations`.

Modeling the example of a video information system in the extended CCM structure model, an `Application` represents the video information system. In addition, the component types of the system shown in Figure 3.1 as well as their ports and connections are defined by means of `Software Components` and referenced by the `Application`. Both customers *A* and *B* are modeled as `Tenants` with a unique identifier. For each customer a `Tenant Configuration` is specified referencing the corresponding `Constraints`. Customer *A* only requires the mandatory components, but not the optional. Assuming customer *A* to have further quality requirements. As explained before, the customer requires strong encryption, high application availability, and the application to run only on servers located in the European Union due to legal restrictions. Functional and quality requirements are expressed as `Tenant Constrains`. The same holds for the requirements of customer `B`. These constraints can be expressed by means of OCL constraints, as discussed in Section 4.5.2.

However, which component types are optional and mandatory is implicitly given by a feature model in the problem space as illustrated in Figure 3.5(a) for the video information system. Hence, this information does not need to be modeled explicitly in CCM.

Furthermore, customers can change their application configuration according to their changing requirements. If a customer changes a configuration, a new customer related `Tenant Configuration` is defined while the current tenant configuration is declared to be obsolete.

Moreover, the variation model is extended accordingly as it defines instances for types in the structure model. Variation models express the runtime state of a system. Hence, for a multi-tenant application, they must be enabled to express currently active tenant configurations. The extended metamodel of multi-tenant aware variation models is depicted in Figure 4.6. The metaclasses `Tenant Context` and `User Context`, as well as further references are introduced to model multi-tenant access on functionality and resources at runtime. These multi-tenancy extensions are highlighted in the figure with a gray background and dotted lines while model elements of the original variation model are white. Referenced elements from the structure model and QCL are highlighted by dashed lines and dashed frames. These elements are additionally

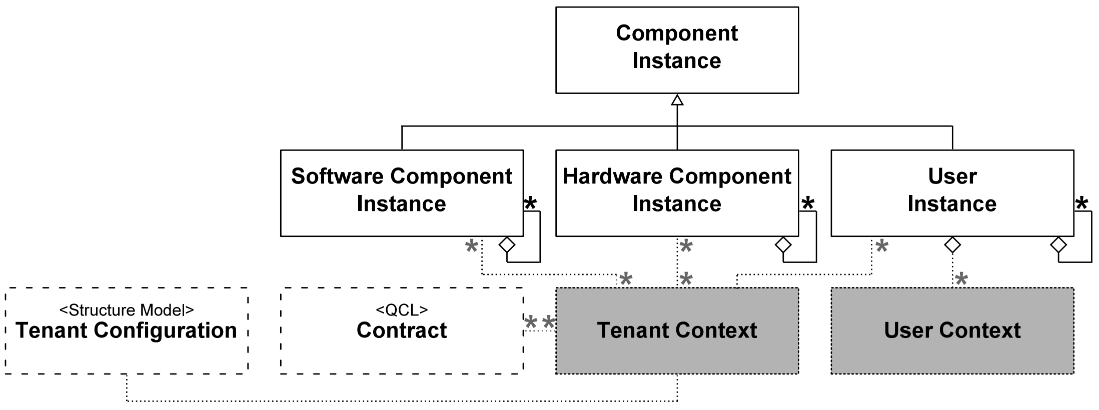


Figure 4.6 Multi-tenant extensions of the metamodel for CCM variation models.

marked with an annotation similar to a UML-profile to depict their origin. Each `User` has potentially multiple `User Contexts`. A `UserContext` defines the functional and quality related user requirements. The metaclass `Tenant Context` is introduced to model instances of `Tenant Configurations` defined in the structure model at runtime. `Tenant Constraints` of a `Tenant Configuration` are evaluated at runtime to define subsets of available component types, instances and related quality contracts. Quality contracts implying unavailable software implementation are filtered. Hence, a `Tenant Context` is related to `Users` currently accessing the system, as well as a subset of available `Software Component Instances`, `Hardware Component Instances`, and `Contracts`. Those referenced subsets are subsequently applied for negotiating contracts in the decide phase of the control loop in the runtime environment.

A `Tenant Context` for a customer references only `Hardware Component Instances` that fulfill the customers quality requirements. According to the video information system example in Section 3.1, the `Tenant Context` of customer *A* only contains highly available servers located in the European Union and offering strong hardware encryption. In addition, referenced `Software Component Instances` of this `Tenant Context` are implementations of the mandatory `Component Types` without implementations of optional `Component Types`. For customer *B* no constraints in server locations are defined, and therefore the referenced `Hardware Component Instances` are not filtered. Although a `Tenant Context` constrains the available instances for a `User` of this tenant, which particular instances are chosen by `Users` depends on the user's requirements defined in the corresponding `User Context`. If a user requests a video in this example, the user specifies a video frame rate which is stored in the `User Context` and subsequently considered in the decide phase of the control loop.

Moreover, if a tenant configuration is declared obsolete while an updated tenant configuration is active, the `Tenant Context` is updated accordingly. However, requests of users are evaluated according to the new `Tenant Context`.

Summarizing, the proposed extensions of the CCM structure and the variation models meet the requirement of concisely managing tenants, users, their requirements, and reconfiguration changes of their requirements as defined in Section 4.3.

4.5.2. Tenant Constraints

The requirements of customers must be expressed in a convenient constraint language. Requirements can be separated into functional and quality related requirements, both constraining component types and instances defined in CCM. Functional requirements directly refer to software component types and component implementations, such that a customer directly requires or excludes a software component type or implementation.

Functional constraints on software component types and implementations are expressed by means of propositional formulae [Bat05]. The definition of available and unavailable components can be specified in the following ways

- *subtractive* by assuming that always all components are contained while a constraint language explicitly declares excluded components,
- *additive* by assuming the opposite that no components are contained except the ones that are explicitly specified by a constraint language, and
- *explicit* by defining the containment state explicitly per component.

The first two approaches are handy as most of the information about available components is given implicitly, although the third approach is most explicit. In this work, the first approach is applied, as it is more natural for customers to define, which functionality should not be included than vice versa [Kru13].

The customer requirements are instantiated as `Tenant Constraint` in the structure model, as explained in Section 4.5.1. Three types of constraints can be defined, which are constraints on software component types, constraints on software component instances, and quality constraints. How to express these constraints by means of OCL [OMG2012] is explained in the following.

Constraints on Software Component Types

Customers can define constraints on software component types. For instance, in the video information system example, customer *A* directly excludes the optional component types `Water marker`, `Subtitle`, and `Video manager` from the related tenant context, while requiring the mandatory component types `Video player`, `Stream processor`, `Decoder`, and `Data provider`. Hence, an OCL expression for component types is defined and one constraining component implementations. For customer *A* in the video information system example, the constraint on component types could be defined, as depicted in Listing 4.2.

```
context Application inv: self.SoftwareComponentTypes-> select(c| not(c.name='
   Water marker') or not(c.name='Subtitle') or not(c.name='Video manager'))
```

Listing 4.2 Component type constraints expressed in OCL.

Hence, only excluded component types are specified in this expression. Evaluating this constraint leads to a subset of available software component types and also excludes the implementations of unavailable component implementations.

Constraints on Software Component Instances

If customer A further constraints the implementations of the required component types, such that the `AVS video player` implementation of the `Video player` component type should not be available, the related OCL constraint is depicted in Listing 4.3.

```
context VideoPlayer inv: self.Implementations-> select(not(name='AVS video
    player'))
```

Listing 4.3 Component instance constraints expressed in OCL.

Hence, the specified OCL constraint in this listing selects all component implementations except the `AVS video player` implementation.

Constrains on Quality Properties

Quality requirements constrain software and hardware component types and corresponding instances. These constraints are specified with respect to component properties. Hence, the particular constrained components are computed by evaluating the constraints. For instance, customer A has the quality requirements that only servers located in the European Union are utilized with a strong encryption and high availability. To express this constraint, a `Server` component type must be specified in the system with the properties `Location` and `Availability`. Furthermore, encryption components are required on each involved server with a long security key as a parameter of the encryption component. The quality related constraint of customer A is expressed in OCL, as depicted in Listing 4.4.

```
context Server inv: self.Location ='EU'
context Server inv: self.Availability ='high'
context Server inv: self.ComponentTypes-> exists(c | c.Type = 'Encryption' and c
    .SecurityKey = 'long')
```

Listing 4.4 Quality constraints expressed in OCL.

The OCL constraint on qualities is evaluated to filter hardware and software component instances. Depending on the properties defined in a system, further quality constraints can be expressed. In general, a customer is not required to specify any of the constraint types. For instance, customer B in the video information system example only specifies component type constraints.

Summarizing, the definition of tenant constraints in OCL addresses the design time requirement identified in Section 4.3 of modeling tenant constraints.

4.5.3. Multi-Tenant Extensions of the Runtime Environment

As explained before a tenant configuration defines constraints on component types and instances. These constraints are evaluated at runtime in THEATRE. Therefore, the runtime environment is extended to restrict the search domain of the contract negotiation process according to tenant constraints. The CSOP generator has to be extended to evaluate the tenant constraint prior to contract negotiation in the decide phase of the control loop. Constraints are evaluated sequentially starting with the constraint on component types, followed by the constraint on implementations, and finally by evaluating the OCL quality constraint. Hence, hardware and software component types and their instances are pre-filtered and fewer constraints are to be defined in the CSOP for negotiating contracts accordingly. However, the evaluation of quality constraints leads to mandatory components reflected by additional constraints in the CSOP.

In a multi-user scenario, the global control loop manager has to take resource sharing into account in the decide phase and their effects on provisioned qualities. The other phases of the control loop, which are collect, analyze and act are to be changed accordingly.

Summarizing, the proposed extensions of the THEATRE runtime environment meet the runtime requirement of concisely managing tenants and evaluating user requests regarding tenant constraints as defined in Section 4.3.

4.6. Discussion

The proposed multi-tenancy extensions to the MQuAT architecture allow for constraining user access with respect to functional and quality related requirements of customers. The explained concepts support functional variability among customers on architectural level and restrict user access on customer-specific data and functionality accordingly. Hence, the extended MQuAT architecture can be integrated into an SPL-based configuration management approach.

In a multi-tenant aware application, further requirements on resource sharing and multi-user contract negotiation arise, but are considered out of scope of this work. As explained in Section1.5.5 resource sharing is a central element of a multi-tenant application architecture. The user access must be isolated and users should not influence each other. These requirements are not particularly multi-tenant specific, but rather related to multi-user environments, as discussed in Section 1.5.5. For instance, to identify a user and the corresponding tenant, each software component type contains a parameter `user id`. This identifier is used to resolve the related `tenant id` from the database storing user and tenant affiliations. Furthermore, this identifier is applicable to access user- and tenant-related data in the system, as explained in Section 1.5.5. Resource sharing can be achieved by state-of-the-art frameworks for session management and load-balancing mechanisms.

Required platform and management services such as authorization, authentication, a multi-tenant aware data base, and session management are required at runtime, as explained in Section 4.2. These services can be modeled as MQuAT component types and components.

A current limitation of the MQuAT architecture is that contract negotiation is not yet multi-user aware and hence cannot handle the request of multiple users at the same time. Thus, contract negotiation in a multi-user scenario is not yet covered. For a robust and comprehensive PLA for reconfigurable cloud applications further research is required. The explained MQuAT extensions focus on how to model customer-specific constraints in the solution space. As explained before, multi-tenancy comprises two concepts, resource sharing as in multi-user environments, and access restriction on resources. The proposed extensions of MQuAT focus on the latter concept by restricting the access on functionality and resources of a single user of a customer by introducing tenant constraints. The explanations in this chapter show that self-adaptive architectures are in general applicable to implement reconfigurable cloud applications.

4.7. Demarcation from Related Work

Related to this work are approaches proposing architectures for variable multi-tenant aware cloud applications, as well as self-adaptive systems.

Koziolek provides an architectural style for implementing multi-tenancy as an extension of the multi-tier architectural style [Koz11]. In this approach, required components and connectors, as well as data elements of a multi-tenant architecture are described and further constraints are imposed on these elements. The concepts presented in this work complement the proposed MQuAT approach in terms of the definition of platform services, for instance to realize load balancing and the access to a multi-tenant aware database.

A component-based application architecture for configurable multi-tenant aware applications is proposed by Mietzner [Mie10]. In this work, multi-tenancy and variability descriptors are added to a service component architecture to allow for customization of an application. An application is configurable during design time choosing from different component types and implementations. Although a deployed application is multi-tenant aware, it is no longer configurable. Hence, all customers accessing the application are served by the same functionality. Furthermore, reconfiguration during runtime is not considered. In contrast, the architecture based on MQuAT comprising the multi-tenant extensions is configurable and reconfigurable at runtime.

Wang et al. define a service model for configurable cloud applications based on a service component architecture [WZL$^+$11]. In this approach, service dependencies are represented by hyper graphs. The authors propose algorithms to verify the dependencies between services at design time. The focus of this work is at design time, and variability at runtime is not considered.

In the DiVa research project, a framework for dynamic adaptive systems is developed [FS09, MBNJ09]. Systems defined by this framework are adapted automatically according to defined optimization goals. Furthermore, the system can be reconfigured with respect to qualities. Although the framework does not yet support multi-tenancy, an extension similar to the proposed extension of MQuAT could be possible. DiVa supports qualities with a logic-based optimization approach, whereas MQuAT in contrast supports numeric qualities and a sub-symbolic optimization. Hence, the optimization in MQuAT is more fine grained and more expressive.

In the MUSIC research project, another framework for dynamic adaptive systems is developed [GKR+06]. The focus in this research project is on defining a component model for self-adaptive applications running on mobile devices. The component model enables the definition of component instances and their distribution on currently available mobile devices with respect to qualities. MUSIC does not yet support multi-tenancy and in contrast to MQuAT, only mobile devices are supported whereas in MQuAT various user-defined devices can be modeled, such as servers, desktop computers, and mobile devices.

4.8. Summary

This chapter introduces a development method for reconfigurable cloud applications comprising different architectural concepts. Particular architectural requirements regarding reconfiguration and functional variability are identified. Main characteristics of reconfigurable cloud applications are explained subsequently, as well as architectural concepts implementing these characteristics. Based on the self-adaptive MQuAT architecture, a potential architecture for implementing reconfigurable cloud applications is discussed comprising the identified architectural concepts. Extensions to MQuAT are introduced regarding multi-tenancy constraints to restrict the access on variable functionality in the solution space. The concepts are illustrated by an example of a video information system.

As hardware and software components are modeled in MQuAT, customers specify quality constraints on both, software and underlying infrastructure. Architectures for variable multi-tenant aware cloud applications consider variability at design time not at runtime, while self-adaptive architectures handle variability at runtime, but are not multi-tenant aware. Hence, the architecture-based on MQuAT and multi-tenancy extensions regarding constraints on functionality and resources presented in this chapter bridges this gab.

This chapter shows that self-optimizing systems such as the multi-tenant aware MQuAT are convenient to implement reconfigurable cloud applications and address Requirement 1 defined in Section 2.8. An architecture such as the one proposed, can be uniformly integrated in an SPL-based approach to automate the derivation and instantiation of customer configurations, as well as their reconfiguration.

5. Multi-Perspectives Simplify Configurations

What we see depends mainly on what we look for.

— *John Lubbock*

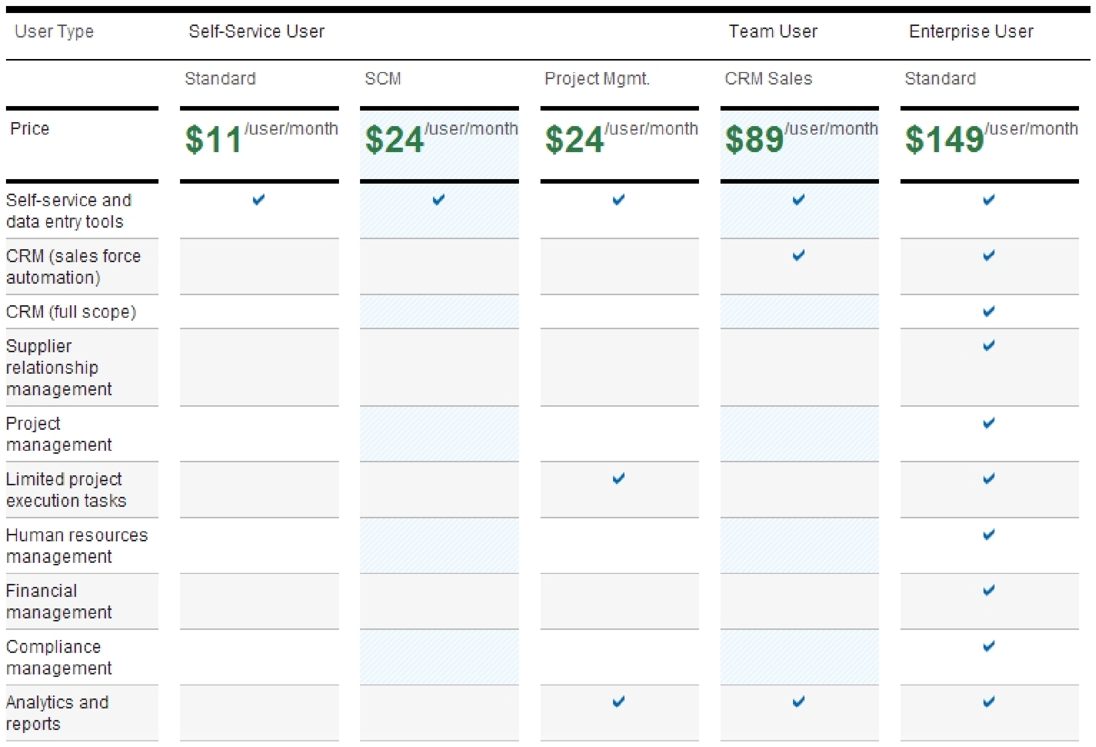
The concepts presented in this chapter are published as a conference paper [SLW12b], a workshop paper [SLW12a], and a technical report [SLW11].

There is a strong demand to provide configurable applications in the cloud that are tailored by customers themselves in a self-service portal, as explained in Section 1.8. For complex applications with a high number of configuration parameters and relations among them, the concise handling of dependencies among configurations options is as important as the processing of configuration operations. Additionally, customers vary in their configuration concerns. While some customers want to configure almost every single application parameters on their own in a self-service-portal, others are interested in a pre-configuration with a typical selection of application features specific for a particular application area and a small amount of variability left. For economic reasons and to prevent customers from getting lost in configuration options, application providers offer pre-configured *editions* [Cha13]. Each edition is tailored to a certain application area containing related features.

Figure 5.1 shows a screenshot[1] of *Business ByDesign* offered as a configurable SaaS application by SAP AG. In this example, for novice self-service users, team users and advanced enterprise users corresponding editions are provided. The functionality of the editions is dedicated to supply chain management, project management and CRM purposes and the amount of functionality varies between application areas and user types. In Figure 5.1, different thematically related features are grouped in a *package* shown in the first row on the left side. For instance, the packages `CRM (sales force automation)` and `CRM (full scope)` group a different amount of CRM functionality. Grouping features in packages furthermore enable variable pricing strategies [NH02]. For instance, a popular business model called *Freemium* is based on feature packages, where basic functionality is offered for free and users are charged for premium functionality [MdlILG08]. Additionally, packages efficiently filter out non-relevant features, as well as features that are unavailable for a certain customer. Further motivation for feature packages are legal restrictions applying in several countries. As discussed in Section 1.5.3, not all features should be available in all countries.

However, a recent study on the configurability of cloud applications reveals that current solutions supporting packages and editions are developed ad hoc, each for a specific application purpose [Cha13]. Thus, adapting these approaches to other scenarios, as well as extending them to support further packages and editions is error prone and costly. Therefore, a generic approach is demanded for concisely modeling packages and editions with respect to constraints among configuration parameters.

[1] `http://www54.sap.com/pc/tech/cloud/software/business-management-bydesign/pricing.html`



User Type	Self-Service User			Team User	Enterprise User
	Standard	SCM	Project Mgmt.	CRM Sales	Standard
Price	$11 /user/month	$24 /user/month	$24 /user/month	$89 /user/month	$149 /user/month
Self-service and data entry tools	✓	✓	✓	✓	✓
CRM (sales force automation)				✓	✓
CRM (full scope)					✓
Supplier relationship management					✓
Project management					✓
Limited project execution tasks			✓		✓
Human resources management					✓
Financial management					✓
Compliance management					✓
Analytics and reports			✓	✓	✓

Figure 5.1 The cloud application *Business ByDesign* offers different editions with varying amount of pre-configured functionality.

This chapter introduces *multi-perspectives on feature models* and their formal semantics to provide a generic approach for defining packages and editions [SLW12b]. The approach constitutes a conservative extension of feature models providing constructs to declaratively specify concerns on feature models in a separate model and to define a concern hierarchy for overlapping concerns. Prior configuration, a set of arbitrary concerns is selected resulting in a pre-configured configuration space. Thus, unrelated configuration parameters are filtered and dependencies among configuration parameters are satisfied.

Hence, multi-perspectives formally specify packages and editions, as depicted in Figure 5.2. The insinuated feature model on the left side of the figure defines the configuration space by specifying configuration parameters and their dependencies. Views on the feature model are defined to reflect particular business and technological concerns by grouping features in packages. These packages are potentially overlapping because some configuration parameters belong to the multiple concerns. These views are then aggregated to form a perspective that refines the feature model and adheres feature model constraints. Thus, a perspective represents an edition tailored to a specific application area with potentially left variability.

Assuming that the variability of the *Business By Design* application is described by a feature model, the following views and perspectives can be defined. In the example in Figure 5.2, the overlapping views `Package Full CRM` and `Package Sales Force CRM` and other view depicted by three dots are aggregated in perspective `Edition Enterprise Standard`. Additionally, the views `Package Sales Force CRM` and `Package Analytics and Reports` are aggregated in the

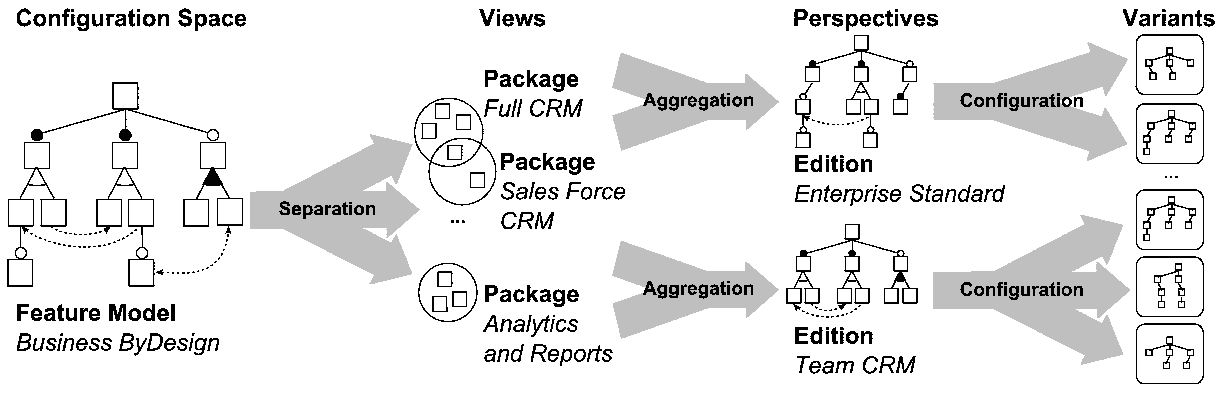


Figure 5.2 Views represent feature packages and perspectives assemble pre-configured application editions.

perspective `Edition Team CRM`. Hence, both perspectives contain a different set of features. The approach guarantees that perspectives are well-formed with respect to the feature model structure and adhere to feature model semantics. Although a perspective narrows the configuration space according to particular concerns, the approach ensures the derivation of valid variant configurations from a perspective.

Perspectives refine a feature model which addresses Requirement 2 defined in Section 2.8. Product variants are derived from the pre-filtered perspective instead of the original feature model. Thus, a perspective does not introduce new variant configurations, but rather bind a large amount of variability. Additionally, the filtering of unavailable features, prior to a customer configuration, reduces the effort for conducting feature constraint analysis during the configuration process [SOS$^+$12].

Multi-perspectives further support customized application features on feature model level due to the view concept. If a customer of the *Business ByDesign* application requires customized CRM functionality, this functionality is modeled by additional features in the feature model which are assigned to a customer-specific view. This customer-specific view is only aggregated in perspectives of that particular customer. Hence, views restrict the availability of features. Especially the ability to model customizations on feature model level is important for configurable applications in the cloud as the explicit knowledge of customized features allows for making future market decisions. For instance, if other customers require the same functionality, customer-specific functionality becomes public available.

The correctness of the proposed approach with respect to derivable variant configurations is proven in this chapter and an efficient algorithm for verifying the consistency of the overall multi-perspective model is provided.

5.1. Views Separate Concerns

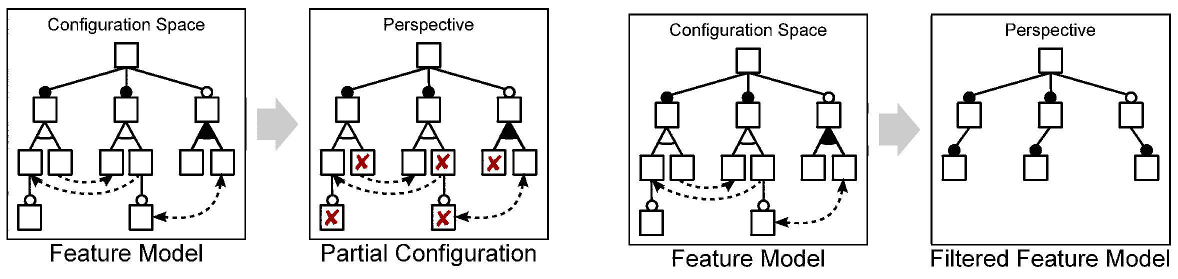
Views on feature models are an established method to separate concerns and express further feature relations additional to the dependencies defined by a feature model [ACLF12, HHS$^+$11, CHE05a]. Multi-view approaches on feature models usually partition feature models to different stakeholders and assume a one-to-one correspondence between a set of views on a feature model and a set of *distinct* stakeholders [CHH09, WDS09, HHS$^+$11]. Views are considered to encapsulate a particular concern and multi-view approaches on feature models aim at Separation of Concerns (SoC). The methodology of SoC was first mentioned in the 1970s [Par72, Dij76] assuming that a system unit is decomposable into smaller parts according to certain hierarchical concerns.

The paradigm of SoC is to focus on one aspect of a system at a time to cope with system complexity. Following this paradigm, the notion of *view* was introduced to describe a cross-cutting view on a system-based on a specific interest [FKN$^+$92, NKF03]. Similarly, the *ISO/IEC/IEEE 42010:2011, Systems and software engineering standard*, as the successor of the standard *IEEE 1471*, defines a view as a "representation of the whole system from the perspective of a related set of concerns" [ISO 42010]. Conferring this methodology to feature-based SPLs, a view decomposes a feature model into a set of features that belongs to a certain concern. In this standard, a view is explicitly separated from the notion of a viewpoint. A view is defined as, *what can be seen*, and a viewpoint is defined as, *where to look from.* This separation is picked up in this work as well, such that a viewpoint explicitly defines a view that consistently filters a feature model. Such a view is called a *perspective*.

5.2. Perspectives Reduce the Configuration Space

A perspective is a semantic refinement of the feature model, and thus, reduces the configuration space defined by the feature model. Therefore, a perspective implicitly decreases the number of derivable products. Hence, at least a single valid product of the original feature model is derivable form a perspective. Figure 5.3 illustrates that a perspective can be perceived as a *partial configuration* and as an *explicit filter* of the original feature model, both leading to a reduced configuration space. The difference between both perceptions is that in a partial configuration all features of the original feature model are contained in a perspective with differing configuration states, as exemplified in Figure 5.3(a), whereas in a filtered perspective, configuration states are untouched, but some features are removed, as depicted in Figure 5.3(b).

On perceiving a perspective as a partial configuration of a feature model, the configuration states of features contained in the perspective are left undecided, where non-contained features are explicitly deselected. In contrast, when perceiving a perspective as a filter of the feature model, a perspective yields a new feature model, where features not contained in the perspective are removed. The latter one is applicable for scenarios, where the feature model describes the product portfolio of a company and a single configurable product is represented by such a filtered feature model. However, perspectives semantically refine feature models by aggregating multiple views and express different stakeholder concerns in the domain engineering process of an SPL. Later on, in the application engineering process, a perspective is derived from the multi-perspective



(a) Features not included in a perspective are deselected in the resulting partial configuration.

(b) Features not included in a perspective are removed from the feature model.

Figure 5.3 A perspective reduces the configuration space and is assumed as an explicit deselection of features, and a feature model filter.

model in a pre-configuration step prior the configuration of particular variants. To illustrate how to apply multi-perspectives in a practical scenario, an example of customizable document management systems is introduced in the next section.

5.3. Illustrative Example for Multi-Perspectives

A typical document management system comprises several features regarding document types, indexing and searching capabilities. An example SPL for document management systems is modeled as a feature model illustrated in Figure 5.4 on the right side. The root feature represents the document management system application and is therefore named `DocumentManagement`, while the child features depict variable functionality. Different search and indexing mechanisms for documents are modeled as features. The management system handles various document formats, such as plain text, Portable Document Format (PDF), and image types and related Object Character Recognition (OCR) functionality for extracting and indexing text stored in pictures. The feature model comprises 23 features and defines 208 derivable variant configurations.

A typical business concern for such information systems is to apply a variable pricing strategy by selling features in packages at different prices. The pricing strategy is modeled in the view model depicted on the left side of Figure 5.4. In this scenario, features are grouped in a basic and a premium line represented by corresponding viewgroups *Premium* and *Basic*. In the premium line there are additional feature packages available depicted by the viewgroups *Silver* and *Gold*. The viewgroup *Core* represents core features available in both pricing lines. Features mapped to these viewgroups are highlighted by the same shading. For instance, core functionality is highlighted with a light gray color comprising 9 features. The function $\sigma Core$ defines the view of core viewgroup containing the features `DocumentManagement`, `DocumentType`, `TextType`, `Indexing`, `GeneralIndex`, `FileNameIndex`, `Search`, `GeneralSearch`, and `FileNameSearch`.

As explained previously, features are related to potentially multiple concerns and therefore are contained in various views. For instance, the feature `ImageType` is contained in the view depicted by light gray stripes dedicated to viewgroup *Basic* and to the view depicted by the light gray check pattern dedicated to viewgroup *Gold*. The viewgroup *Customized* is a singleton viewgroup

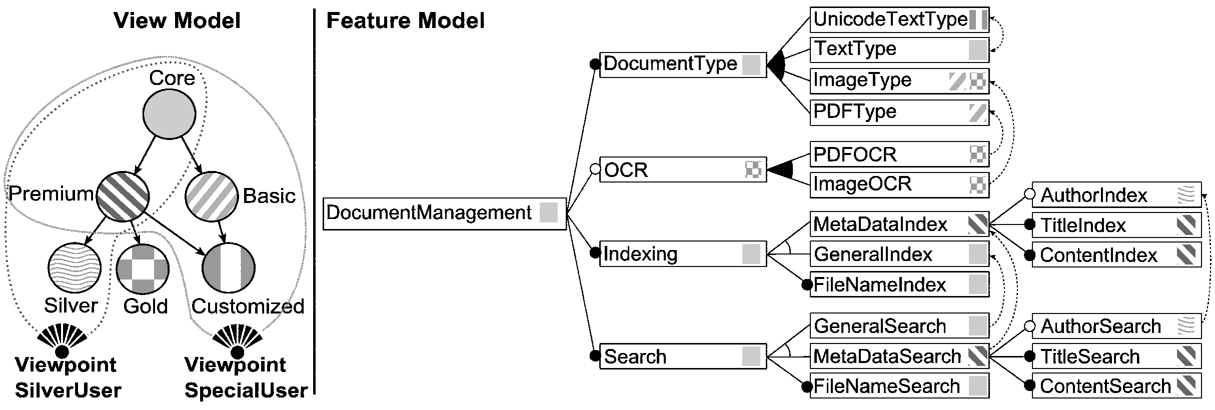


Figure 5.4 A multi-perspective model for a sample document management product line.

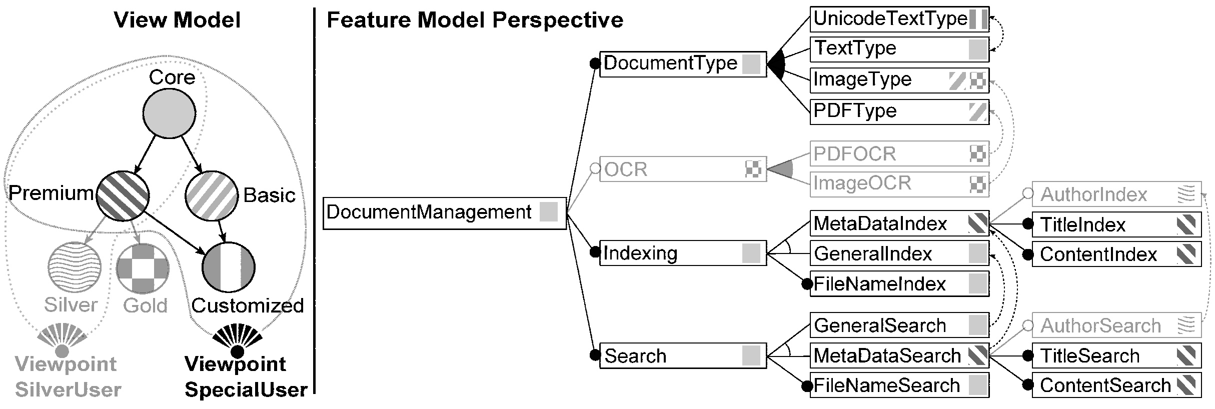


Figure 5.5 The perspective for viewpoint *SpecialUser* filters features and constraints unrelated to this viewpoint.

holding customized features for the user named *SpecialUser* according to the explanation in Section 5.13. In this case, the customized feature is `UnicodeTextType` as a replacement for the standard `TextType` feature. This feature is restricted to the viewpoint of user *SpecialUser* due to the viewpoints singleton viewgroup and thus not available for other users.

Currently, there are two viewpoints defined in the view model, one for the *SpecialUser* and one for a *SilverUser* shown in the bottom of the view model and named accordingly *Viewpoint SilverUser* and *Viewpoint SpecialUser*. A viewpoint in this scenario represents a pre-configuration of the feature model according to a pricing strategy. For instance, the *SpecialUser* is interested in buying features from the premium line and from the basic line, while requesting a customized feature as well. The user's viewpoint *Viewpoint SpecialUser* reflects this in the view model. This viewpoint is therefore related to the viewgroups *Core*, *Basic*, *Premium* and *Customized*, which are framed by a thin gray line in the figure.

A user cannot select features that are solely mapped to the viewgroups *Silver* and *Gold*. The features `OCR`, `PDFOCR`, and `ImageOCR` are unavailable to this user, as they are mapped to viewgroup *Gold*. In addition the features `AuthorIndex` and `AuthorSearch` ae unavailable as well, as they are mapped to viewgroup *Silver*. Furthermore, features which are solitary mapped to the viewgroup *Gold* are not referenced by any viewpoint. Hence, the features `OCR`, `PDFOCR`, and `ImageOCR` are not derivable in any variant configuration. These features are currently dead. They become alive when a viewpoint is added that is related to the viewgroup *Gold*.

Each viewpoint defines a *perspective* that refines the feature model and thus narrows the configuration space by filtering unrelated features and constraints. Instead of directly deriving a variant configuration for the user *SpecialUser* from the original feature model, a filtered perspective is instantiated in a pre-configuration step, as depicted in Figure 5.5. Thus, only features a user is interested in are available for selection in the pre-configured perspective. For the user *SpecialUser* the perspective defined by *Viewpoint SpecialUser* contains 18 features out of originally 23 as shown in the figure. Filtered features and viewgroups are grayed out. Hence, features representing OCR functionality and index and search of document author metadata, as well as corresponding cross-tree constraints are filtered from the resulting perspective. However, still 55 variant configurations are derivable from the resulting perspective.

As the meaning of views and perspectives is ambiguous, the next section provides an overview of the applied terminology in the multi-perspective approach.

5.4. Multi-Perspective Terminology

The multi-perspective approach uses the notions of views and perspectives common in various areas of computer science. However, their meanings differ depending on the application context. Figure 5.6 illustrates the terminology applied in the multi-perspective approach.

A multi-perspective model *MP* is visualized by a gray background in the figure comprising a view model *VM* depicted on the left side and a feature model *FM* on the right side. In addition,

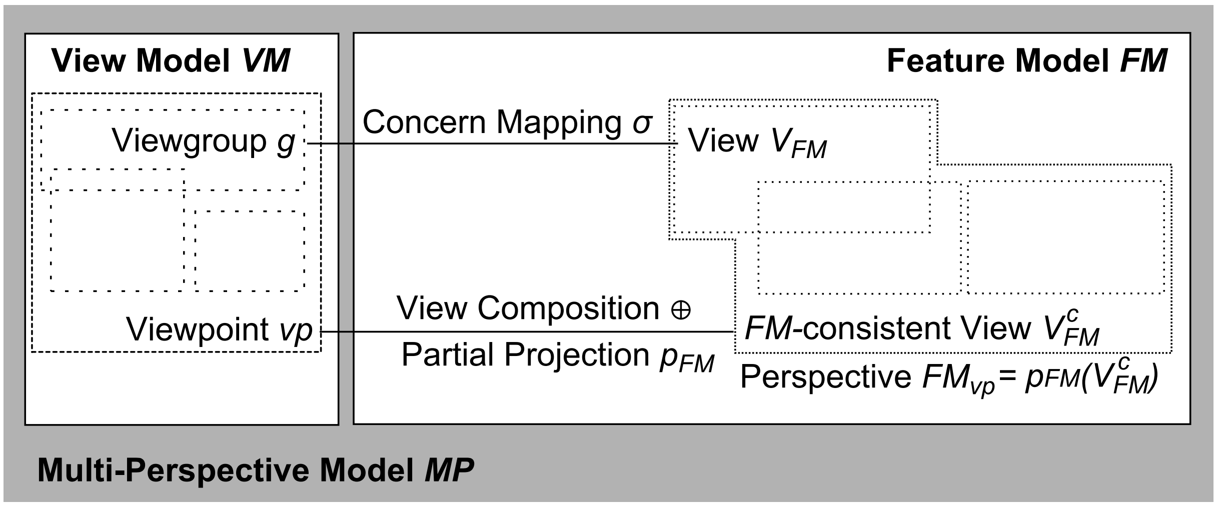

View Model VM
Feature Model FM
Viewgroup g
Concern Mapping σ
View V_{FM}
Viewpoint vp
View Composition ⊕
Partial Projection p_{FM}
FM-consistent View V^c_{FM}
Perspective $FM_{vp} = p_{FM}(V^c_{FM})$
Multi-Perspective Model MP

Figure 5.6 Relation between the terminology applied in the multi-perspective approach.

a multi-perspective model contains a mapping σ between viewgroups g of the view model and views V_{FM} in the feature model.

Views on the feature model are defined on a set of features and related feature constraints and are framed in the figure on the right side by a dotted line. Views on the feature model can overlap depicted by overlapping frames. In addition, viewgroups are highlighted by a dotted frame on the left side of the figure and overlap as well. A viewpoint is related to a set of viewgroups visualized in the figure by the striped line surrounding all three abstracted viewgroups. Thus, the views on the feature model related to those viewgroups are aggregated applying the composition operator $\oplus$, whose semantic is defined in Section 5.7.

The result of the view composition must be an *FM*-consistent view V^C_{FM} obeying feature model constraints. Applying the partial projection function $p_{FM}(V_{FM})$ on the resulting view V_{FM} leads to a perspective FM_{vp} which is a feature model with a smaller configuration space than defined by the original feature model *FM*. In addition, the terminology applied in the multi-perspective approach is summarized in Table 5.1.

Table 5.1 Terminology of the multi-perspective approach.

Terminology	**Description**
Multi-perspective model *MP*	is the combination of a feature model, a view model, and a mapping between views on the feature model and viewgroups in the view model. For instance, the combination of the document management feature model and the business concern-related view model in Figure 5.4.
Feature model *FM*	defines the configuration space in terms of a feature hierarchy and cross-tree constraints. In the example in Section 5.3, the feature model defines the configuration space of document management systems.
View V_{FM}	is a concern-specific projection on a feature model containing a set of arbitrary features and related cross-tree constraints. A view groups conceptually similar features due to a specific concern. Hence, views are potentially overlapping as a feature can belong to various concerns. In the document management system example in Figure 5.4, views are represented by the shadings in the feature model. Hence, features with the same shading belong to the same view.
View model *VM*	defines hierarchical relations over viewgroups to express overlapping concerns and captures the dependencies between viewpoints and viewgroups. In the example in Figure 5.4, the view model depicted on the left side defines a hierarchy among business related concerns.
Viewgroup *g*	is a node in the view model and hierarchically related to other viewgroups. A viewgroup is associated with a view on the feature model and represents a specific concern. In the document management example in Figure 5.4, the `Premium` concern is expressed by the corresponding viewgroup, while features of the feature model related to this concern are highlighted with a dark gray striped shading.

Continued on next page

Table 5.1 – continued from previous page

Terminology	Description
Viewpoint vp	is defined in the view model and is related to a set of viewgroups representing concerns. An FM-consistent view on the feature model results from the aggregation of views associated to these viewgroups. Applying the projection function p_{FM} on the resulting FM-consistent view leads to a perspective. A viewpoint is a set of concerns specifying a partial representation of a feature model by means of a perspective. In the document management example in Figure 5.4 two viewpoints `SilverUser` and `SpecialUser` related to different customers are defined.
Perspective FM_{vp}	is a semantic refinement of the original feature model constituting a feature model with a reduced configuration space. A perspective is defined by a viewpoint and results from a projection of the aggregated viewpoint related views. In the document management example in Section 5.3, two perspectives are derivable, one per viewpoint. Figure 5.5 shows the perspective on the right side defined by viewpoint `SpecialUser`. Features which are filtered out in this perspective are grayed out in the feature model representation.

This approach defines perspectives and views on group-cardinality based feature models, which are formally introduced in the next section.

5.5. Feature Models with Group-Cardinality

Various feature modeling techniques exist, as explained in Section 2.3. However, in the multi-perspective approach, feature models with group-cardinality, as well as require and exclude cross-tree concerns are considered, as depicted in Figure 5.7. The feature model shown in the figure contains 15 features, 3 cross-tree constraints on features and, thus defines 136 variant configurations. The abstract syntax of such feature models is formally defined as follows.

Definition 5.1 (Group-Cardinality Based Feature Model). *A* feature model *FM with group cardinality is a 4-tuple* $(F, \prec, \lambda, \Phi)$*, where* F *is a finite set of* features*,* $\prec \subseteq F \times F$ *is a decomposition relation on* F*,* $\lambda : \mathcal{P}(F) \rightharpoonup \mathbb{N}_0 \times \mathbb{N}_0$ *is a partial* cardinality function *assigning intervals to feature groups, and* Φ *is a set of* propositional formulas *over* F *defining cross-tree constraints.*

The cardinality $\lambda(F') = (k, l)$ of feature groups $F' \in dom(\lambda)$ defines the minimum and maximum number of selectable features in a variant configuration. The root feature f_r is solely contained in the root feature group F_r. Hence, a group cardinality $\lambda(F_r) = (1, 1)$ is assumed. In general, the four common decomposition types for feature groups are *mandatory*, *optional*, *alternative*, and *or* feature groups [CE00]. They are represented by the following cardinality, where $n = |F'|$.

- $\lambda(F') = (n, n)$ for *mandatory* feature groups, such as $F' = \{f_1\}$ with $n = 1$ in Figure 5.7,

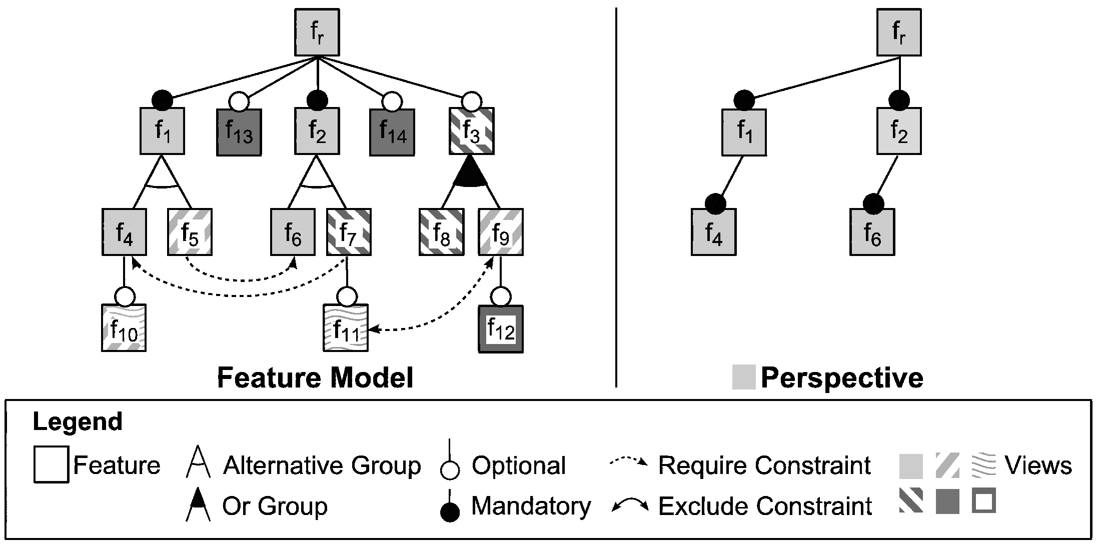


Figure 5.7 Views and a perspective on a feature model.

- $\lambda(F') = (0, n)$ for *optional* feature groups, such as $F' = \{f_3\}$ with $n = 1$ in Figure 5.7,
- $\lambda(F') = (1, 1)$ for *alternative* feature groups, such as $F' = \{f_4, f_5\}$ in Figure 5.7, and
- $\lambda(F') = (1, n)$ for *or* feature groups, such as $F' = \{f_8, f_9\}$ with $n = 2$ in Figure 5.7.

A feature model *FM* is *well-formed* iff it satisfies all of the following four conditions. $\mathcal{FM}(F)$ is the set of all well-formed feature models over the features in F.

1. The decomposition relation $\prec$ builds a *rooted tree* on all features in F. Hence, a unique root feature f_r exists, such that every other feature $f \in F \cap \{f_r\}$ has exactly one predecessing feature $f' \in F$ with $f' \prec f$.
2. The root feature f_r is solely contained in the singleton feature group F_r, and does not have a parent feature. All other features f' contained in feature groups $F' \in dom(\lambda)$ have the same parent feature f'' under the relation $\prec$, such that
   - $\forall F' \in dom(\lambda) \setminus \{F_r\} \mid \exists f'' \in F \, : \, f'' \prec F'$.
3. Each feature $f \in F$ belongs to exactly one feature group F' and feature groups are non-empty. In other words, F is fully partitioned by the domain $dom(\lambda)$ of the partial cardinality function λ, such that
   - $\emptyset \notin dom(\lambda)$,
   - $\forall F', F'' \in dom(\lambda) : F' \neq F'' \Rightarrow F' \cap F'' = \emptyset$, and
   - $\bigcup_{F' \in dom(\lambda)} F' = F$.

4. The cardinality $\lambda(F') = (k, l)$ of feature groups $F' \subseteq F$ define reasonable intervals for child features, such that $k \leq l$ and $l \leq |F'|$ holds.

In the following, semantics of syntactically well-formed feature models are explained with respect to the derivation of valid variant configurations.

5.5.1. Feature Model Semantics

The semantics of a feature model *FM* defines the set of all valid variant configurations. This set is commonly referred to as *variant space*. A variant configuration is a subset of selected features $F_{vc} \subseteq F$ and is *valid*, iff

- the root feature $f_r \in F_{vc}$ is selected in the configuration,
- for each selected feature $f'' \in F_{vc}$, its parent feature f''' is selected, such that $f''' \in F_{vc}$ defined by the relation $f''' \prec f''$,
- the cardinality $\lambda(F') = (k, l)$ of all feature groups F' is satisfied, such that $k \leq |F'_s| \leq l$, where $F'_s \subseteq F'$ and $F'_s \subseteq F_{vc}$,
- all cross-tree constraints Φ are satisfied on F_{vc}.

An example for a valid variant configuration of the feature model depicted in Figure 5.7 is $F_{vc1} = \{f_r, f_1, f_2, f_4, f_6, f_{10}\}$.

Cross-tree constraints $\phi \in \Phi$ over features in F are expressed by Boolean propositional formulas $\phi \in \mathbb{B}(F)$. A feature model is conceptually complete, if Φ only contains implications to express binary *require* and *exclude* constraints between two features f and f' [HST$^+$08].

- The require constraint is defined as $\phi_{rq} = f \rightarrow f'$.
- The exclude constraint is defined as $\phi_{ex} = f \rightarrow \overline{f'}$.

The set of all cross-tree constraints Φ is interpreted as the logical conjunction $\bigwedge_{\phi \in \Phi} \phi$.

The example feature model in Figure 5.7 comprises two require constraints, that are highlighted by a dashed directed arrow. Feature f_7 requires feature f_4, which is expressed by $f_7 \rightarrow f_4$, and feature f_5 requires feature f_6 which is expressed by $f_5 \rightarrow f_6$.

Additionally, the feature model comprises a binary exclude constraint between features f_9 and f_{11} visualized in the figure by a dashed bidirectional arrow. The exclude constraint is expressed by $f_9 \rightarrow \overline{f_{11}}$.

The semantical evaluation function

$$[\![\cdot]\!] : \mathcal{FM}(F) \rightarrow \mathcal{P}(\mathcal{P}(F))$$

defines the variant space by mapping well-formed feature models $FM \in \mathcal{FM}(F)$ over features F into the domain of sets of valid variant configurations $F_{vc} \in \mathcal{P}(F)$. The domain of sets of valid variant configurations is $[\![FM]\!] \in \mathcal{P}(\mathcal{P}(F))$. In addition, the semantical evaluation function $[\![\cdot]\!]$ defines the *maximum set* of valid variant configurations, such that

$$\begin{aligned}[\![FM]\!] =\{F_{vc} \in \mathcal{P}(F) \mid\ & f_r \in F_{vc} \wedge \\ & (f'' \in F_{vc} \wedge f''' \prec f'' \Rightarrow f''' \in F_{vc}) \wedge \\ & (f \in F_{vc} \wedge f \prec F' \wedge \lambda(F') = (k,l) \Rightarrow k \leq |\{f' \in F' \cap F_{vc}\}| \leq l) \wedge \\ & F_{vc} \models \bigwedge_{\phi \in \Phi} \phi\}\end{aligned}$$

where $f \prec F' : \forall f' \in F' : f \prec f'$. A variant configuration F_{vc} contains the root feature f_r. For each contained child feature f'', its parent feature f''' is contained, which is defined by $f''' \prec f''$. In addition, if a contained feature f comprises a child feature group F', the group cardinality constraint $\lambda(F')$ defining an lower and upper bound (k, l) for the number of contained features is obeyed. Furthermore, the variant configuration F_{vc} satisfies all cross-tree constraints Φ.

The variant space defined by $[\![\cdot]\!]$ of the feature model comprising 15 features in Figure 5.7 comprises 136 valid variant configurations. An example variant configuration of this feature model is $F'_{vc} = \{f_r, f_1, f_2, f_4, f_6, f_{10}\}$. In contrast, the set of feature $F'' = \{f_r, f_1, f_2, f_4, f_5, f_6\}$ is not a valid variant configuration. F'' is invalid, as the features f_4 and f_5 must not occur in the same configuration due to their relation in an alternative feature group.

A feature model *FM* is *satisfiable*, if at least one non-empty set of features $F_{vc} \subseteq F$ exist, that satisfies the feature relations defined in the feature model [HST$^+$08].

Definition 5.2 (Satisfiable Feature Model). *A feature model FM is* satisfiable*, iff* $[\![FM]\!] \neq \emptyset$.

Hence, a feature model is satisfiable, if the variant space is not empty. In Section 5.1, views on feature models are introduced informally. Views on feature models are formally defined in the next section, based on the formalization of a feature model *FM*, Any feature model *FM* considered in the following sections is assumed to be well-formed and satisfiable.

5.6. Views and Perspectives on Feature Models

A *view* $V_{FM} \in \mathcal{V}_{FM}$ is a projection from a feature model $FM \in \mathcal{FM}(F)$ to a subset of features $F_V \subseteq F$ and corresponding constraints $\Phi_V \subseteq \Phi$. Hence, a view is defined on an arbitrary set of features, which do not need to be related by hierarchical and cross-tree constraints. The set of all possible views on a feature model *FM* is referred to as $\mathcal{V}_{FM}$.

Definition 5.3 (Feature Model View). *A* view *of feature model $FM \in \mathcal{FM}(F)$ is a pair (F_V, Φ_V) that consists of a subset $F_V \subseteq F$ of* selectable *features, and a subset $\Phi_V \subseteq \Phi$ of* constraints*, such that $\phi_V \in \mathbb{B}(F_V)$ for each $\phi_V \in \Phi_V$.*

In Figure 5.7, six views on the feature model are highlighted via different shadings of the features. For instance, the light gray view comprises the five features f_r, f_1, f_2, f_4, and f_6. Features can be projected into multiple views such as feature f_{10}, which is contained in two views, one highlighted by a wavy line shading, and the other is highlighted by light gray oblique lines.

A *perspective* is a view that corresponds to a partial tree of the original feature model, which represents a variability-reduced feature model tree [SLW12b]. Hence, in a perspective feature model constraints remain satisfiable. In this example in Figure 5.7, the light gray view corresponds to a valid perspective depicted on the right side of the figure.

The set of valid feature models $\mathcal{FM}(F)$ comprises feature models $\mathcal{FM}(F_V)$ on feature subsets $F_V \subseteq F$, where $\mathcal{FM}(F_V) \subseteq \mathcal{FM}(F)$. Thus, a perspective $FM_V \in \mathcal{FM}(F)$ for a view $V_{FM} \in \mathcal{V}_{FM}$ is defined by the partial projection function

$$p_{FM} : \mathcal{V}_{FM} \rightharpoonup \mathcal{FM}(F)$$

where $p_{FM}(V_{FM}) = (F_V, \prec_V, \lambda_V, \Phi_V)$, such that the decomposition function $\prec_V \subseteq F_V \times F_V \subseteq \prec$ represents the *restriction* of $\prec$ on the feature subset F_V. Furthermore, the cardinality function λ_V is reduced to the feature subset F_V of the view as follows

- if features F' are contained in a feature group $F' \in dom(\lambda)$ and some of these features F_V are comprised in a view $F' \cap F_V \neq \emptyset$, then all features not contained in the view are filtered out of the feature group $F' \cap F_V \in dom(\lambda_V)$, and

- if the cardinality $\lambda(F') = (k, l)$ of the feature group has an upper bound l which equals the number of features contained in the feature group $l = |F'|$, then the upper bound is refined according to the reduced number of contained features $\lambda_V(F' \cap F_V) = (k, l- \mid F' \setminus \{F_V \cap F'\} \mid)$.

The projection function p_{FM} is partial, as views $V_{FM} \in \mathcal{V}_{FM}$ exist, where the application of p_{FM} yields a syntactically *ill-formed*, and therefore invalid feature model $p_{FM}(V_{FM}) \notin \mathcal{FM}(F)$. Moreover, views exist that yield syntactically well-formed feature models, such that $p_{FM}(V_{FM}) \in \mathcal{FM}(F)$ holds. The resulting perspective $p_{FM}(V_{FM})$ does not *semantically* refine the original feature model *FM*, such that more variant configurations are derivable from the perspective than from the original feature model, such as $[\![p_{FM}(V_{FM})]\!] \not\subseteq [\![FM]\!]$. Hence, the notion of *FM-consistent* views is introduced. A view V_{FM} is *FM*-consistent, iff

- the view is well-formed, such that $p_{FM}(V_{FM}) \in \mathcal{FM}(F)$,

- $[\![p_{FM}(V_{FM})]\!] \subseteq [\![FM]\!]$ that is a semantic refinement of the original feature model, and

- $FM_V = p_{FM}(V_{FM})$ is *satisfiable.*

The defined properties of an *FM*-consistent view lead to Lemma 1. The first property holds, if the feature selection F_V preserves the tree structure of original feature model *FM* and obeys feature group cardinality constraints. The second property holds, if cross-tree constraints are not violated. Hence, cross tree constraints not contained in the view $\phi \in \Phi \setminus \Phi_v$ only constrain features $F' \subseteq F$ that are likewise not contained in the view. Therefore, $F' \cap F_V = \emptyset$ is required. This property is weakened, as in this work only feature models with binary require and exclude constraints are considered. Thus, only require constraints must be satisfiable by the feature selection F_V of a view V_{FM}. In contrast, exclude constraints cannot be invalidated in a view. Exclude constrains are either fully supported as both features are present, or one of the excluded features is not present in the view.

Lemma 1. *A view $V_{FM} \in \mathcal{V}_{FM}$ on a satisfiable feature model FM is* FM-consistent*, iff*

- $f_r \in F_V$*,*
- *if $f \in F_V$ and $f' \prec f$, then $f' \in F_V$,*
- *if $f \in F_V$ and $f \prec F'$ with $\lambda(F') = (k, l)$, then $|F' \cap F_V| \geq k$ and*
- *if $f \in F_V$ and $f \rightarrow f' \in \Phi$, then $f' \in F_V$, and $f \rightarrow f' \in \Phi_V$.*

The subset of *FM*-consistent views on a feature model *FM* is referred to $\mathcal{V}^c_{FM} \subseteq \mathcal{V}_{FM}$. In contrast, non-*FM*-consistent views $V_{FM} \notin \mathcal{V}^c_{FM}$ are called *partial* views. For instance, features with a light gray shading in Figure 5.7 comprise an *FM-consistent* view that satisfies all conditions of Lemma 1. Hence, this view forms a perspective, as depicted on the right side of the figure. The other views in the figure are solely non-*FM*-consistent. However, potentially partial views can be aggregated into an *FM*-consistent view, as explained in the following.

5.7. View Composition

For aggregating various potentially non-*FM*-consistent views, the *view composition operator*

$$\oplus : \mathcal{V}_{FM} \times \mathcal{V}_{FM} \rightarrow \mathcal{V}_{FM},$$

is introduced. The composition of two views V_{FM} and V'_{FM} yields a view V''_{FM} by combining feature subsets and constraints of the composed views, such as

$$V''_{FM} = V_{FM} \oplus V'_{FM} = (F_V \cup F_{V'}, \Phi_{V''}).$$

However, potentially more cross-tree constraints $\Phi_{V''}$ are contained in the resulting view than resulting from the union of the constraints contained in both composed views. Hence, the number of constraints affecting the set of view features F'' is higher than the aggregated constraints of the composed views, such that $|\Phi_{V''}| \geq |\Phi_V \cup \Phi_{V'}|$. Thus, the resulting set of cross-tree constraints of the composed view is a subset of the constraints of the original feature model $\Phi_{V''} \subseteq \Phi$. A constraint ϕ is contained in the composed view V'', if both features are contained in this view, such that $\phi \in \Phi_{V''} :\Leftrightarrow \phi \in \mathbb{B}(F_{V''})$.

For instance, in Figure 5.7, the *FM*-consistent view highlighted in light gray V_{gray} contains five features and no binary constraints, such that $V_{gray} = \{\{f_r, f_1, f_2, f_4, f_6\}, \emptyset\}$. The non-*FM*-consistent view with dark gray stripes $V_{stripes}$ contains three features and no binary constraints, such that $V_{stripes} = \{\{f_3, f_7, f_8\}, \emptyset\}$. Composing these two views $V_{gray} \oplus V_{stripes}$ yields a new *FM*-consistent view $V_{composed} = \{\{f_r, f_1, f_2, f_3, f_4, f_6, f_7, f_8\}, \{f_7 \rightarrow f_4\}\}$.

The set of features $F_{composed}$ is the result of the union of both feature sets, such as $F_{composed} = F_{gray} \cup F_{stripes}$. The resulting view $V_{composed}$ comprises the additional constraint $f_7 \rightarrow f_4$, which is neither contained in V_{gray} nor $V_{stripes}$. This constraint is added as it is defined on the two features f_4, f_7 that are both contained in the feature set $F_{composed}$ of the composed view, such as $\phi_{rq} \in \mathbb{B}(F_{composed})$. View composition with respect to Boolean constraints is more than the union of constraint and feature sets to preserve feature model semantics. Feature model semantics define sets of valid variant configurations in terms of allowed feature combinations.

The following properties of the composition operator are obtained by defining view composition in terms of conjunction and set union.

Lemma 2. *The view composition operator is* commutative *and* associative*.*

The proof for Lemma 2 follows directly from the definition of the composition operator.

Feature model semantics is generally not compositional due to the cross-tree constraints Φ in feature models [ACF+09]. Thus, view composition and the derivation of perspectives is not distributive in terms of feature model semantics due to the binary constraints contained in views.

Proposition 1 (Non-Distributivity of View Composition). *For two arbitrary views $V_{FM}, V'_{FM} \in \mathcal{V}_{FM}$ on a feature model FM, the equation $[\![p_{FM}(V_{FM} \oplus V'_{FM})]\!] \neq [\![p_{FM}(V_{FM} \cup V'_{FM})]\!]$ holds.*

The proof for Proposition 1 follows directly from counter-examples addressing non-compositional feature model semantics due to cross-tree constraints. For instance, the set of derivable variant configurations of a view composed by applying the composition operator $\oplus$ on the two views V_{gray} and $V_{stripes}$ differs from the set of derivable variant configurations of a view resulting from set union of these views. The perspective defined by $p_{FM}(V_{gray} \oplus V_{stripes})$ is a refinement of the original feature model as it contains the constraint $f_7 \rightarrow f_4$, which is neither contained in V_{gray} nor $V_{stripes}$. The set of derivable variant configurations of this perspective is a subset of the variant configurations derivable from the original feature model *FM*, such that $[\![p_{FM}(V_{FM} \oplus V_{FM'})]\!] \subseteq [\![FM]\!]$. From this perspective are 12 variant configuration derivable compared to 136 variants derivable from the original feature model.

In contrast, composing the two views V_{gray} and $V_{stripes}$ by set union does not result in a subset of derivable variant configuration of the original feature model. For instance, as the constraint $f_7 \rightarrow f_4$ is neither contained in V_{gray} nor $V_{stripes}$, this constraint is not contained in the resulting view and hence, the perspective defined by $p_{FM}(V_{gray} \cup V_{stripes})$ is not a refinement of the original feature model. Due to the missing constraint, variant configurations containing only feature f_7 without feature f_4 are derivable. These variant configurations are not valid according to the semantics of the original feature model.

As a consequence of $\Phi_{V''_{FM}} \neq \Phi_{V_{FM}} \wedge \Phi_{V'_{FM}}$, the set of cross-tree constraints $\Phi_{V''_{FM}}$ in view V'' that results from the composition of the two views V_{FM} and V'_{FM} differs from the conjunction of the cross-tree constraints of the solitary views. Additionally, constraints $\phi \in \Phi_{V''_{FM}} \cap \Phi$ exist that are not contained in each of the aggregated views V_{FM} and V'_{FM} with $\phi \notin \Phi_{V_{FM}} \cup \Phi_{V''_{FM}}$, but in the original feature model Φ.

View composition is closed under *FM*-consistent views. Hence, the composition of two *FM*-consistent views yields an *FM*-consistent view again. This characteristic is beneficial for checking the consistency of the overall multi-perspective model efficiently as explained in Section 5.11.2.

Proposition 2 (Closedness of FM-consistent View Composition). $V_{FM} \oplus V'_{FM} \in \mathcal{V}^c_{FM}$ *holds for* *FM*-consistent *views* $V_{FM}, V'_{FM} \in \mathcal{V}^c_{FM}$.

Proof. *The well-formed tree structure and feature group constraints are preserved in the resulting view* V''_{FM} *as both aggregated views* V_{FM}, V'_{FM} *are* *FM*-consistent *on the feature model FM and the composition operation is monotonically non-decreasing on* F. *For require constraints* $\phi = f \rightarrow f'$ *with* $f, f' \in F_{V''}$, $\phi \in \Phi_{V''}$ *is guaranteed as* ϕ *is either contained in one of the aggregated views* V, V', *or in both. Otherwise, one view must have contained* f *without* f' *which contradicts the* *FM*-consistency *assumption.* □

The opposite direction of Proposition 2 does not hold as the aggregation of non-*FM*-consistent views may lead to an *FM*-consistent view. For instance, the result of the composition of all views in Figure 5.7 is a view on all features and constraints of the original feature model, which is therefore *FM-consistent.*

5.8. Relations between Feature Models, Views, and Perspectives

The relations between syntactic and semantic domain concepts of feature models, views and perspectives are illustrated in Figure 5.8. The circles on the left side represent the view domain, concepts of the feature model domain are depicted in the middle, and the variant space domain is represented on the right side. The set-theoretic concepts of views, feature models and variant configurations are depicted in black color, whereas the new introduced concepts of *FM*-consistent views, perspectives and reduced sets of variant configurations are highlighted in gray. In the middle of the figure, the feature model *FM* is visualized as a single black dot, which is contained in the set of all valid feature models $\mathcal{FM}(F)$ defined on the set of features F.

The semantic evaluation function $[\![.]\!]$ maps feature models into the domain of sets of variant configurations. Thus, $[\![FM]\!]$ is the set of all variant configurations defined by the feature model *FM*. A valid variant configuration F_{vc} consists of a set of features F and is a subset of the power

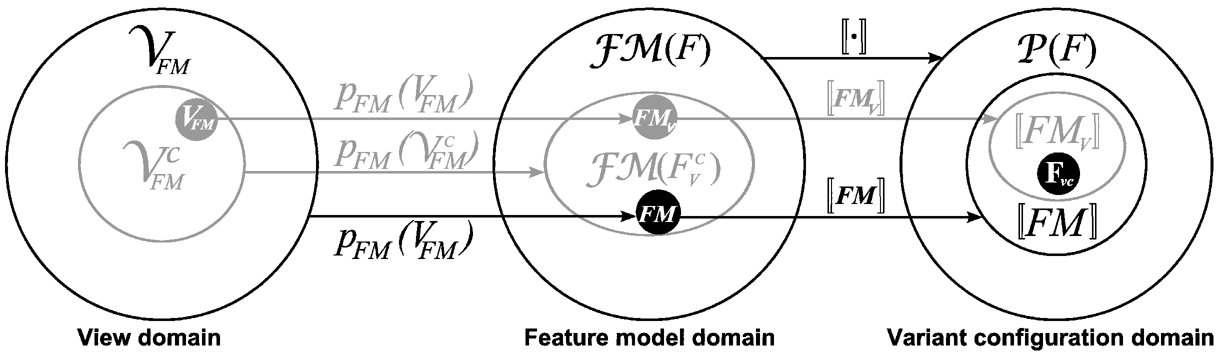


Figure 5.8 Formal concepts on views, feature models, and variant configurations.

set $\mathcal{P}(F)$, which is the set of all subsets of features. Additionally to this general feature model concepts, further notions of perspectives and views are depicted in gray color in the figure.

The feature model *FM* and a perspective FM_V on the feature model are contained in the set of perspectives $\mathcal{FM}(F^c_V)$ which is a subset of $\mathcal{FM}(F)$. $\mathcal{V}_{FM}$ is the set of all views on the feature model *FM*. $\mathcal{V}^c_{FM}$ is the set of all *FM*-consistent views and thus a subset of $\mathcal{V}_{FM}$. The partial projection p_{FM} defines perspectives on a feature model according to *FM*-consistent views in $\mathcal{V}^c_{FM}$. For instance, the *FM*-consistent view V_{FM} maps to the perspective FM_V, which in turn semantically refines *FM* by restricting the set of derivable variant configurations to a subset $[\![FM_V]\!]$.

Views modularize feature models for certain *concerns* and allow for SoC, as discussed in Section 5.1. Concerns are potentially interrelated, and thus concern-specific views overlap. Stakeholders are potentially interested in multiple concerns at once, thus concern-specific views are composed. In general, view composition is applicable to derive stakeholder-specific perspectives on a given feature model.

For capturing the relationships between overlapping concerns and perspectives a *view model* is introduced in the next section. The concept of a *viewpoint* is applied to collect multiple, not necessarily hierarchically related viewgroups to build a perspective. Each viewpoint must define a valid perspective on the feature model by composing the views defined by collected viewgroups.

5.9. View Model

A *view model* defines the relations between concerns in a hierarchically structured model. As shown in the *Business ByDesign* example in Figure 5.2 concerns and therefore views overlap. In this example, the view `Package Full CRM` overlaps with the `Sales Force CRM` in terms of features. A view model defines such views and their overlapping in a hierarchical structure. Therefore, viewgroups representing certain concerns are introduced. Viewpoints collect multiple viewgroups to define a concern-specific perspective on the feature model. A view model is defined as follows.

Definition 5.4 (View Model). *A* view model *VM is a triple (VP, G, ρ), where a finite set of m* viewpoints *$VP = \{vp_1, vp_2, \ldots, vp_m\}$, a finite set of n* viewgroups *$G = \{g_1, g_2, \ldots, g_n\}$, and a relation $\rho \subseteq VP \times G$ are specified.*

An example view model is depicted in Figure 5.9. Circles denote viewgroups and the lattice structure visualizes the hierarchical relation on viewgroups. Viewpoints are illustrated by an eye-like symbol in the figure, where the dashed lines originating from a viewpoint symbol illustrate viewgroups that are related to a viewpoint. The graphical illustration of a view model in the figure is inspired by the graphical notation of feature diagrams. Viewgroups G are hierarchically structured to represent interrelated concerns on a feature model. Each viewgroup $g \in G$ corresponds to a concern. Viewgroups that are lower in the hierarchy comprise concerns of viewgroups that are further up in the hierarchy. The core viewgroup represents a core concern, common to all hierarchically related viewgroups, and equal to a root feature in a feature diagram.

Viewpoints VP define which concerns are aggregated to derive a feature model pre-configuration. In the view model, ρ defines a relation between viewpoints and viewgroups. Thus, a viewpoint vp is related to a set of viewgroups $G_i \subseteq G$, such that $\rho(vp_i) = G_i = \{g_i \in G \mid (g_i, vp_i) \in \rho\}$. In the figure, the relation between a set of viewgroups and a viewpoint is highlighted by a frame surrounding related viewgroups and a eye-like symbol naming the viewpoint. For instance, viewpoint vp_2 is related to a set of viewgroups $G_{vp_i} = \{g_{core}, g_1, g_2\}$.

Each viewgroup g_i is related to potentially multiple viewpoints $VP_i \subseteq VP$, such that $\rho(g_i) = VP_i = \{vp_i \in VP \mid (g_i, vp_i) \in \rho\}$. Thus, all viewpoints VP_i that are related to a viewgroup g_i share the same concerns dedicated to that viewgroup. For instance, the viewpoint vp_1 is only

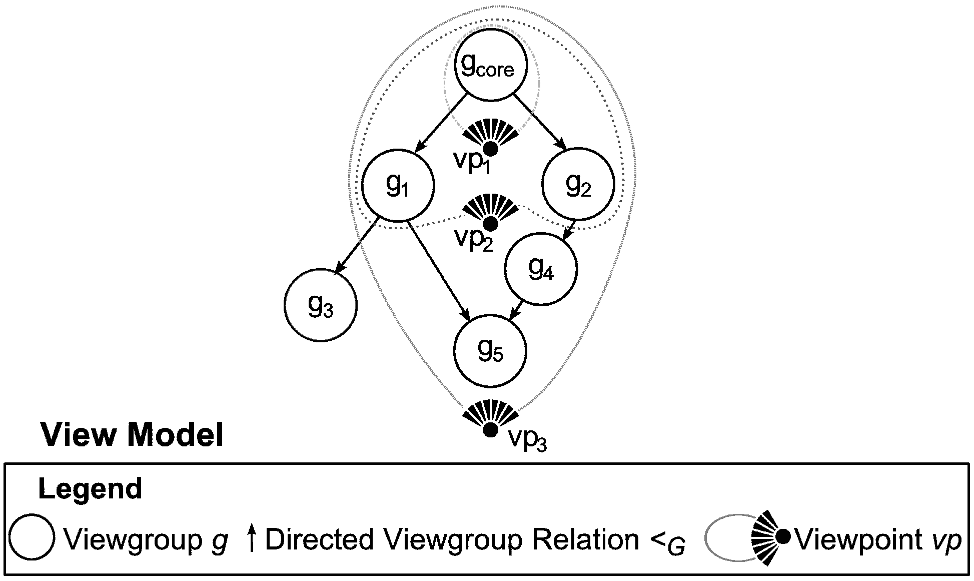


Figure 5.9 A view model captures the relationship between viewgroups and defines viewpoints that collect viewgroups.

related to the core viewgroup g_{core}, where viewpoint vp_2 is related to g_{core}, g_1 and g_2. Hence, both viewpoints have the viewgroup g_{core} in common.

Additionally, the relation ρ implicitly introduces an *hierarchical relation* $<_G \subseteq G \times G$ on viewgroups, where $g <_G g' :\Leftrightarrow g' \subsetneq g$ defines a *predecessor* relation among viewgroups via proper inclusion of their viewpoint sets. The relation $<_G$ is a *strict* partial order, as viewgroups with equal subsets of viewpoints defined by the relation $\rho(g_i) = \rho(g_j)$ are distinguished in G by their indices i and j. Hence, two viewgroups g_i and g_j are either related under $<_G$ or incomparable. For instance, in Figure 5.9, the viewgroups g_{core} and g_1 are related under $<_G$ as both are related to the viewpoints vp_2 and vp_3. However, the viewpoint vp_{core} is related to g_{core}, but the viewpoint is not related to the viewgroup g_1. Additionally, the viewgroups g_4, g_2 and g_{core} are related hierarchically, such that $g_{core} <_G g_2 <_G g_4$ holds, whereas g_2 and g_1 are unrelated under $<_G$. Viewgroups are incomparable, if they are either disjoint or overlapping, where the latter includes set equality. The viewgroups g_2 and g_3 are incomparable as g_3 does not have any viewpoint. The *overlapping viewgroup relation* $\sqcap_G \subseteq G \times G$ is defined as

$$g_i \sqcap_G g_j :\Leftrightarrow\ i \neq j\ \wedge\ g_i \cap g_j \neq \emptyset\ \wedge\ g_i \not<_G g_j\ \wedge\ g_j \not<_G g_i.$$

Thus, two viewgroups g_i and g_j overlap, if they do not represent the same viewgroup indicated by differing indices i and j and both viewgroups are neither disjoint nor related under $<_G$. For instance, in Figure 5.9, the viewgroups g_1 and g_2 overlap, as both are related to the viewpoints vp_2 and vp_3. Therefore, $g_1 \sqcap_G g_2$ holds, as the viewpoint is related to both viewgroups $vp_2 \in \rho(g_1) \cap \rho(g_2)$. In addition, the overlapping viewgroup relation $\sqcap_G$ is irreflexive, symmetrical and non-transitive.

A view model is *well-formed*, if the hierarchical viewgroup relation $<_G$ is closed upwards in G, such that a unique *core viewgroup* $g_{core} \in G$ exists with $g_{core} <_G g$ for each viewgroup $g \in G$. Hence, the core viewgroup represents a single root and any viewpoint is related to the core viewgroup represented by $\rho(g_{core}) = VP$.

The following notations for a view model VM are applied,

- a viewgroup $g \in G$ is a *direct predecessor* of $g' \in G$, if $g <_G g'$ and there exists no $g'' \in G$, such that $g <_G g'' <_G g'$,

- a viewgroup $g \in G$ is *most specific* for a viewpoint $vp \in VP$, if vp is related to g, i.e., $g \in \rho(vp)$, and there exists no $g' \in G$ with $g <_G g'$ and $g' \in \rho(vp)$.

Thus, the core viewgroup, which is g_{core} in Figure 5.9, has no predecessors. The core viewgroup g_{core} is unique and collects concerns common to all stakeholders. Any other viewgroup in the view model has potentially multiple direct predecessing viewgroups due to the overlapping of viewgroups. For instance, viewgroup g_5 has two direct predecessing viewgroups g_1 and g_4.

The view model further enables viewpoints to have multiple most specific viewgroups. For instance, the viewgroups g_1 and g_2 in the figure are most specific for viewpoint vp_2. Additionally, singleton most specific viewgroups g_s are applied to assign exclusive properties to viewpoints. A single viewpoint vp_i is related to a singleton most specific viewgroup g_s, such that $\rho(g_s) = vp_i$ and $|\rho(g_s)| = 1$ These viewgroups enable customization on feature model level as further explained in Section 5.13.

Additionally, a distinction between *abstract* and *concrete* viewgroups is made. This definition is picked up later in Section 5.11.2 to define an efficient algorithm for checking the consistency of multi-perspective models containing view models. Thus, a viewgroup $g \in G$ is

- *concrete*, if g is *most specific* to at least one viewpoint vp, and
- *abstract*, if g is *not most specific* to any viewpoint vp.

Viewgroup g_4 in Figure 5.9 is an example for an abstract viewgroup. Only viewpoint vp_3 is related to this viewgroup and thus $\rho(g_4) = \{vp_3\}$. In turn, the viewpoint vp_3 is related to a set of five viewgroups, such that $\rho(vp_3) = \{g_{core}, g_1, g_2, g_4, g_5\}$. Viewgroup g_4 is not concrete, as for the viewpoint vp_3 exists a viewgroup g_5 in $\rho(vp_3)$ further down in the viewgroup hierarchy, such that $g_5 <_G g_4$. In contrast, the viewgroup g_2 is an example for a concrete viewgroup, as g_2 is most specific for viewpoint vp_2.

To summarize, a view model specifies hierarchical relationships among concerns in viewgroups. Each viewpoint is related to a set of viewgroups, where each viewgroup defines a view on the feature model. A view, and therefore a viewgroup, is dedicated to a concern and collects multiple features. Thus, a viewpoint in the view model specifies sets of views that are composed in an *FM*-consistent view, which is called perspective. A perspective preserves feature model semantics by imposing a refinement of the configuration space. A *multi-perspective model* specifies the set of viewpoints and valid perspectives on the feature model.

5.10. Multi-Perspective Model

A multi-perspective model is a conservative extension to common feature models as it extrinsically models overlapping views on the feature model. Concern-specific views are defined by viewgroups in a separate view model, as introduced previously. Combining a feature model and a view model by mapping feature model views to viewgroups yields a multi-perspective model as visualized in Figure 5.10. Hence, a multi-perspective model comprises a feature model, a hierarchical view model and a mapping between views on the feature model and viewgroups in the view model. A viewpoint in the view model is related to multiple viewgroups representing concerns. Hence, a viewpoint defines a view on the feature model comprising multiple concerns. This view defined by a viewpoint is called a perspective. A perspective is a semantic refinement of the feature model as the number of derivable valid variant configurations is decreased. In Figure 5.10 three perspectives are defined, each corresponding to a viewpoint in the view model. A multi-perspective model is defined as follows.

Definition 5.5 (Multi-Perspective Model). *A* multi-perspective model *is a triple* (FM, VM, σ)*, where* $FM \in \mathcal{FM}(F)$ *is a* feature model*,* $VM = (VP, G)$ *is a* view model*, and* $\sigma : G \to \mathcal{V}_{FM}$ *is the* view mapping function *between viewgroups* G *and views* $\mathcal{V}_{FM}$*.*

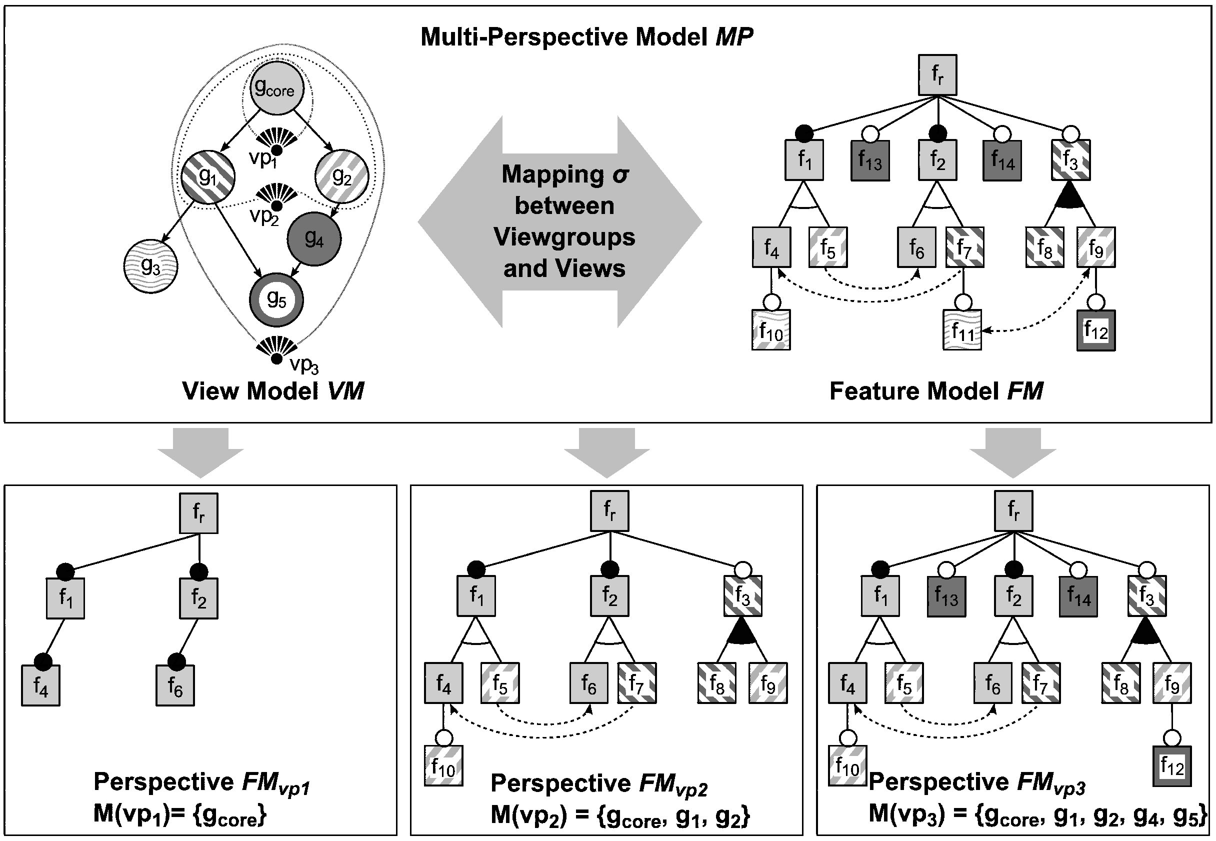


Figure 5.10 A multi-perspective model consistently pre-configures a feature model by deriving perspectives.

Every feature of the feature model *FM* is required to be mapped to at least one view. Thus, for each $f \in F$ exists a viewgroup $g \in G$ with $\sigma(g) = (F_g, \Phi_g)$ such that $f \in F_G$. In Figure 5.10, the mapping function σ is denoted by the same shading of viewgroups and features. Generally, the mapping between features and viewgroups is $n : m$, and thus viewgroups are projected in multiple views accordingly. For instance, feature f_{10} is mapped to both viewgroups g_3 and g_2, where viewgroup g_3 additionally maps to feature f_{11}.

Additionally, the mapping between the core viewgroup g_{core} and features is marked with a solid grey shading. The resulting view on the feature model is called *core view* and is *FM*-consistent. The other single views defined by σ and represented by different shadings are solely not *FM*-consistent. Instead, they are interpreted to refine the core view by individual features for particular concerns.

A viewpoint refers to an aggregated view V_{vp} that must be *FM*-consistent. This view is called a *perspective* on the feature model *FM*.

$$V_{vp} = \sigma(g_{core}) \oplus \sigma(g_1) \oplus \sigma(g_2) \cdots \oplus \sigma(g_k), \text{ where } \forall i \in \mathbb{N}_1, i \leq k : vp \in \rho(g_i)$$

A viewpoint V_{vp} composes views on the feature model to build a valid perspective. Each view $\sigma(g)$ is defined by a viewgroup g related to the viewpoint according to the relation $\rho(vp) = G_{vp}$ and $\forall g \in G_{vp} : \sigma(g)$. In Figure 5.10, there are three viewpoints in the view model, entitled vp_1, vp_2, and vp_3. Each viewpoint defines a valid perspective by aggregating different views, as depicted in the bottom of the figure. For instance, viewpoint vp_2 is related to the viewgroups g_{core}, g_1, and g_2. Thus, the view $\sigma(g_{core})$ defined by the viewgroup g_{core} contains the features f_r, f_1, f_2, f_4, and f_6 as highlighted by a light grey background in the figure. This view is aggregated with two further views $\sigma(g_1)$ and $\sigma(g_2)$. The features $F_{g_1} = \{f_3, f_7\}$ of view $\sigma(g_1)$ are highlighted with a dark striped background, while the features $F_{g_2} = \{f_5, f_9, f_{10}\}$ of view $\sigma(g_2)$ are highlighted with a light striped background.

The set of all perspectives defined by viewpoints VP on a feature model FM in a multi-perspective model MP is denoted by $\mathcal{V}_{MP} \subseteq \mathcal{V}_{FM}$. Hence, the views $\mathcal{V}_{MP}$ project multiple perspectives $FM_{vp} = p_{FM}(V_{vp})$ on a feature model. For instance, viewpoint vp_2 in Figure 5.10 is related to the viewgroups g_1, g_2, and g_{core}. The resulting view on the feature model defines the perspective vp_2, which is shown in the middle of the lower area of the figure. Eventually, all perspectives are *FM-consistent*, and thus preserve the semantics of the original feature model.

The *FM-consistency* criteria for perspectives are relevant to define the consistency of a multi-perspective model as described in the next section.

5.11. Consistency of Multi-Perspective Models

A multi-perspective model is required to be consistent. Consistency is defined with respect to the viewpoints in the view model and their relation to views on the feature model. Each viewpoint is required to define a perspective which represents a feature model refinement. Hence, a multi-perspective model MP is *consistent*, iff all views defined by viewpoints are *FM-consistent*.

Lemma 3. *The multi-perspective model* $MP = (FM, VM, \sigma)$ *is* consistent*, iff* $\mathcal{V}_{MP} \subseteq \mathcal{V}^C_{FM}$.

The proof of Lemma 3 follows directly from the *FM-consistency* of all views defined by viewpoints.

In the example in Figure 5.10, three viewpoints vp_1, vp_2, and vp_3 are defined in the multi-perspective model. All viewpoints specify $FM - consistent$ views obeying the semantics of the original feature model. Thus, the three viewpoints specify the three perspectives shown in the lower area of the figure and the multi-perspective model is consistent.

Changing the mapping of feature f_3 from viewgroup g_1 to viewgroup g_3 in this example renders two viewpoints inconsistent, as depicted in Figure 5.11. Compared to Figure 5.10, only the mapping of feature f_3 is changed in this example while the structure of the view model and the feature model remain the same. The views of both viewpoints vp_2 and vp_3 are inconsistent in this case. The composed viewpoint views are inconsistent, as the non-contained feature f_3 violates the decomposition relation $f_r \prec f_3 \prec f_8$ of the original feature model.

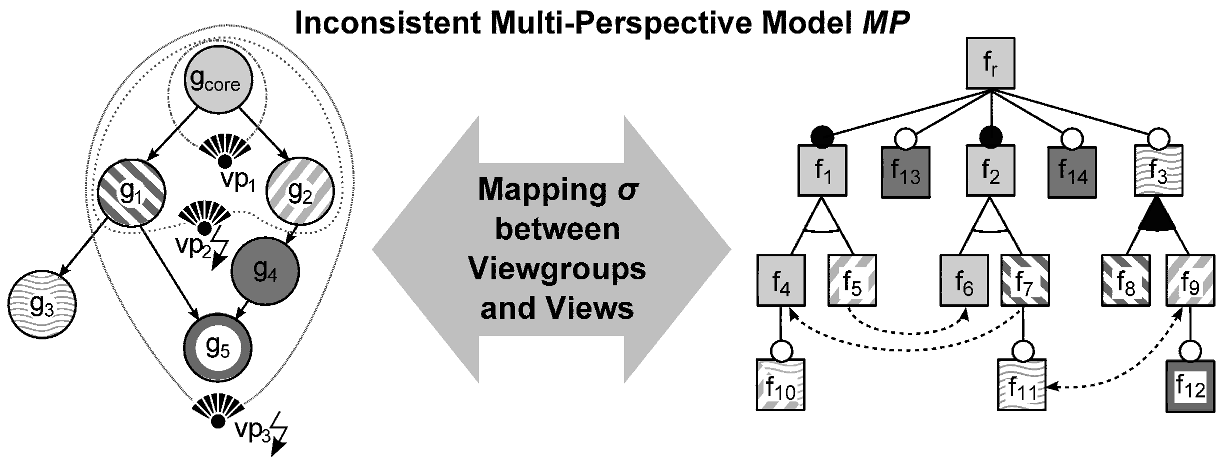

Inconsistent Multi-Perspective Model MP
g_core
g1
g2
g3
g4
g5
vp1
vp2
vp3
Mapping σ between Viewgroups and Views
fr
f1
f2
f3
f4
f5
f6
f7
f8
f9
f10
f11
f12
f13
f14

Figure 5.11 A multi-perspective model with two inconsistent viewpoints vp_2 and vp_3, and one consistent viewpoint vp_1.

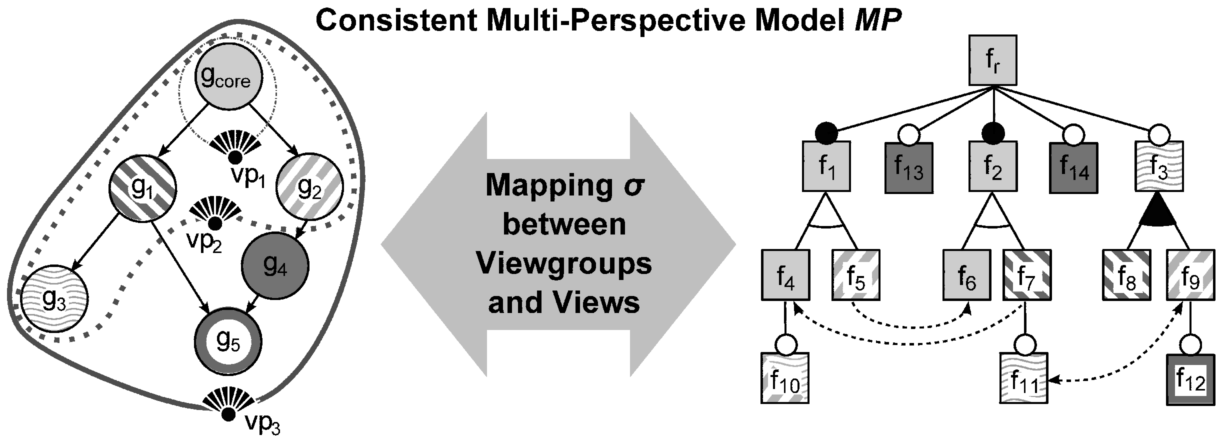

Consistent Multi-Perspective Model MP
g_core
g1
g2
g3
g4
g5
vp1
vp2
vp3
Mapping σ between Viewgroups and Views
fr
f1
f2
f3
f4
f5
f6
f7
f8
f9
f10
f11
f12
f13
f14

Figure 5.12 Changing the relation between viewpoints and viewgroups in the view model yields a consistent multi-perspective model.

The view specified by viewpoint vp_1 remains consistent in this example. Both inconsistent viewpoints vp_2 and vp_3 become consistent by defining a relation from both viewpoints to viewgroup g_3, such that $\rho(g_3) = \{vp_2, vp_3\}$. The resulting multi-perspective model is depicted in Figure 5.12. The changed relations of the viewpoints vp_2 and vp_3 are highlighted in the figure by thick lines surrounding the viewgroups related to these viewpoints. Hence, the resulting multi-perspective model is consistent as all contained viewpoints are consistent. In addition, viewpoint vp_3 specifies a perspective equal to the original feature model.

Ensuring the consistency of a multi-perspective model is a complex problem due to overlapping and hierarchically structured viewgroups in the view model.

The consistency of a single viewpoint is determined by performing a depth-first or a breadth-first search on the view model to collect all related viewgroups of the viewpoint. For instance, in the

Figure 5.11, the consistency of viewpoint vp_2 is checked, by collecting the viewgroups g_{core}, g_1 and g_2. Subsequently the views defined by these groups are aggregated by applying the composition operator $\oplus$, as explained in Section 5.7. The result is a single viewpoint view. This viewpoint view is then evaluated against the consistency criteria of Lemma 1. If the viewpoint view is *FM*-consistent, the viewpoint is valid. In this example, the resulting view of viewpoint vp_2 is not *FM-consistent* as features f_8 and f_9 are contained but not their parent feature f_3. This consistency check is subsequently repeated for each viewpoint to determine the consistency of the overall multi-perspective model and can be expressed as a brute-force algorithm, as explained in the following.

5.11.1. Brute-Force Algorithm Verifying the Consistency of the Multi-Perspective Model

Algorithm 1 verifies the consistency of a multi-perspective model *MP* by iterating over all viewpoints $vp \in VP$ specified in the view model *VM* of the multi-perspective model *MP*.

The following data structures are applied in the brute-force algorithm.

- V_{vp} – the aggregated view for viewpoint *vp* contains a set of features and a set of constraints.
- FM_{vp} – the perspective is derived for the observed viewpoint *vp*.
- *checkConsistency*(FM_{vp}, FM) – method to check the consistency requirements according to Lemma 1 of a perspective which is a view on the feature model imposed by a viewpoint.
- *vp.cons* – a flag for indicating viewpoint consistency. The flag is set to the return value of method *checkConsistency*.

To verify consistency of a multi-perspective model, the brute-force algorithm composes a view V_{vp} for each viewpoint *vp*. The view V_{vp} is defined by aggregating the set of views $\sigma(g)$ of (partial) viewgroups $g \in G$, where $\rho(vp) = G$. Subsequently, the perspective FM_{vp} for viewpoint *vp* is projected from the feature model *FM*.

Algorithm 1 Brute-Force Consistency Check of a Multi-Perspective Model

```
1: input:  multi-perspective model $MP = (FM, VM, \sigma)$
2: for all viewpoints $vp \in VP$ do
3:     for all viewgroups $g \in G$ where $vp \in M(g)$ do
4:         $V_{vp} := V_{vp} \oplus \sigma(g)$
5:     end for
6:     $FM_{vp} := p_{FM}(V_{vp})$
7:     $vp.cons := checkConsistency(FM_{vp}, FM)$
8: end for
9: return true if $\forall vp \in VP : vp.cons =$ true
10: return false otherwise
```

The method *checkConsistency* performs the *FM-consistency* check of the perspective FM_{vp} according to Lemma 1. The *FM-consistency* check includes the verification of *satisfiability* of each reduced feature model represented by a perspective. It is reported in literature that the satisfiability check of a feature model is reducible to SAT, which is presumably NP-complete [Bat05]. In addition, the viewgroup hierarchy in the view model defines set inclusions and overlapping on viewgroups. Thus, viewpoints related to non-disjoint viewgroups have many viewgroups in common, which leads to multiple redundant and costly satisfiability checks.

To summarize, the brute-force algorithm works well for multi-perspective models of small size with respect to the number of features and viewgroups. The algorithm is also applicable to check single viewpoints for consistency. However, the algorithm does not scale for complex multi-perspective models with numerous features and viewpoints, as shown in the performance evaluation described in Section 7.5.4. For complex multi-perspective models, an incremental algorithm is discussed in the next section.

5.11.2. Incremental Algorithm Verifying the Consistency of the Multi-Perspective Model

The incremental algorithm is based on the closedness property of the view composition operator subsumed in Proposition 2. The algorithm incrementally iterates over viewgroups instead of viewpoints. The following assumptions on a multi-perspective model *MP* are related to the closedness property of the view composition operator.

1. The original feature model *FM* is *satisfiable*.

2. The *FM*-consistency of the views V_g of all potential viewgroups g_i in separate, ensure *FM*-consistency of all aggregated views V_{VP} of every viewpoint $vp \in VP$.

Based on these assumptions, an efficient incremental algorithm for checking the consistency of a multi-perspective model *MP* is defined by interpreting view models $VM = (VP, G)$ as acyclic lattices $(G_c, \rightarrow)$, where

- nodes $g \in G_c$ refer to *concrete* viewgroups $G_c \subseteq G$ that are directly assigned to at least one viewpoint, and

- edges $g \rightarrow g'$ connect concrete viewgroups $g, g' \in G_c$, where both viewgroups are in a hierarchical relation $g' <_G^* g$ and each intermediate viewgroup $g'' \in G$ with $g' <_G^* g'' <_G^* g$ is *abstract*, such that $g'' \notin G_c$.

Algorithm 2 starts with observing the core viewgroup, and incrementally checks further concrete viewgroups for consistency.

The following data structures are applied in the incremental algorithm.

Algorithm 2 Incremental Consistency Check of Multi-Perspectives

```
1: **Input:** multi-perspective model $MP = (FM, (G_c, \rightarrow), \sigma)$
2: **Require:** $g_{core} \in G_c$
3: $\forall g \in G_c : g.F = \sigma(g)$
4: $\forall g \in G_c : g.cons = \mathbf{true}$
5: $g_{core}.cons := checkConsistency(g_{core}.F, FM)$
6: $\forall g \in G_c : g.done = \mathbf{false}$
7: $g_{core}.done := \mathbf{true}$
8: **for all** $g \in G_c$ **where** $g.done = \mathbf{true}$ **do**
9:     **for all** $g' \in G_c$ **where** $g \rightarrow g'$ **do**
10:        $g'.F := g'.F \cup g.F \cup F_{g \rightarrow g'}$
11:        $g'.cons := checkConsistency(g'.F, FM) \wedge g.cons$
12:        **if** $\forall g'' \in G_c$ **where** $g'' \rightarrow g' : g''.done = \mathbf{true}$ **then**
13:            $g' done := \mathbf{true}$
14:        **end if**
15:    **end for**
16:    $G_c := G_c \setminus g$
17: **end for**
18: **return true if** $\forall g \in G_c : g.cons = \mathbf{true}$
19: **return false otherwise**
```

- $G_C \subseteq G$ – the subset of concrete viewgroups that are most specific to at least one viewpoint. In addition, the core viewgroup is always required to be contained in G_C, even though this viewgroup might be abstract.
- $g \rightarrow g'$ – the viewgroup hierarchy relation considering concrete viewgroups G_C only.
- g_{core} – the unique root viewgroup of the view model
- $g.cons$ – a flag for indicating viewgroup consistency. The flag is set to false (and stays false) as soon as one potentially inconsistent view of a viewgroup is detected.
- $g.done$ – a flag for indicating that a viewgroup is checked. The flag is set to true if all predecessing viewgroups of that viewgroup are completely checked. Thus, the traversal continues at this viewgroup.
- $g.F$ – the set of features in the view of that viewgroup incrementally collected from all predecessing viewgroups defined by $<_G$.
- $checkConsistency(F, FM)$ – method to check the consistency requirements of a set of features comprising a view according to Lemma 1.
- $F_{g \rightarrow g'}$ – the union of features is mapped into views of abstract viewgroups g'' between two concrete viewgroups g and g'.

Algorithm 2 assumes that a viewpoint is solely assigned to the core viewgroup, whose perspective represents a satisfiable feature model. In the example in Figure 5.10, the viewpoint vp_1 is solely assigned to the core viewgroup g_{core}. Due to this assumption, the incremental algorithm requires a satisfiability check only once for the core viewgroup, while all other viewgroups are checked incrementally. Hence, the algorithm first checks the satisfiability of the perspective that belongs to viewpoint vp_1, which comprises the features f_r, f_1, f_2, f_4, and f_6. The resulting feature model, illustrated in the left corner of the figure, is satisfiable as the set of contained features represents a valid variant configuration. Initially, all concrete viewgroups are marked as unfinished by setting the flag $g.done$ for each viewgroup g to *false*. After checking the consistency of the core group, this group is marked as done by setting its flag $g_{core}.done$ to *true*. Subsequently, the algorithm collects feature sets of the concrete viewgroups that are in a hierarchical relation. Features assigned to the the previous concrete viewgroup g are combined with the features assigned to the subsequent concrete viewgroup g' and joined with features assigned to all intermediate abstract viewgroups. This joint feature set is assigned to the concrete viewgroup g'.

Subsequently, the algorithm checks concrete viewgroups for *FM*-consistency by verifying the conditions of Lemma 1. A concrete viewgroup is directly assigned to a viewpoint, while an abstract viewgroup is only indirectly assigned to a viewpoint via hierarchical inclusion. For instance, concrete viewgroups are g_1, g_2 and $g5$, where abstract viewgroups are g_3 and g_4 in the example in Figure 5.10. Hence, only the hierarchical relations $g_{core} \rightarrow g_1$, $g_{core} \rightarrow g_2$, $g_1 \rightarrow g_5$, and $g_2 \rightarrow g_5$ are considered by the algorithm. The hierarchical relation $g_{core} \rightarrow g_1$ between the concrete viewgroups g_1 and g_{core} is direct without intermediate abstract viewgroups. In contrast, the hierarchical relation $g_2 \rightarrow g_5$ comprises the abstract viewgroup g_4. Features assigned to abstract viewgroups that are contained in the hierarchical relation between concrete viewgroups are determined.

Thus, via the hierarchical relation $g \rightarrow g'$ between concrete viewgroups g, g', a set of features is determined by collecting features assigned to the concrete viewgroup g. Feature sets are added incrementally by evaluating the mapping function $\sigma(g)$ for each viewgroup g and for all abstract predecessing viewgroups $F_{g' \rightarrow g}$. The incremental algorithm benefits from the property that the composition of *FM*-consistent views results in an *FM*-consistent view.

The incremental algorithm is an algorithm without backtracking as each viewgroup is checked. Thus, the performed traversal of the hierarchical relation of concrete viewgroups $(G_c, \rightarrow)$ has a complexity equivalent to breath-first-search, where each segment $g \rightarrow g'$ is checked based on the previously conducted steps. Hence, the incremental algorithm is a wavefront algorithm on an acyclic lattice. The application of the incremental algorithm on a multi-perspective model *MP* results in Theorem 1 extending Lemma 3.

Theorem 1 (Multi-Perspective Model Consistency). *If the multi-perspective model MP passes Algorithm 2 successfully, then MP is* consistent.

Theorem 1 is proved by induction as follows.

Proof. *First, the algorithm always terminates because the construction of* $(G_c, \rightarrow)$ *always results in a finite, connected, directed, and acyclic graph. Therefore, predecessing nodes always exists for the traversal, and no cyclic traversals may arise. The preservation of FM-consistency can* *be shown by induction over the traversal of paths in* $(G_c, \rightarrow)$. *The induction starts by ensuring FM-consistency of the view imposed by the core viewgroup, which is predecessor of any other viewgroup. In every step, the algorithm incrementally ensures for every* edge $g \rightarrow g'$ *to be consistency preserving. This is done by assuming the view of* g *to be already checked as FM-consistent, which reflects the induction hypothesis, and by verifying that the aggregated views for the set* $F_{g \rightarrow g'}$ *preserve FM-consistency. Thus, the incremental traversal ensures concrete views to preserve FM-consistency, if all their predecessors under* $<_G$ *are FM-consistent. Thereby, the view of* g *is aggregated from views of viewgroups under* $<_G$ *via hierarchical inclusions. Moreover, consistency of overlapping viewgroups is given according to Proposition 2. Thus, views arbitrarily aggregated for viewgroups* $g \sqcap_G g'$ *also implicitly preserve FM-consistency due to the closedness of FM-consistent view composition. This property of the view composition operator always holds, and does not need to be checked explicitly in the algorithm.* $\square$

The opposite direction of Theorem 1 does not hold, as the algorithm can produce false negatives for multi-perspective models that contain viewpoints with consistent views that are aggregated by inconsistent partial views of concrete viewgroups. As a result, the consistency of all viewpoints is correctly determined by drastically reducing the number of explicit satisfiability checks in comparison to the brute-force algorithm.

To apply the multi-perspective approach on an SPL, the domain and the application engineering processes are extended.

5.12. Multi-Perspectives in Software Product Line Engineering

In SPL engineering, the processes of domain and application engineering are distinguished, as explained in Section 2.2. The presented multi-perspective approach is integrated into these main processes. In addition to model commonality and variability of variant configurations in a feature model, the domain engineering process is extended to extrinsically model reusable stakeholder concerns on the feature model. As the instantiation of a feature model is independent from the multi-perspective approach, multi-perspectives are also applicable to existing feature models. To apply the approach on a feature model, a hierarchical view model is firstly created, and features are mapped to viewgroups. After that, viewpoints are identified in the domain engineering process. These viewpoints are intended to derive perspectives in the application engineering process.

The application engineering process is extended by creating a consistent perspective prior to variant derivation. Thus, products are not directly derived from the original feature model, but from a refined perspective. Therefore, the process is extended with an initial step for defining or selecting a viewpoint in the view model that represents certain stakeholder concerns. According to the specified viewpoint, a perspective is derived automatically. Finally, instead of the original feature model, the derived perspective is further refined in the configuration process to create a variant configuration. Furthermore, the proposed multi-perspective approach allows for expressing customizations on feature model level.

5.13. Customization on Feature Model Level

In most SPL approaches, a product is derived from the SPL, which is then further customized on source code level [ESSPL10]. As such, there is no information about the customization on feature model level. This lack of knowledge is inconvenient for the evolution of an SPL, where features may be modified, removed or replaced.

On a product update, the customizations must be retained, but the impact of the evolutionary steps is not obvious as feature customizations are untraceable. Customizations for single customers can only be incorporated into the domain feature model, if their access is restricted.

The proposed multi-perspective approach supports customization on feature model level in terms of perspectives. For that reason, a special kind of viewgroups is used in the view model to exclusively assign customization properties to particular viewpoints $vp \in VP$. These viewgroups are named *singleton* viewgroups $g_s \in G$, as they are related to a single viewpoint, such that $|\rho(g_s)| = 1$. Such viewgroups are unique in G. For instance, assuming that feature f_{12} in Figure 5.10 is customized, the singleton viewgroup g_5 restricts the availability of the feature. Furthermore, g_5 is a singleton viewgroup of viewpoint vp_3 and thus only contained in the derivable perspective of that viewpoint shown below in the figure.

Customer-specific features can be added to the original feature model, and are restricted to a customer's perspective by mapping the features to a singleton viewgroup of the customer's viewpoint. Other viewpoints must not be affected and still remain *FM*-consistent. Thus, already derived variant configurations are not invalidated.

Additionally, a customization requested by a single customer may later on be requested by further customers turning it into a public-available domain feature. The customized feature initially mapped to the customer's singleton viewgroup, will be mapped to a different viewgroup available to further customers. The multi-perspective approach enables traceable customizations on feature model level in evolutionary scenarios.

5.14. Best Practices in Modeling Views and Perspectives

Multi-perspective models can be constructed in different ways. However, various experiments revealed best practices in modeling multi-perspectives. As a result, the rules listed below explain how to create meaningful models with reduced computation complexity [SLW11].

1. *Core features are to be mapped to the core viewgroup.*
   Features that are contained in every derived product are called *core* features [BSRC10]. All features mapped to the root viewgroup are contained in any derivable perspective as this viewgroup is related to every viewpoint. In Figure 5.10, core features are the root feature f_r, as well as the two mandatory features f_1 and f_2. Hence, they are mapped to the core viewgroup g_{core}.

2. *Features that should not be available in every perspective are best modeled as optional features.*
   Optional features in the feature model can be excluded from particular perspectives by an appropriate mapping to viewgroups. Furthermore, subtrees of the feature model can be explicitly excluded from certain perspectives by introducing optional features on a higher level in the feature model hierarchy. Such features can also be *abstract*, such that they do not correspond to a particular implementation, but are used for structuring purposes only [TKES11].

   For instance, in Figure 5.10 features f_{11}, f_{12}, f_{13} and f_{14} are modeled as optional features and assigned to viewgroups further down the viewgroup hierarchy. Thus, they are not included in every perspective.

3. *All viewgroups should be related to viewpoints to prevent dead features*
   Features that are not contained in a derived product are called *dead* features [TBD$^+$08]. Therefore, features are dead, if they are not contained in a perspective. If features are solely mapped to viewgroups that are not related to viewpoints, the features are excluded from any derivable perspective.

   This holds for feature f_{11} in Figure 5.10, for instance. The feature is solely mapped to viewgroup g_3, which is not referenced by any viewpoint. Hence, f_{11} is not included in any of the three derivable perspectives show on the bottom of the figure. Consequently, the exclude constraint $f_{11} \rightarrow f_9$ is not contained in any perspective as, due to the absence of feature f_{11}, this constraint can never be invalidated.

4. *The hierarchies of feature model and the view model should be aligned.*
   The incremental consistency check algorithm traverses the view model from top to bottom starting from the core viewgroup. The algorithm works best, if partial views created for concrete viewgroups are *FM*-consistent and thus obey feature model semantics. If features on a lower level in the feature model hierarchy are mapped to viewgroups on a high level in the view model hierarchy, potentially many partial views are inconsistent, which causes the algorithm to perform more costly SAT checks.

   The mapping of features to viewgroups should reasonably follow the feature decomposition relation of the feature model. For instance, this holds for the left branch of the view model in Figure 5.10 comprising viewgroups g_{core}, g_1 and g_3 and the features f_r, f_2, f_6, f_7 and f_{11} of the middle feature branch mapped to these viewgroups. Hence, the root feature f_r, as well as features f_2 and f_6 in the hierarchical relation, are mapped to the core viewgroup g_{core}, while feature f_7 is mapped to viewgroup g_1 one level lower in the view model hierarchy. Finally, the lowest feature f_{11} in this hierarchy is mapped to the lowest viewgroup g_3.

   In contrast, viewgroup g_4, which is on the same low level as g_3, is mapped to the two features f_{13} and f_{14} that are on a high level. This is reasonable as these features do not have child features, which are mapped to a viewgroup on a higher level in the view model hierarchy.

5. *Features of a require relation should be mapped to viewgroups in the same hierarchy.*
   Both features of a require constraint should be mapped to same viewgroup or to viewgroups in the same hierarchy in the view model to achieve good performance of the incremental consistency algorithm. In Figure 5.10, there is a require relation between feature f_7 and f_4 and another between features f_5 and f_6. However, in both cases, the features are not mapped to the same viewgroups, but to viewgroups in the same hierarchy. Alternatively, the feature that is required by another feature could be mapped to a viewgroup further up in the view model hierarchy. Hence, both features are contained in the same view.

6. *The selection of a feature can be restricted by excluding it from a perspective.*
   A feature is excluded from a perspective by assigning it to a viewgroup unrelated to the viewpoint of the perspective. Due to particular concerns, features should explicitly not be available for selection. It is reported recently that for business reasons customers rather define which features should not be included in a product than the define what should be included [Kru13].

7. *A domain feature can be replaced by a customized feature in a perspective.*
   A domain feature that is not required by another feature in a require cross-tree constraint, can be replaced by a customized feature. Both features, customized and domain feature, must share the same ancestor feature in the feature model, and the customized feature is mapped to the singleton group of the stakeholder's viewpoint. Additionally, the domain feature must be mapped to a viewgroup unrelated to the stakeholder's viewpoint. Thus, in the resulting perspective, the domain feature will not be available for selection, and instead be replaced by the customized feature. Replacing a domain feature that is required by another feature leads to further modification of the feature model.

   Domain feature and the replacing customized feature must be added to the same feature group, and an abstract parent feature must be inserted between this group and the former ancestor feature in the feature model. In addition, the require constraint must be changed to point to that abstract feature. Furthermore, the abstract feature must be mapped to the same viewgroup as the original feature. This concept is illustrated in Section 5.3.

5.15. Applying Multi-Perspectives to Support Staged Configuration

The multi-perspective approach extends common staged configuration approaches by supporting overlapping views, and the explicit exclusion of features from manual configuration. A staged configuration workflow describes the incremental process of deriving valid variant configurations operationally [CHE04, CHH09, Hub12]. A staged configuration workflow is separated into multiple stages that are associated to potentially overlapping stakeholder-specific views, as explained in Section 2.4.2. A staged configuration workflow must lead to valid variant configurations, even if not all features are configurable.

A multi-perspective model can be applied to check, if the defined configuration views allow to derive variant configurations. To illustrate how, the staged configuration example based on the *Business ByDesign* case study and introduced in Section 2.4.2 is picked up. The staged configuration workflow contains two stages, where features are assigned to the corresponding views $view_1$ and $view_{2a}$ as shown in Figure 5.14. Thus, $view_1$ contains the 8 features highlighted by a dark gray background, while $view_{2a}$ contains the remaining 11 features with a light gray background. Views may overlap. The views $view_{2a}$ and $view_{2b}$ both contain the features `Product Development`, `Market Development`, `Purchase Request and Order Management`, `Supply Chain Design`, and `Product Engineering`.

In the example workflow, introduced in Section 2.4.2, two stages and corresponding views are defined. The first stage `stage 1` is related to the view `view 1`, while the second stage `stage 2` is related to `view 2` in the given workflow. These stages are represented by the viewgroups $stage_1$ and $stage_{2a}$ in Figure 5.13 accordingly. The view model depicted in the figure generally defines a viewgroup per stage.

Features associated to configuration stages are mapped to the viewgroups accordingly. The same background color of features and viewgroups highlights this mapping. In the visualized example, no features are mapped to the root viewgroup *core* and thus the views $view_1$ and $view_{a2}$ contained in the workflow are disjunct. In addition, a viewpoint vp_{full} is defined representing the overall staged configuration workflow applied in the use case in Section 2.4.2. As the workflow in this use case has two views, the viewpoint is related to the viewgroups $stage_1$ and $stage_{2a}$. The viewgroup $stage_{2b}$ does not belong to the viewpoint vp_{full}. Hence, features solely dedicated to such viewgroups, that are not related to the workflow viewpoint, cannot be configured in the staged configuration workflow represented by this viewpoint.

The aggregated view imposed by a workflow viewpoint must be *FM*-consistent according to the consistency requirements defined in Lemma 1. In other words, a staged configuration workflow is only valid, if the corresponding perspective is valid. In this example, the aggregated view ($view_{full}$) contains all feature of the original feature model. Thus, the corresponding perspective

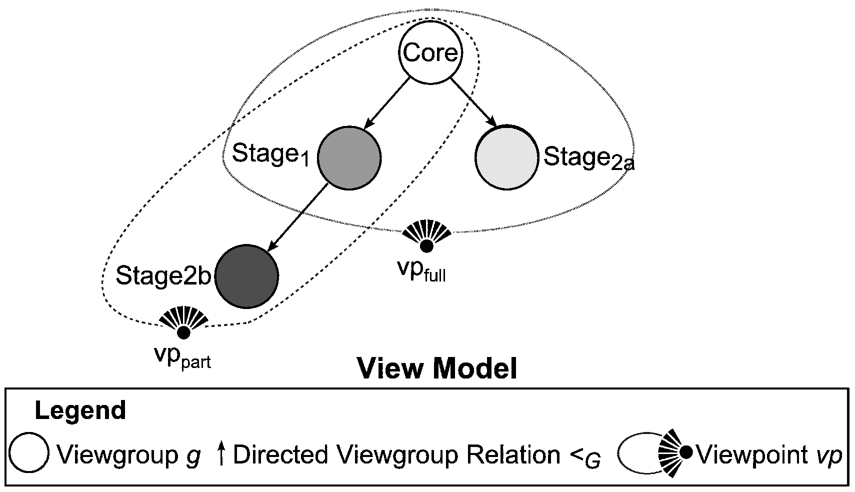


Figure 5.13 Views of a staged configuration workflows are expressed by viewgroups in a view model.

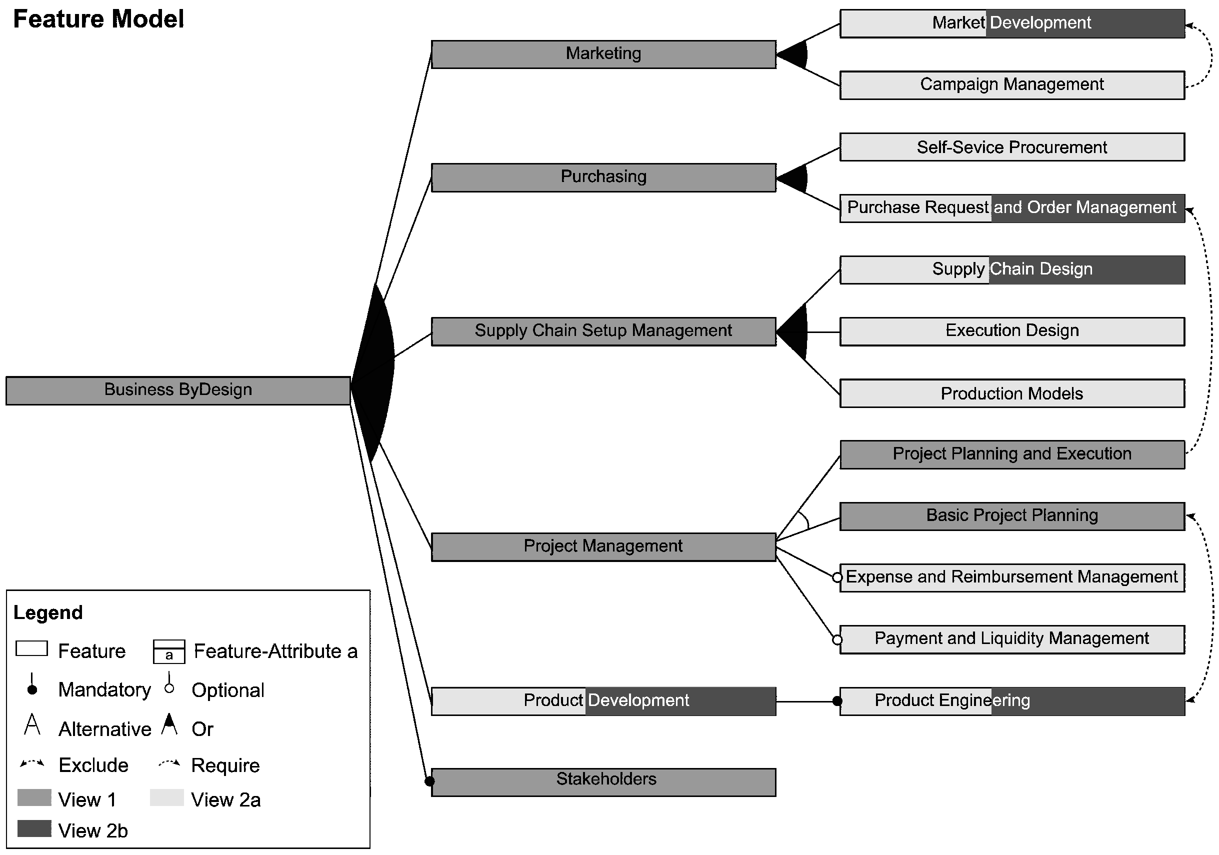


Figure 5.14 Features of the Business ByDesign application are grouped in three configuration views.

of viewpoint vp_{full} equals the original feature model and is therefore valid. The multi-perspective model ensures that each workflow definition represented by a viewpoint is valid.

Viewgroups can be reused in other staged configuration workflows by introducing further viewgroups for additional stages and differing viewpoints for other workflows. As such, viewpoint vp_{part} represents another valid workflow and reuses the viewgroup $stage_1$ of the first workflow, while introducing another stage $stage_{2b}$. The corresponding views are $view_1$ and $view_{2b}$ in Figure 5.14.

Features neither assigned to $stage_1$ nor to $stage_{2b}$ are not selectable in this workflow. Features that cannot be selected in this example are `Campaign Management`, `Self-Service Procurement`, `Execution Design`, `Production Models`, `Expense and Reimbursement Management`, and feature `Payment and Liquidity Management`. Additionally, as $stage_{2b}$ is hierarchically related to $stage_1$, the configuration view defined by this viewgroup contains the aggregated features mapped to $stage_1$ and to $stage_{2b}$.

However, a multi-perspective model does not define an order among configuration stages. An order can be defined by applying a behavioral workflow model, as explained in Section 2.4.2.

5.16. Demarcation from Related Work

Extending SPL engineering with multi-perspectives on feature models, as proposed in this work, is related to work in the area of staged configuration, and in realizing SoC on feature models in general. For instance, views separate stakeholder configuration decisions in staged configuration processes [CHE05a, Hub12, HHS$^+$11]. A notion of views on feature models in general, and the definition of overlapping and reconciled views is given in the notion of category theory by Clarke and Proença [CP10]. Thus, a view belongs to a stakeholder concern and is a projection on the feature model that covers a set of features. Therefore, each view realizes one semantic perspective of a stakeholder. In this thesis, the notion of a perspective is applied to denote a special view that complies with further consistency properties. As such, a view is any fragment of the feature model, whereas a perspective is a view that equals a reduced feature model.

Acher and colleagues introduce a decomposition operator to slice feature models according to particular concerns [ACLF11, ACLF12]. A feature model slice is constructed by explicitly cutting of features from the feature model. A view is similar to a feature model slice. However, a perspective in this thesis is constructed by composing views. In contrast to view-based approaches in SPL engineering presented in literature, the approach proposed in this work aggregates and integrates views to tailor the variant space. In addition, multi-perspectives define not only overlapping views, but a hierarchically structure of views.

Other work on views, viewpoints and perspectives presented in literature is at a different level of abstraction. For instance, Rosenmüller defines the concept of a binding unit, which incorporates a set of features and is understood as a view from an implementation point of view [RSPA11b]. Pohl an colleagues propose the concepts of *external* and *internal* variability to distinguish between customer and SPL developer related views on the variability of an SPL [PBvdL05]. However, these views have different levels of abstraction according to stakeholder concerns and are applied OVM, not on feature models.

Zaid and colleagues propose multi-perspectives to handle the complexity of large systems [ZKT10]. This approach assumes, that a software system consists of various perspective each representing a specific concern. The method of *feature assembly modeling* is introduced for separating features from variability information to reuse features in different applications. Another approach with a similar understanding of perspectives is presented by Meerkamm [Mee11]. Process variants are modeled as an SPL and perspectives are introduced on these processes for differing aspects and on varying abstraction levels. The notion of a perspective in both approaches equals the notion of a view in this thesis. However, aggregating views is not considered in these works.

5.17. Summary

This chapter introduces multi-perspectives as a conservative extension to convenient feature models. Potentially overlapping concerns are assumed as hierarchically structured views and defined in an external model beside the feature model. Prior configuring product variants, a set of views is selected to tailor the variant space by means of a feature model pre-configuration.

A pre-configuration is called a perspective. A perspective composed concern related views and is specified declaratively as a projecting view on a feature model. Only combinations of views are allowed to compose a perspective that satisfy feature model dependencies. The defined consistency criteria prevent the derivation of invalid perspectives.

Multi-perspective models consistently define arbitrary cross-cutting views on feature models. Those views can reflect various business concerns occurring in cloud scenarios. For instance, variable pricing strategies based on feature bundles, packages, and editions are widespread in offering cloud applications, as shown in Figure 5.2. A multi-perspective model is therefore a good foundation to express such strategies formally. In addition, multi-perspectives address legal restrictions. As discussed in Section 1.5.3, some features of cloud applications are only available in certain countries. Thus, the multi-perspective model allows for expressing and restricting country-specific variability by means of perspectives.

Generally, in a cloud scenario, multiple different stakeholders with different concerns are involved in provisioning an application, as discussed in Section 1.6. Hence, perspectives are a promising concept for tailoring the variant space of a feature model for the concerns of various stakeholders. The concepts are exemplified on different examples.

However, the proposed approach is not restricted to cloud applications, but generally applicable to scenarios, were SoC and overlapping concerns on feature models are to be expressed, and filters of the configuration space are appropriate. For instance, multi-perspectives are applicable to narrow the configuration space of a DSPL to increase performance by filtering unrelated concerns [SOS+12]. Additionally, the multi-perspective approach explicitly supports customization on feature model level, where customer-specific features are only available in the customer-specific perspective while hidden from other perspectives.

Moreover, this chapter reveals that consistency of a multi-perspective model can be checked efficiently by applying the proposed incremental algorithm. Thus, when applying a staged configuration workflow to configure a cloud application, the multi-perspective model ensures satisfiability of the configuration workflow, as explained in Section 5.15.

6. Adaptive Staged Reconfiguration Workflows Reduce Configuration Redundancies

The first rule of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency. The second is that automation applied to an inefficient operation will magnify the inefficiency.

— *Bill Gates*

The concepts presented in this chapter are partially published in a conference paper [MS13], workshop papers [Sch11, SMM$^+$12], and a technical report [LMSW13].

This chapter introduces the concept of *adaptive staged reconfiguration workflows* based on staged configuration workflows that are explained in Section 2.4.2. Adaptive staged reconfiguration workflows extend these concepts to reduce configuration redundancies by reusing partial feature model configurations and supporting reconfiguration. Furthermore, the workflow is adaptive to enable a dynamic management of stakeholders involved in the configuration workflow. Dynamic in this context means, that stakeholders can be integrated and removed from a staged configuration workflow during workflow execution. The introduction of adaptive staged reconfiguration workflows addresses Requirement 3.

Extended feature models with attributes are applicable to express functional and quality variability of reconfigurable cloud applications. Hence, resources, platform services, application functionality, qualities, as well as their dependencies are modeled in a feature model [Sch11]. However, different stakeholders are involved in the configuration and reconfiguration of cloud applications with different impact of their configuration decisions. *Staged configuration processes* specify the dependencies between configuration stakeholders and prioritize their configuration operations as explained in Section 2.4.2. Each stakeholder has different configuration concerns. For instance, a resource provider configures resources, while a customer decides among available application functionality and quality. Each stakeholder requires an own configuration view accessing restricted configuration operations.

In multi-tenant aware cloud applications, the decisions of particular stakeholders constrain the configuration operations of multiple depending stakeholders. For instance, if the resource provider deselects particular servers required by a certain application functionality, then this application functionality cannot be selected by any customer. Furthermore, if a resource provider decides to enable the servers again due to finishing server maintenance, all customers are to be notified of this reconfiguration change. Staged configuration concepts explained in Section 2.4.2 are extended to model these dependencies and support reconfiguration of partial configurations. A sub class of staged configuration workflows is introduced therefore, called *specialization trees*.

A specialization tree reduces configuration redundancies of multiple specialization processes by comprising them in a single workflow. Hence, specialization trees allows for reusing partial configurations and directly propagating reconfiguration changes [SMM+12].

Moreover, not all configuration stakeholders are known at design time of a cloud application. For instance, customers and their users subscribe for application services at runtime requiring a dynamic management of stakeholders. Dynamic stakeholder management is implemented by applying *workflow adaptation* comprising sequences of rewrite rules to adapt the staged configuration workflow during execution. An illustrative example is introduced in the following.

6.1. Illustrative Example for Adaptive Staged Reconfiguration Workflows

An example of a variable document management system is applied to explain the concepts of adaptive staged reconfiguration workflows. In this example, a document management system is offered as a reconfigurable SaaS application. Configuration parameters comprise functional and quality related parameters of the application for indexing and searching different document types, infrastructure resources, and platform services. Various stakeholders are involved in configuring the document management system corresponding to the stakeholder types explained in Section 1.6. Stakeholders and their configuration order are depicted in Fig. 6.1.

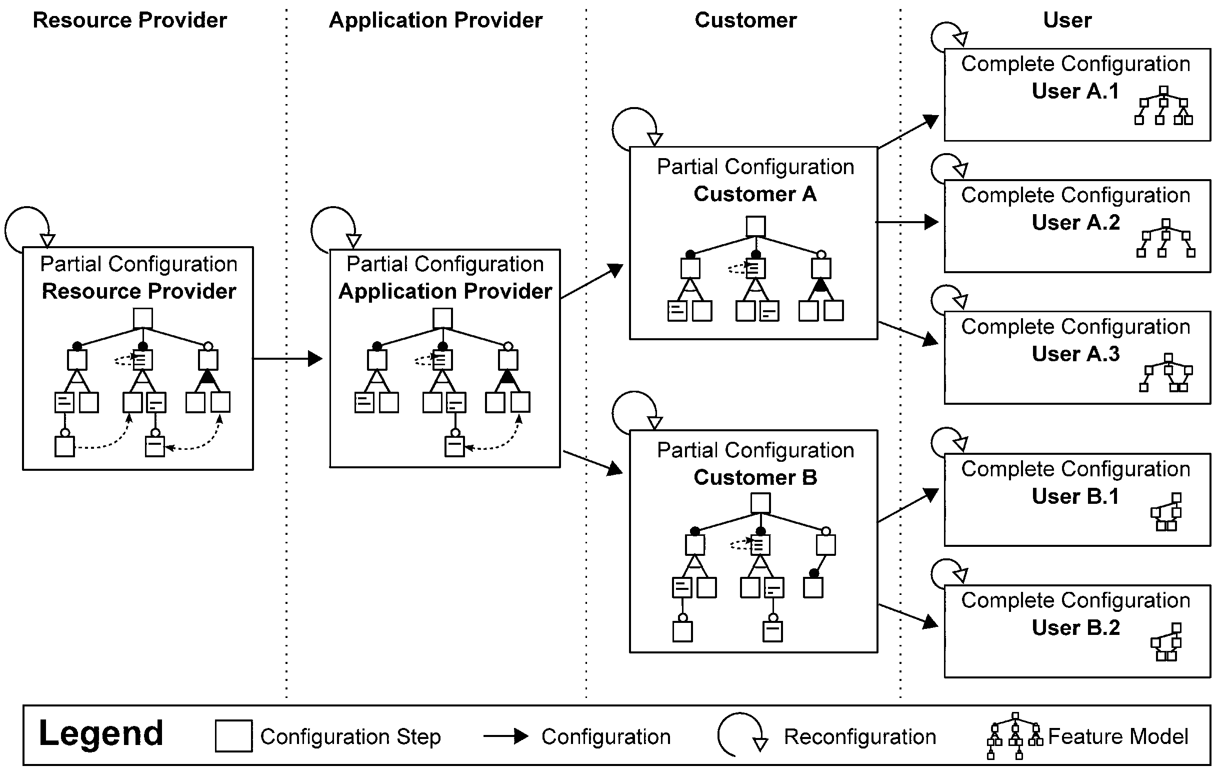

Resource Provider
Application Provider
Customer
User
Partial Configuration Resource Provider
Partial Configuration Application Provider
Partial Configuration Customer A
Partial Configuration Customer B
Complete Configuration User A.1
Complete Configuration User A.2
Complete Configuration User A.3
Complete Configuration User B.1
Complete Configuration User B.2
Legend
Configuration Step
Configuration
Reconfiguration
Feature Model

Figure 6.1 Configuration stakeholders in the document management system example.

A *resource provider* is responsible to configure infrastructure and platform services required to run the application and an *application provider* pre-configures the application for customers. Configuration operations of provider-related stakeholders have impact on all customer and user configuration operations.

Customers can subscribe to the application to offer customized application functionality as a service to *users*. In this example, two customers A and B are considered. They constrain the availability of functionality and quality for their users. Each user bind remaining variability of a given partial configuration leading to a complete configuration. While three users *A.1*, *A.2*, and *A.3* are affiliated to customer A, customer B is related to two users *B.1* and *B.2*. Each stakeholder conducts configuration operations on the defined configuration parameters until all variability is bound in the last specialization step. Users subscribe and unsubscribe for the usage of customer-specific application at anytime. If customers decide to unsubscribe, all their users are affected and are automatically unsubscribed as well.

Furthermore, every stakeholder can reconfigure the application instance. For example, due to server maintenance the resource provider marks data centers as currently not available. Hence, this change is propagated to the other stakeholders and potentially affects their configurations. If the requirements of customers change, reconfiguration occurs affecting only users of this customer. How to model this example configuration scenario by defining an adaptive reconfigurable workflow is explained in the following.

6.2. Modeling Adaptive Staged Reconfiguration Workflows

In general, a model is an abstraction of something complex and applied in different science contexts for different purposes. In MDSD, models capture the essentials of a specific domain and abstract from implemented source code [Jac06]. MDSD aims at improving software quality by deriving software automatically from formally specified models [SVC06]. An adaptive staged reconfiguration workflow models and automates configuration processes on feature models by means of a workflow. The workflow can be adapted to integrate and remove configuration stakeholders during execution the workflow. An *adaptive staged reconfiguration workflow* is specified by different structural and behavioral models, as depicted in Figure 6.2.

An *extended feature model* defines the configuration space and expresses configuration states of features and qualities of a cloud application. Section 6.3 introduces extended feature models

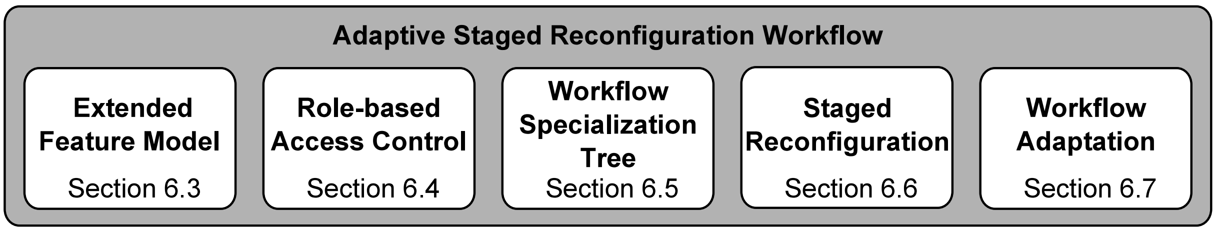


Figure 6.2 Conceptual models combined in an adaptive staged reconfiguration workflow.

for specifying variability of cloud applications. An *access control model* defines permissions on configuration operations of an extended feature model by applying RBAC. RBAC is common in shared cloud applications and therefore applied to restrict the access on configuration operations to specific stakeholders. Section 6.4 introduces the concepts of access control models. A staged configuration workflow orders stakeholders conducting configuration operations.

A *specialization tree* is a special class of staged configuration workflows for modeling multiple configuration processes in a single workflow to reduce configuration redundancies [SMM$^+$12]. Dependencies between stakeholders lead to a tree-like structure of the workflow, where each workflow path leads to a complete variant configuration of the feature model. Section 6.5 explains the concepts of specialization trees. A concept to support *reconfiguration* of partial feature model configurations in a specialization tree is introduced in Section 6.6.

An adaptation engine and a set of *adaptation rewrite rules* allow for a dynamic stakeholder management. During executing the specialization tree, sequences of rewrite rules are applied to integrate and remove configuration stakeholders by adapting the specialization tree. Section 6.7 explains the adaptation of the specialization tree and introduces rewrite rules for dynamic stakeholder management.

6.3. Extended Feature Models

Configuration parameters of reconfigurable cloud applications are modeled in this approach by group-cardinality based feature models with attributes over finite domains. The definition of group-cardinality based feature models given in Section 5.5 is extended with attributes and the capability of expressing feature and attribute configuration states. The abstract syntax of extended feature models is defined by a metamodel. The metamodel depicted in UML class diagram notation in Figure 6.3 defines the structure and well-formedness criteria of attributed group-cardinality based feature models.

The metaclass `FeatureModel` represents the root container node of an extended feature model. This metaclass comprises the root feature of the tree-like feature model. A feature is represented by the metaclass `Feature` and further specified by the attributes `name` and the unique identifier `id`. Features represent functional entities with a configuration state. Hence, features are either *selected*, *deselected* or yet *undecided* in a variant configuration. The configuration state of a feature is represented by the attribute `state` of the enumeration type `ConfigurationState`.

Group-cardinality is a generalization of a unified modeling of features without the need to distinguish between solitary features, i.e., *mandatory* and *optional* features, and groups of features, i.e., *alternative* and *or* relations, as explained in Section 2.3.2. With exception of the root feature, each feature is modeled in a feature group. A feature group is represented by the `Group` metaclass in the metamodel. Features and groups are explicitly modeled as separated elements instead of choosing a composite pattern. Groups define constraints on the number of selectable features in the group in terms of a minimum and maximum cardinality, similar to cardinality in the UML.

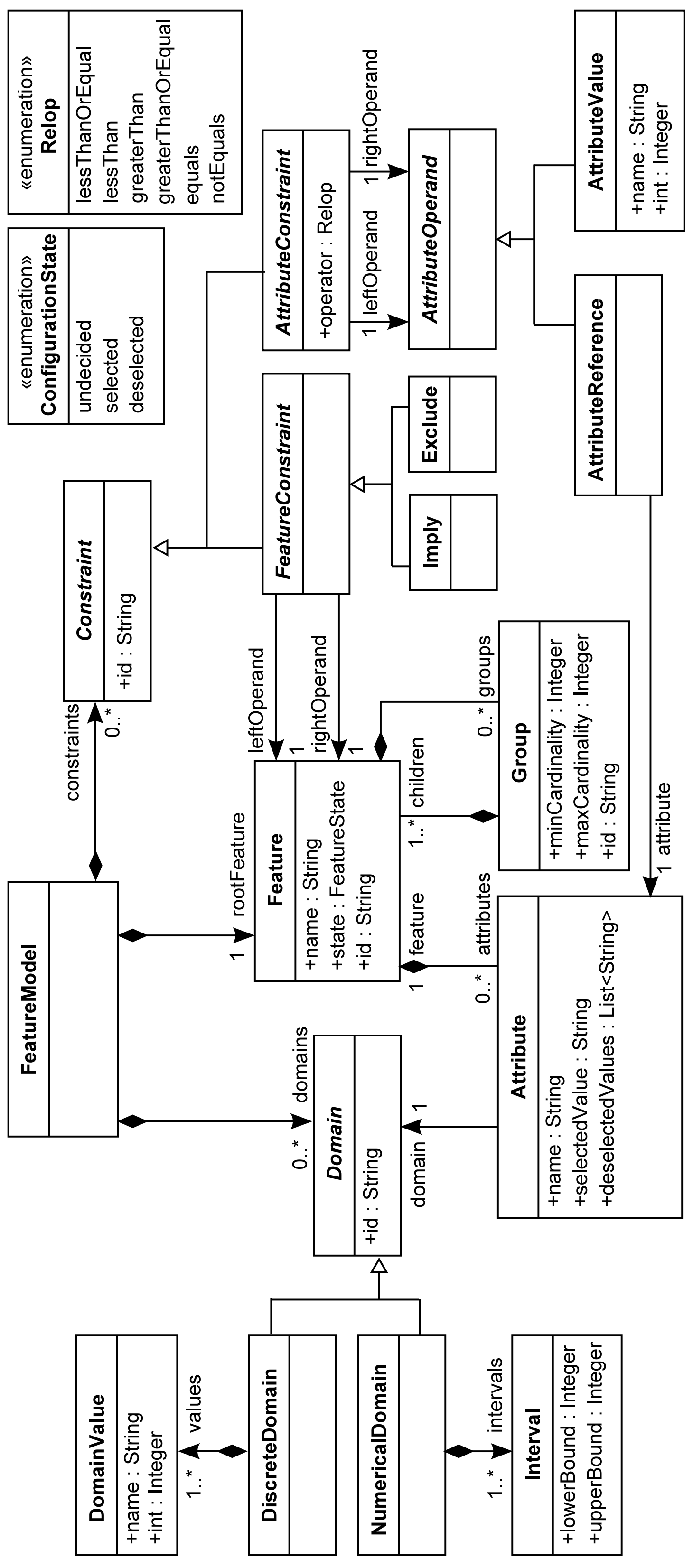


Figure 6.3 Metamodel for attributed group-cardinality based feature models.

In extended feature models, attributes further classify features, where each feature allocates zero or an arbitrary number of attributes, as explained in Section 2.3.2. Feature attributes represented by the metaclass `Attribute` express quantifiable configuration properties. In the context of cloud applications, feature attributes express QoS agreements and numerical application-specific configuration parameters. Configuration parameters are, for instance, the number of users of an application, and the database size.

In the document management example, attributes model application qualities, such as the hardware encryption level and the availability of the application. Features are related to potentially many attributes as specified in the metamodel. A feature may contain several attributes, where each attribute is explicitly contained in a single feature. An attribute further classifies a feature and corresponds to a scalar variable. Hence, a single domain value can be assigned to an attribute modeled by the `selectedValue` property of an `Attribute`. However, the domain of an attribute can be restricted during a configuration process, such that domain values become unavailable for assignment. Domain restrictions are modeled by the property `deselectedValues` of an `Attribute`.

A finite discrete domain is assigned to each attribute according to the explanation in Section 2.3.2. Finite domains are chosen to ensure that attribute constraints are efficiently analyzable by a CSP solver [PBN+11]. In general, each domain must contain at least one domain value. Both domain types are defined globally and referenced by the `FeatureModel`, as globally defined domains are reusable for different attributes. Numerical and discrete domains are explicitly distinguished in the metamodel represented by the metaclasses `DiscreteDomain` and `NumericalDomain`. In a numerical domain, elements correspond to Integers, and are therefore ordered. A numerical domain contains several Integer intervals represented by the corresponding metaclass `Interval`.

The definition of intervals abstract from solitary domain values by specifying lower and upper bound of the interval instead of each single domain value. Hence, intervals ease the specification of large numerical domains in the feature model and allow for an optimized evaluation by a CSP solver. Discrete domains contain a set of domain values represented by the metaclass `DomainValue`. Each domain value is given by a String and a unique Integer index, where indices do not overlap. The order of index values is applied to compare domain values. Hence, domain values of discrete and numerical domains are comparable by means of their Integer representation. In addition, cross-tree constraints on features and attributes are modeled.

A `FeatureModel` contains all cross-tree constraints on features and attributes. Each cross-tree constraint inherits from the abstract `Constraint` metaclass defining the attribute `id` for each constraint instance to be uniquely identifiable. Imply and exclude constraints on features can be represented by the corresponding metaclasses. Both constraint types are interpreted as logical expressions. Hence, both constraint types inherit from the common abstract metaclass `FeatureConstraint`, which references a left and a right `Feature` operand.

Values of feature attributes are constrained by means of conditional expressions [KOD10]. Attribute constraints are represented by the metaclass `AttributeConstraint`, which comprises a relational operator defined by the `Relop` enumeration and references a left and a right attribute operand. An attribute operand is specified by means of the `AttributeOperand` metaclass and may be either a reference to an attribute, or an attribute value. The introduction of the metaclass

AttributeValue allows to specify constraints on attributes with discrete, as well as numerical domains. The metaclass is comparable to a DomainValue of a discrete domain. However, in an attribute constraint a value can be used which is not explicitly specified as domain value.

An instance of the metamodel is depicted in Figure 6.4 representing a feature model of a variable document management system. In this example, the number of users concurrently accessing the application instance can be specified by a corresponding attribute. This attribute is related to a numerical domain represented by an instance of the NumericalDomain metaclass. The domain comprises a value Interval with numerical values between 1 and 1000 represented by the corresponding lower and upper bound of the Interval. In an attribute constraint, the

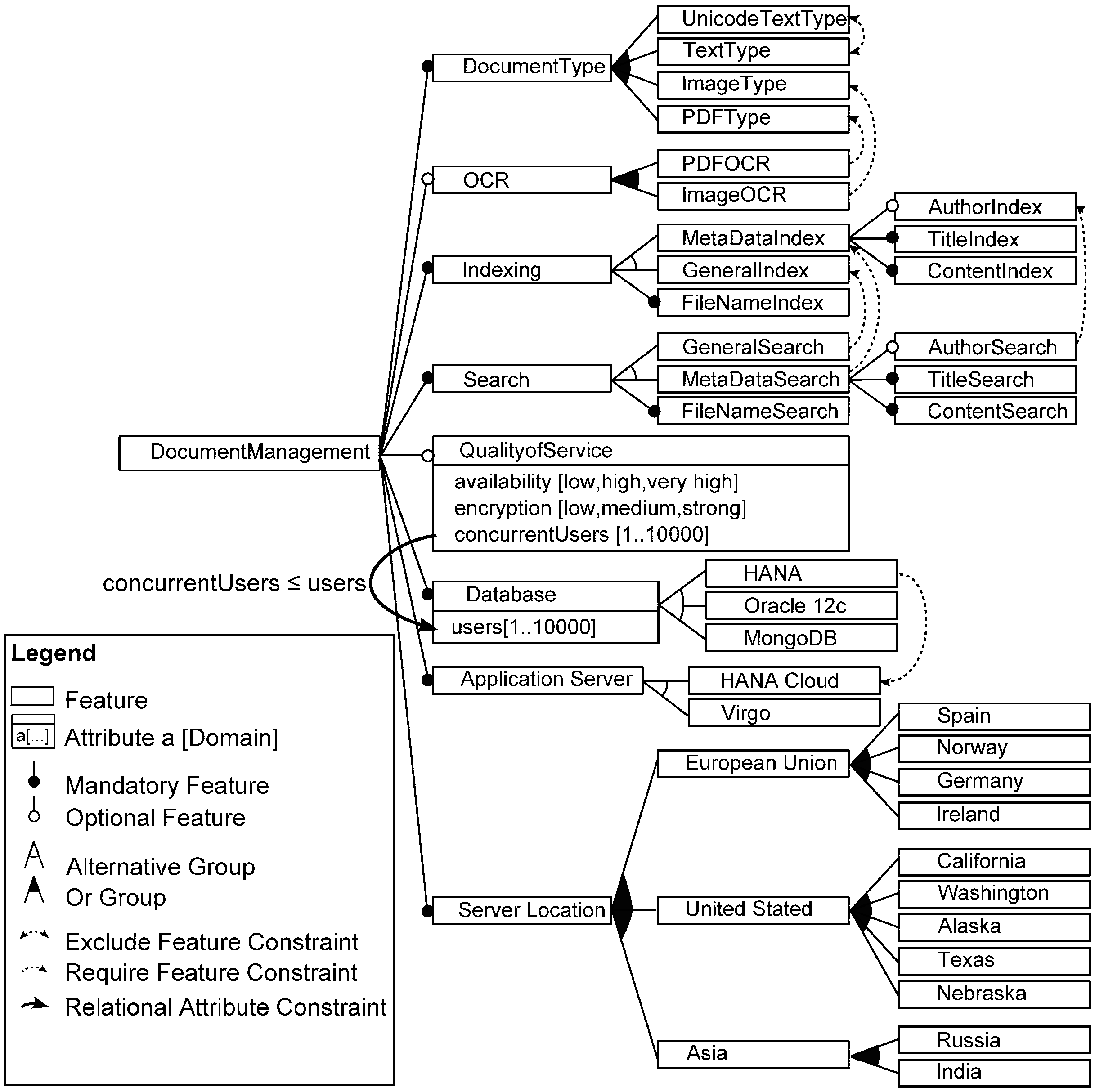


Figure 6.4 Feature model with attributes of a variable document management system.

number of users can be restricted to be lower than 500. This is achieved by modeling this value as an `AttributeValue`. However, this value is not directly represented by an instance of the `DomainValue` metaclass as the constrained domain of the attribute is numerical. In the example in Figure 6.4, an attribute constraint is defined to restrict the number of users currently accessing the application, to be equal or less than the number of users of a database. The corresponding textual representation of the feature model can be found in Appendix B.4.

An instance of the feature model metamodel defines dependencies between features and attributes and expresses their current configuration states. Hence, instances are applicable to define the configuration space of reconfigurable cloud applications. The instance depicted in Figure 6.4, specifies an attributed feature model for the variable document management example introduced in Section 6.1. The example feature model comprises 46 features, 4 attributes, and 8 cross-tree constraints on features and attributes. Application functionality for managing, indexing, and searching different document types are represented by features. In addition, quality of service properties, such as application availability, hardware encryption, and the number of users concurrently accessing the application can be specified.

The feature `Quality of Service` is related to SLA-specific concerns that are explained in Section 1.5.3. This feature comprises attributes that represent SLA related configuration parameters. For instance, the attribute `availability` represents the application availability and implicitly specifies the allowed application downtime when the service is not usable.

The domain of this attribute comprises three domain values `low`, `high`, and `very high`. These domain values are assignable to the attribute and represent the corresponding availability levels. Furthermore, platform specific features, such as different application servers to deploy the application and data bases for storing indexed documents are configurable. Server locations are modeled as well, as cloud applications run in a virtual cloud environment, which is spread over multiple data centers.

The process of conducting configuration operations on a feature model is referred to as *specialization* in the application engineering phase, as explained in Section 2.4. In this process, configuration operations are conducted that change the states of features and attribute, and result in partial and eventually complete variant configurations. Configuration operations on extended feature models are explained in the following.

6.3.1. Configuration Operations on Extended Feature Models

In related work, a variant configuration is often represented as a set of features and qualified attributes without considering configuration states [BSRC10]. The metamodel depicted in Figure 6.3 allows for explicitly modeling feature and attribute configuration states. Configuration states are changed by applying configuration operations on features and attributes. In this approach, partial and complete variant configurations of an extended feature model are distinguished by means of feature and attribute states. In a complete variant configuration, all features and attributes are in a final state, while in a partial configuration some are still undecided.

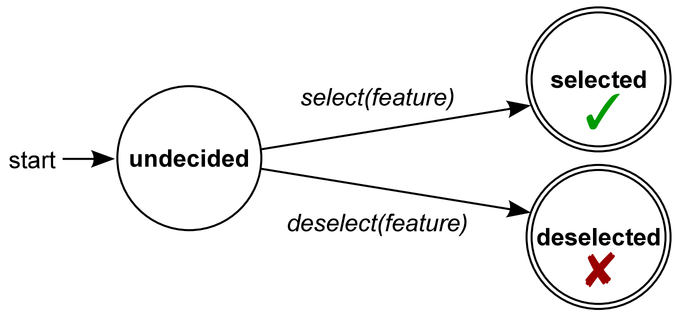


Figure 6.5 Feature configuration operations change the configuration state of a feature.

In a complete product configuration, each feature must be either *selected* or *deselected*. Hence, two mutually exclusive configuration operations are defined per feature, *select(feature)* and *deselect(feature)* [LMSW13]. Both operations can only be conducted on a feature that is in the *undecided* configuration state.

A feature is in the state *undecided*, if it is not yet decided whether to include, or discard the feature from the complete variant configuration. Figure 6.5 applies the notation of a Moore automaton to depict the configuration states of a feature, and the configuration operations leading to state transitions. The *selected* and *deselected* states are final states of a feature with respect to a variant configuration. For instance in the document management example, for feature `Search` the configuration operations *select(Search)* and *deselect(Search)* are specified to change the configuration state of this feature.

Figure 6.6 depicts the configuration states of an attribute and the state transitions in the configuration process. An attribute is in the *undecided* state, if no value is selected yet, in the state *assigned* when an attribute value is selected, and *disabled*, if the attribute feature is deselected. An attribute further classifies a feature. Hence, an attribute value assigned to an attribute is only evaluated, if the attribute's feature is selected. The configuration state of an attribute depends on the domain values and on the configuration state of the attribute feature.

In contrast to features, attributes are not just selectable or de-selectable for variant configurations [BTRC05]. A finite domain is defined for each attribute specifying a set of values assignable to the attribute. Configuration operations are defined on the domain values of an attribute. Two mutual exclusive configuration operations are defined per domain value, *select(attribute value)* and *deselect(attribute value)*.

Assigning a domain value to an attribute changes the attribute state to *assigned*, and is either achieved by deselecting all domain values except one, or by explicitly selecting an attribute value. If all domain values are deselected except one, the remaining value is assigned to the attribute. An assigned attribute has a single domain value assigned and comprises potentially multiple deselected domain values. The domain of an attribute contains at least one value as defined in the metamodel for extended feature models. Hence, a domain can never comprise 0 attribute values which is therefore omitted in Figure 6.6.

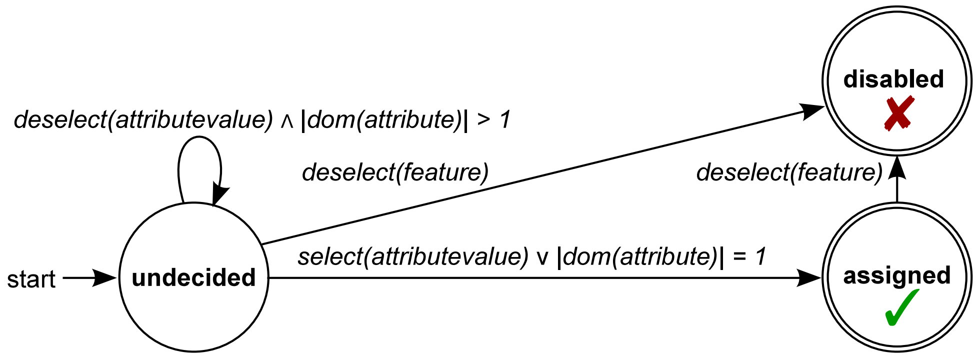

deselect(attributevalue) ∧ |dom(attribute)| > 1
deselect(feature)
deselect(feature)
disabled
start
undecided
select(attributevalue) ∨ |dom(attribute)| = 1
assigned

Figure 6.6 Configuration states of an attribute depend on configuration operations on attribute values and on the configuration state of the related feature.

Attribute states are explained by an example from the document management system. For instance, feature `Quality of Service` has the attribute `availability`. The values of this attribute are defined by a discrete domain containing three values `low`, `high`, and `very high`. If the configuration operation *deselect(low)* is conducted on the `availability` attribute, the attribute remains in the *undecided* state, while the domain value `low` is in the set of deselected attribute values, and hence, cannot be assigned to this attribute.

Conducting the same deselect configuration operation on the domain value `high` implies that the remaining attribute value `very high` is assigned to the attribute. Thus, by assigning this value, the configuration state of the attribute changes to *assigned*. As an attribute further characterizes a feature, the attribute value must be assigned, if the feature is selected. The configuration operation *deselect(feature)* causes the attribute to be disabled regardless wether it is already assigned or not.

Summarizing, configuration operations are only applicable if the feature or attribute the configuration should be applied on is in the *undecided* state. In addition, a configuration operation must preserve feature model integrity. Hence, after conducting a configuration operation, the resulting feature model instance must be satisfiable and allow for deriving a valid variant configuration. Satisfiability can be ensured by applying a CSP solver, as explained in Section 2.5.

The access on configuration operations on features and attributes must be restricted in a specialization process where multiple stakeholders are involved. In the document management system example, for instance, users are not allowed to configure QoS-related configuration parameters. Only customers are allowed to specify these parameters as part of their SLA with an application provider. The next section discusses how to constrain the access on feature and attribute configuration operations.

6.4. Access Control on Extended Feature Models

In multi-user environments, and especially in shared cloud environments, operations on a system need to be restricted to prevent misuse, as discussed in Section 1.7. RBAC concepts are applicable

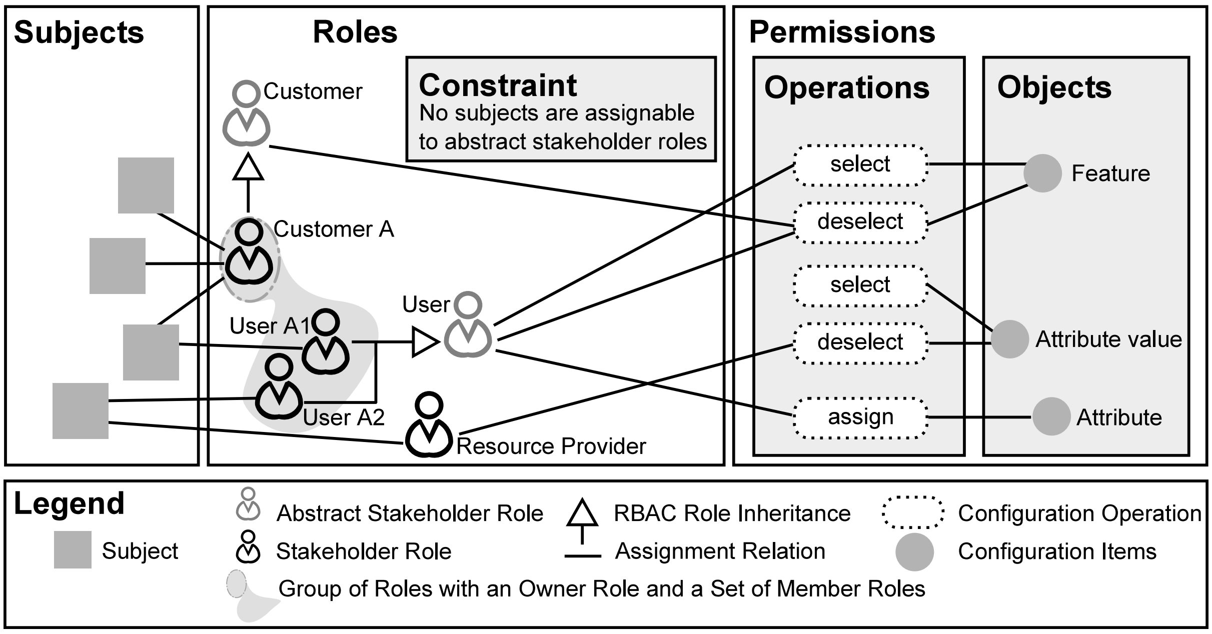


Figure 6.7 Schematic representation of RBAC on feature models.

to implement a fine-grained security policy for restricting the access on feature models related information in common SPL tasks [MS13]. Configuration tasks on extended feature models are restricted by applying $RBAC_3$ paradigms in this work. $RBAC_3$ comprises the concepts of subjects, roles, permissions on objects, role hierarchy and role constraints. The additional concepts of *role groups* and the distinction between *abstract* and *concrete* roles are introduced in this work to model further dependencies between stakeholders participating in configuration processes of cloud applications. The concepts are schematically depicted in Figure 6.7.

In this work, configuration items of an extended feature model represent RBAC *objects*. Hence, RBAC objects are features, attributes, and attribute values. *Permissions* refer to configuration operations applicable to these configuration items, as explained in Section 6.3.1. Configuration operations are select and deselect of a feature or attribute value, and the assignment of an attribute. For features, permissions on the configuration operations `select(feature)` and `deselect(feature)` are defined, encapsulating feature configuration operations. For attributes, a permission on the configuration operation `assign(attribute)` comprises all permissions on the attribute's domain values. Hence, instead of specifying all permissions on attribute values, this permission is applicable. However, permissions are fine-granularly defined on the configuration operations of attribute values as well. These operations are `select(attribute value)` and `deselect(attribute value)`.

An RBAC *role* has multiple permissions and the same permission is assignable to various roles. Stakeholders participating in the configuration process of an extended feature model are represented by roles. The concept of *role hierarchy* specifies an inheritance relation among roles, where multiple role inherit the permissions of potentially multiple parent roles. Hence, a role hierarchy allows for SoC on permissions. *Abstract* and *concrete* roles are introduced to distinguish between concrete stakeholders that participate in the configuration process, and abstract stakeholder types applied to define configuration permissions.

For instance, the configuration stakeholders of the document management system example, described in Section 6.1, are modeled as roles. As depicted in Figure 6.7, `Customer A` is modeled as a concrete role. This role inherits configuration permissions from the abstract role `Customer`. Hence, the abstract role `Customer` encapsulates permissions of all concrete customer roles. The same holds for the abstract role `User` and the concrete roles `User A1` and `User A2`. Roles have different permissions. For instance, in the document management system, the deselection of data centers is only allowed by resource providers. In addition, users of a customers are not allowed to change quality of service parameters.

Particular persons or automata play a role to conduct configuration operations on the extended feature model. These persons or automata are modeled as *subjects*. Subjects can only be assigned to concrete roles in the access control model. Hence, a constraint is introduced that abstract roles cannot be assigned to a subject. According to the document management example, only the concrete role `Customer A` can be assigned to a subject, while the abstract role `Customer` cannot be directly assigned.

Stakeholders participating in the configuration process, such as `User A1` and `User A2`, are modeled as concrete roles instead of subjects as beside the inherited permissions from an abstract parent role, concrete roles can have further permissions directly assigned. In RBAC permissions are always assigned to roles and not to subjects.

Additionally, the concept of *role groups* is introduced to define a relation between roles without inheriting permissions. Role groups represent organizational relations between roles. Each role group has a single *owner* role and multiple *member* roles. A constraint is defined, that roles participating in a role group must be concrete roles, and an owner of a role group cannot be member of the same group. In the document management example, the relationship between a customer and its users is represented as a role group. For instance, role `Customer A` is the owner of a role group comprising the users `User A.1` and `User A.2` in Figure 6.7. The group relation between concrete stakeholders is relevant for evaluating dependencies in a structured configuration process as owners of a group are higher proritized in conducting configuration operations than their members. For instance, the members `User A.1` and `User A.2` are dependent on the configuration decisions of their owner `Customer A`.

Access control concepts on extended feature models are modeled in an *access control model*. The structure of access control models and well-formedness criteria are specified by means of a metamodel. The metamodel is depicted in UML class diagram notation in Figure 6.8. The metaclass `AccessControlModel` is the root container of an access control model. This metaclass comprises the metaclasses `Subject, Role` and `Permission` introduced by RBAC. Structure and relation of these metaclasses are taken from the RBAC specification, as explained in Section 1.7.1.

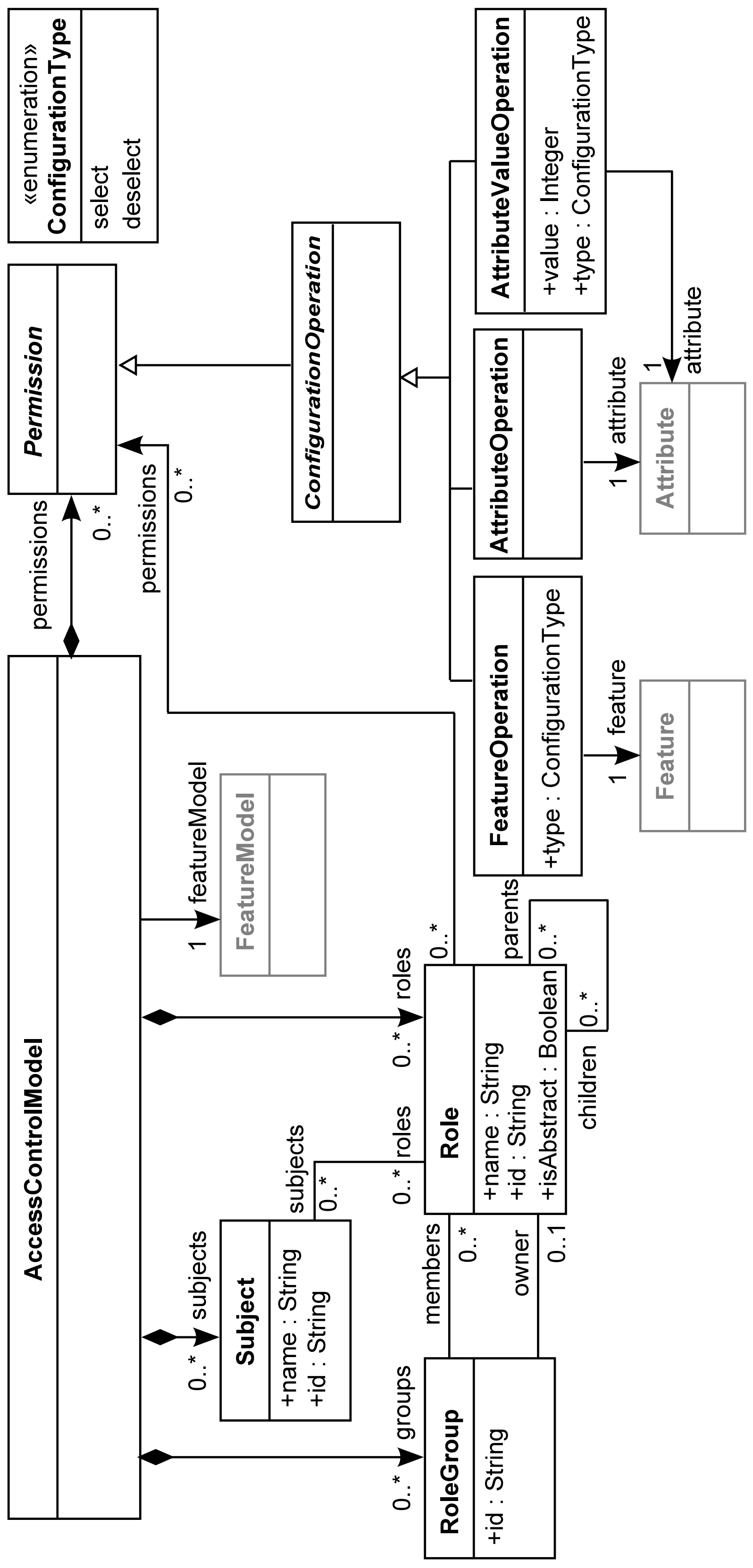


Figure 6.8 Metamodel for restricting access on feature and attribute configuration operations.

In this approach, RBAC is extended with role groups, the differentiation between abstract and concrete roles, and feature model-specific configuration operations. Hence, an `AccessControlModel` further contains a set of role groups represented by the metaclass `RoleGroup` specifying organizational relations between concrete roles. Each `RoleGroup` references a set of `Roles` via a `member` relationship and at most one `Role` via an `owner` relationship.

Furthermore, configuration operations on referenced features and attributes are defined by corresponding metaclasses. The select and deselect feature operations are represented by the metaclass `FeatureOperation`, where the enumerated attribute `type` specifies if the feature operation is a *select* or *deselect* operation. Configuration operations on attribute values are defined similarly by the metaclass `AttributeValueOperation` but refer to a corresponding attribute instead of a feature. The `AttributeOperation` metaclass is defined to abstract from all configuration operations on attribute values.

An instance of the access control model defining permissions on configuration operations of the document management feature model is depicted in textual notation in Listing 6.1. This instance defines permissions on the document management feature model shown in Figure 6.4. The concrete syntax rules of the textual notation is described in Section 7.3.1. Permissions on feature and attribute configuration operations of the document management example are restricted to stakeholders as explained in Section 6.1. Unique element identifiers are depicted in angle brackets and applied to reference model elements, while element names are specified in quotation marks.

```
access control on <extended_dms.eft>

abstract role "Provider" <Provider>

role "Resource Provider" <ResourceProvier> extends Provider {
// deselect data center due to maintenance
  deselect ESP, deselect GER, deselect NOR, deselect IRL,
  deselect CA, deselect WA, deselect AK, deselect TX,
  deselect NE, deselect RUS, deselect IND
}
role "Application Provider" <ApplicationProvider> extends Provider {
  // root feature of the application
  select dms,
  // platform services
  select AppServer, deselect AppServer,
  select HANACloud, deselect HANACloud,
  select Virgo, deselect Virgo,
  select DB, deselect DB,
  select HANA, deselect HANA,
  select Oracle, deselect Oracle,
  select Mongo, deselect Mongo,
  assign DB.users
}
abstract role "Functionality Configuration" <FunctionalityConfiguration> {
  // application functionality
  select DocumentType, deselect DocumentType,
  select TextType, deselect TextType,
  select ImageType, deselect ImageType,
  select PDFType, deselect PDFType,
  select OCR, deselect OCR,
  select PDFOCR, deselect PDFOCR,
  select ImageOCR, deselect ImageOCR,
  select Indexing, deselect Indexing,
  select MetaDataIndex, deselect MetaDataIndex,
  select AuthorIndex, deselect AuthorIndex,
  select TitleIndex, deselect TitleIndex,
  select ContentIndex, deselect ContentIndex,
  select GeneralIndex, deselect GeneralIndex,
  select FileNameIndex, deselect FileNameIndex,
  select Search, deselect Search,
  select MetaDataSearch, deselect MetaDataSearch,
  select AuthorSearch, deselect AuthorSearch,
  select TitleSearch, deselect TitleSearch,
  select ContentSearch, deselect ContentSearch,
  select GeneralSearch, deselect GeneralSearch,
  select FileNameSearch, deselect FileNameSearch,
  select UnicodeTextType, deselect UnicodeTextType
}
abstract role "Customer" <Customer> extends FunctionalityConfiguration {
  // quality of service properties
  select QoS, deselect QoS,
  assign QoS.availability,
  assign QoS.encryption,
  assign QoS.concurrentusers,
  // preferred server location
  select EU, deselect EU,
  select US, deselect US,
  select AS, deselect AS,
  select ESP, deselect ESP,
  select GER, deselect GER,
  select NOR, deselect NOR,
  select IRL, deselect IRL,
  select CA, deselect CA,
  select WA, deselect WA,
  select AK, deselect AK,
  select TX, deselect TX,
  select NE, deselect NE,
  select RUS, deselect RUS,
  select RUS, deselect IND
}
abstract role "User" <User> extends FunctionalityConfiguration

role "Customer A" <CustomerA> extends Customer
role "Customer B" <CustomerB> extends Customer
role "User A1" <UserA1> extends User
role "User A2" <UserA2> extends User
role "User A3" <UserA3> extends User
role "User B1" <UserB1> extends User
role "User B2" <UserB2> extends User

group "Application" <groupApp> of ApplicationProvider has members CustomerA,
    CustomerB
group "Company A" <groupA> of CustomerA has members UserA1, UserA2, UserA3
group "Company B" <groupB> of CustomerB has members UserB1, UserB2

subject "John Smith" <js> plays UserA1
```

Listing 6.1 Example of an access control model defining roles and their permissions on configuration operations of a document management system.

Stakeholders introduced in the document management example in Section 6.1 are modeled by roles in this access control model instance. Configuration operations are assigned to these roles to restrict their access. For instance, the role `Resource Provider` is only allowed to deselect features representing data center locations. The role `Application Provider` is allowed to select the root feature of the application and features representing platform services, such as databases and application server.

In the access control model, all stakeholders of the document management system example, introduced in Section 6.1, are modeled as concrete roles. Hence, nine concrete roles are defined, which are `Resource Provider`, `Application Provider`, `Customer A`, `Customer B`, `User A1`, `User A2`, `User A3`, `User B1`, and `User B2`. In this example, `Customer A` has the same permissions as `Customer B`. Hence, a hierarchical role `Customer` is defined for specifying configuration operations of all roles representing customers. The same holds for the five users which inherit their permissions from an abstract role `User`.

In addition, an abstract role `FunctionalityConfiguration` is introduced for comprising permissions on features representing application functionality. The abstract roles `Customer` and `User` represent application customers and their users. These roles inherit the permissions of the abstract role `FunctionalityConfiguration`, which is introduced to encapsulate the permissions on features representing application functionality.

Particular customers and users of the application are modeled by concrete roles in the access control model. For instance, the role `Customer A` represents the corresponding stakeholder depicted in Figure 6.1. The roles `Customer`, `User`, and `FunctionalityConfiguration` are abstract and hence, not directly assignable to subjects. Only concrete roles are assignable to subjects. In this access control model, a subject `John Smith` is defined playing the role `User A1` to conduct configuration operations allowed for this role.

Configuration operation of stakeholders can be prioritized by means of a *staged configuration process*, as explained in Section 2.4.2. The next section explains how to integrate the proposed RBAC concepts into staged configuration processes.

6.4.1. Access Control in a Staged Configuration Workflow

A workflow language is applicable to define the behavior of staged configuration processes, as explained in Section 2.4.2. Executing the workflow definition leads to a complete variant configuration.

In configuration processes of cloud applications, multiple stakeholders depend on the partial configuration of higher prioritized stakeholders. For instance, in the document management example, explained in Section 6.1, both customers `Customer A` and `Customer B` depend on the configuration decisions of the `Application Provider`. Thus, the configuration operations of some stakeholders, such as `Resource Provider` and `Application Provider`, have an impact on the configuration operations of depending stakeholders. Configuration operations of stakeholders can also be independent of each other. For instance, the configuration operations of `Customer A` do not influence the configuration operations of `Customer B`. The same holds for the subsequent

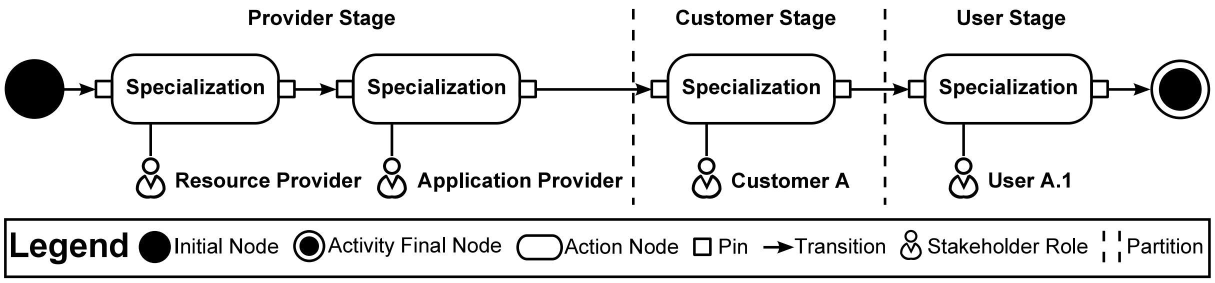


Figure 6.9 Executing a staged configuration workflow leads to a single complete variant configuration.

configuration operations of users, which depend on customer pre-configurations. However, the configuration process results in user-specific variant configurations. Each variant configuration specifies functionality and quality of the document management application available for the corresponding user of the application.

Applying staged configuration to the described cloud scenario, a first approach is to model a workflow per user leading to a complete configuration. A staged configuration workflow can operationally be specified by means of a UML activity diagram as explained in Section 2.4.2. Figure 6.9 shows an example staged configuration workflow for deriving a single complete variant configuration for `User A.1` in UML activity diagram notation. The *initial node* depicts the start of the workflow, where the *activity final node* specifies the end. A configuration task performed by a stakeholder is represented as an *action node* in the activity diagram. In a configuration task, multiple configuration operations are conducted on an input feature model. Stakeholder configuration operations in a configuration task are restricted by applying RBAC in this work. The permissions are modeled in a access control model, as explained before, and evaluated during configuration.

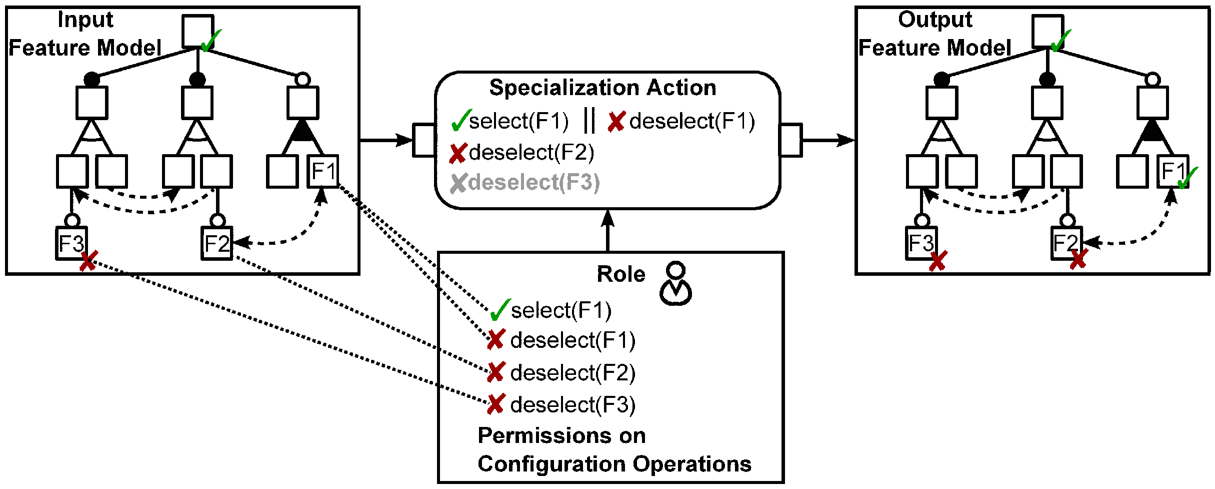


Figure 6.10 Input for a specialization action in the staged configuration workflow are a feature model and a role defined in the access control model.

Input for an action node in the staged configuration workflow is a feature model and a stakeholder role with configuration permissions. Figure 6.10 depicts the input of an action node and the relation between configuration operations on a feature model, a role in an access control model, and an action node in the staged configuration workflow model. The set of permissions assigned to a role defines the stakeholder's configuration view on the feature model in the corresponding configuration task. Hence, a stakeholder is only allowed to conduct configuration operations on a feature model assigned to the corresponding role in the access control model.

However, not all allowed configuration operations are applicable in a configuration task. Inapplicable configuration operations are displayed in gray in the example configuration view of the action node. In this example, the *deselect(F3)* configuration operation is inapplicable as feature *F3* is already deselected in the input feature model. Furthermore, configuration operations on the same attribute and feature respectively are mutually exclusive, for instance, the *deselect(F1)* and *select(F1)* configuration operation in the example in Figure 6.10. A subset of the applicable configuration operations is conducted in a configuration task changing the configuration states of features and attributes, as explained in Section 6.3.1. Hence, output of a specialization action is a refined feature model. In particular, the result of the last action node in the workflow is a complete variant configuration, where all features and attributes are in a final configuration state. In addition, action nodes are sequentially ordered in a workflow to prioritize configuration operations of stakeholders, which are modeled by roles in the access control model. For instance, configuration stakeholders modeled in the example in Figure 6.9 are `Resource Provider`, `Application Provider`, `Customer A` and `User A.1`. Control and data flow between configuration steps by means of directed transitions between action nodes. The control flow creates chronological sequences and logical interdependencies between nodes.

For the example of a document management system, depicted in Figure 6.1, five activities representing staged configuration workflows are to be specified, one per user. However, configuration steps of stakeholders with global impact, such as the `Application Provider` are modeled and executed redundantly. Modeling stakeholder relations in a single staged configuration workflow instead reduces configuration redundancies by reusing partial configurations of particular stakeholders. A special class of staged configuration workflows is introduced, enabling the derivation of multiple complete variant configurations in a single workflow. The workflow class is named *specialization tree*, as the underlying workflow structure corresponds to a is a directed tree, which is introduced in the next section.

6.5. Specialization Tree

Generally, a staged configuration process leads to a single complete configuration by specializing a particular feature model. This concept is extended by forking to support the derivation of multiple complete configurations. The concept of a *specialization tree* is introduced to extend staged configuration workflows by creating a tree-like structure for reusing partial configurations, while leading to multiple variant configurations [SMM+12]. Figure 6.11 depicts the concept of a specialization tree. Specialization trees may have an arbitrary width and depth represented by dots in the figure. However, the depth of a tree is constant, where the number of stages of each path equals. This assumption is necessary to evaluate configuration operations conducted

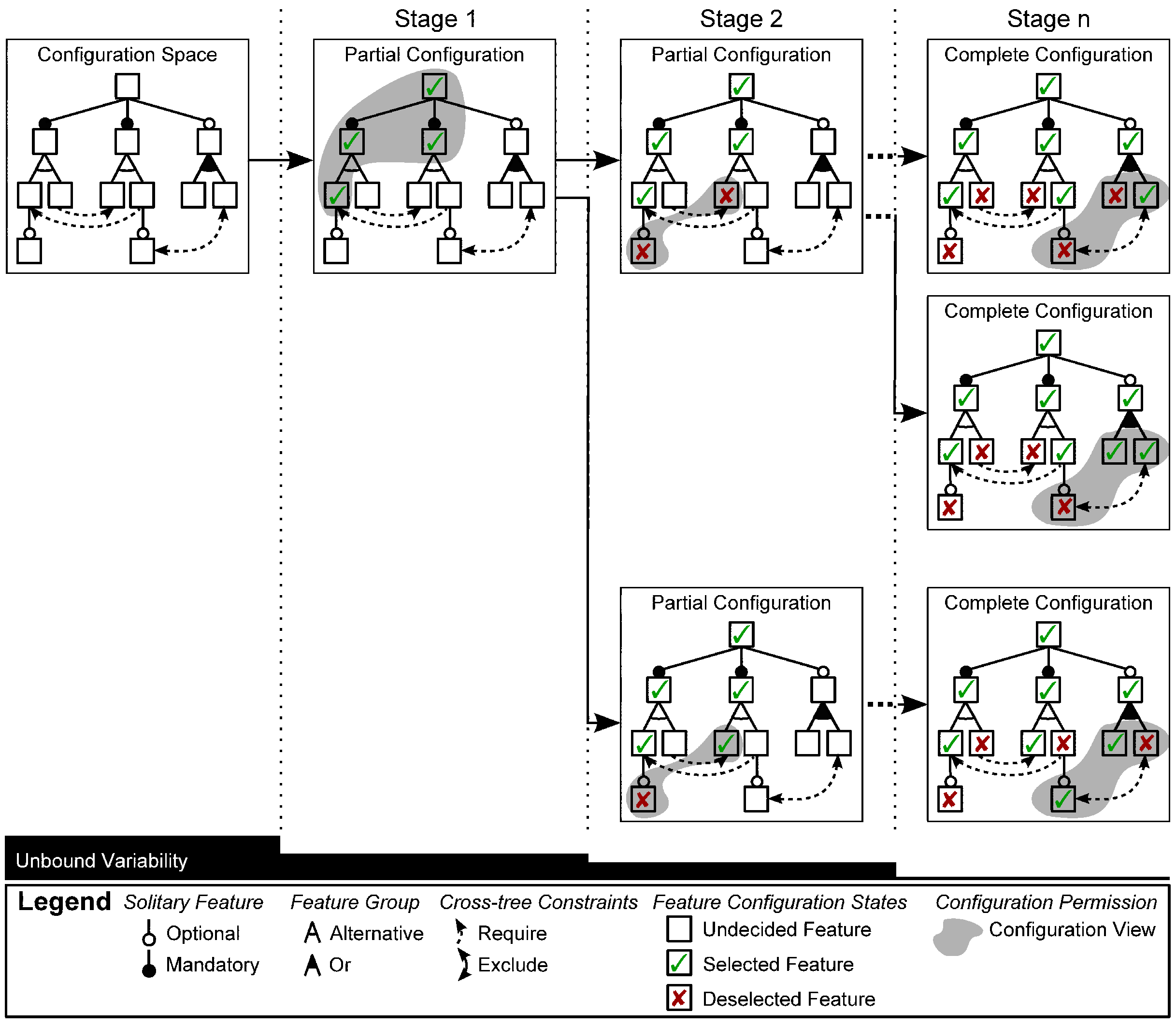


Figure 6.11 A forked staged configuration workflow forms a specialization tree.

in a stage and to identify the final stage after which all variability must be bound in a complete variant configuration. In each stage, the variability defined by the feature model is further refined by means of configuration steps. The result of a specialization step conducted in a stage leads to a partial configuration of the feature model. However, some of these partial configurations are input for multiple further specialization steps where configuration operations are conducted independently. Furthermore, the configuration views of configuration steps of the same stage equal, while configuration steps of a stage are independent.

Definition 6.1 (Specialization Tree). *A staged configuration workflow comprising forks without joins to derive multiple complete variant configurations is a* specialization tree*.*

Modeling a specialization tree as an activity diagram, multiple forks without joins are defined leading to separated independent configurations as shown in Figure 6.12. The activity diagram defines the control flow of the specialization tree. The control flow is separated by *fork* nodes

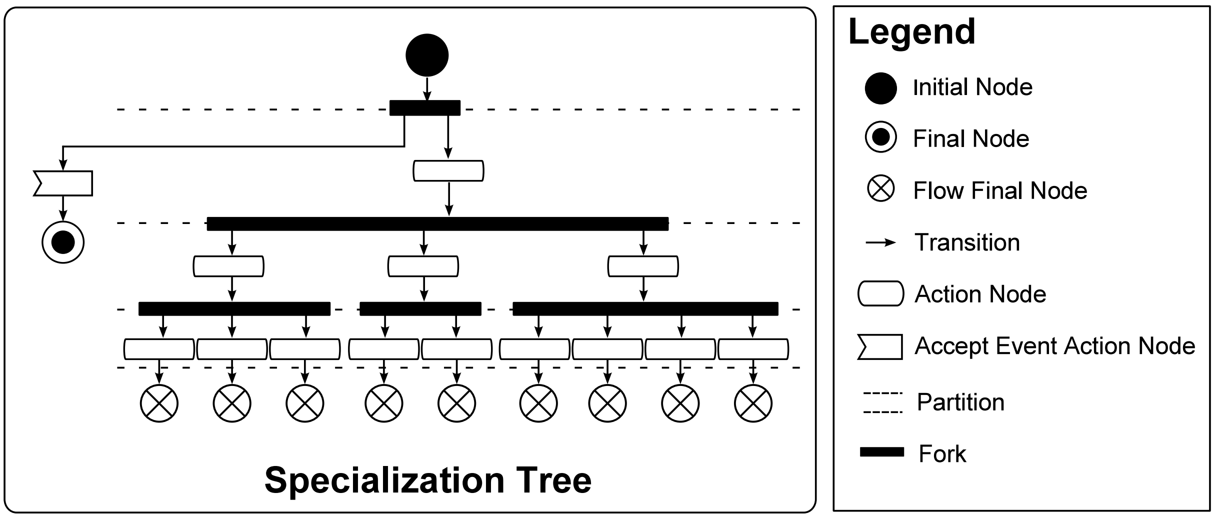


Figure 6.12 Modeling a specialization tree as activity diagram.

leading to separated configuration paths. A fork activates all succeeding paths in the control flow concurrently and has one incoming control and data flow as well as multiple outgoing flows. Each *action* represents a specialization step in a staged configuration process. The UML semantics of a *final node* define that the activity is stopped, when one of its final nodes is reached. Therefore, the notion of a *flow final node* is applied to represent the end of each staged configuration path keeping the workflow alive if a variant configuration is derived. Each path between the *initial node* and a *flow final node* leads to a complete variant configuration and thus equals a staged configuration process in the common sense. Thus, the specialization tree stays alive, when a complete configuration is derived, such that further complete configurations are still derivable.

The underlying tree of the UML activity diagram representation of the specialization tree is a connected directed acyclic graph with a single root node. The root node in a specialization tree is the initial flow node. Each node in the tree is parent of potentially multiple child nodes, whereas in turn each child node only has one parent node. A leaf node is a node without children. Leaf nodes in a specialization tree are flow final nodes and an activity final node. Each flow final node represents the end of a staged configuration process, whereas the activity final node represents the termination of the staged configuration workflow. One path leading from the root initial node to one leaf flow final node represents a staged configuration process for deriving a single complete variant configuration. The structure of a specialization tree definition and well-formedness criteria are defined by means of a metamodel, as explained in the next section.

6.5.1. Structure of Specialization Tree Definitions

A metamodel for specifying the structure and well-formedness criteria of specialization trees is shown in Fig 6.13. The syntax is adopted from the UML 2.4.1 superstructure specification comprising concepts of activity diagrams [OMG2011b]. The root container node comprising specialization tree concepts is the `WorkflowModel` metaclass.

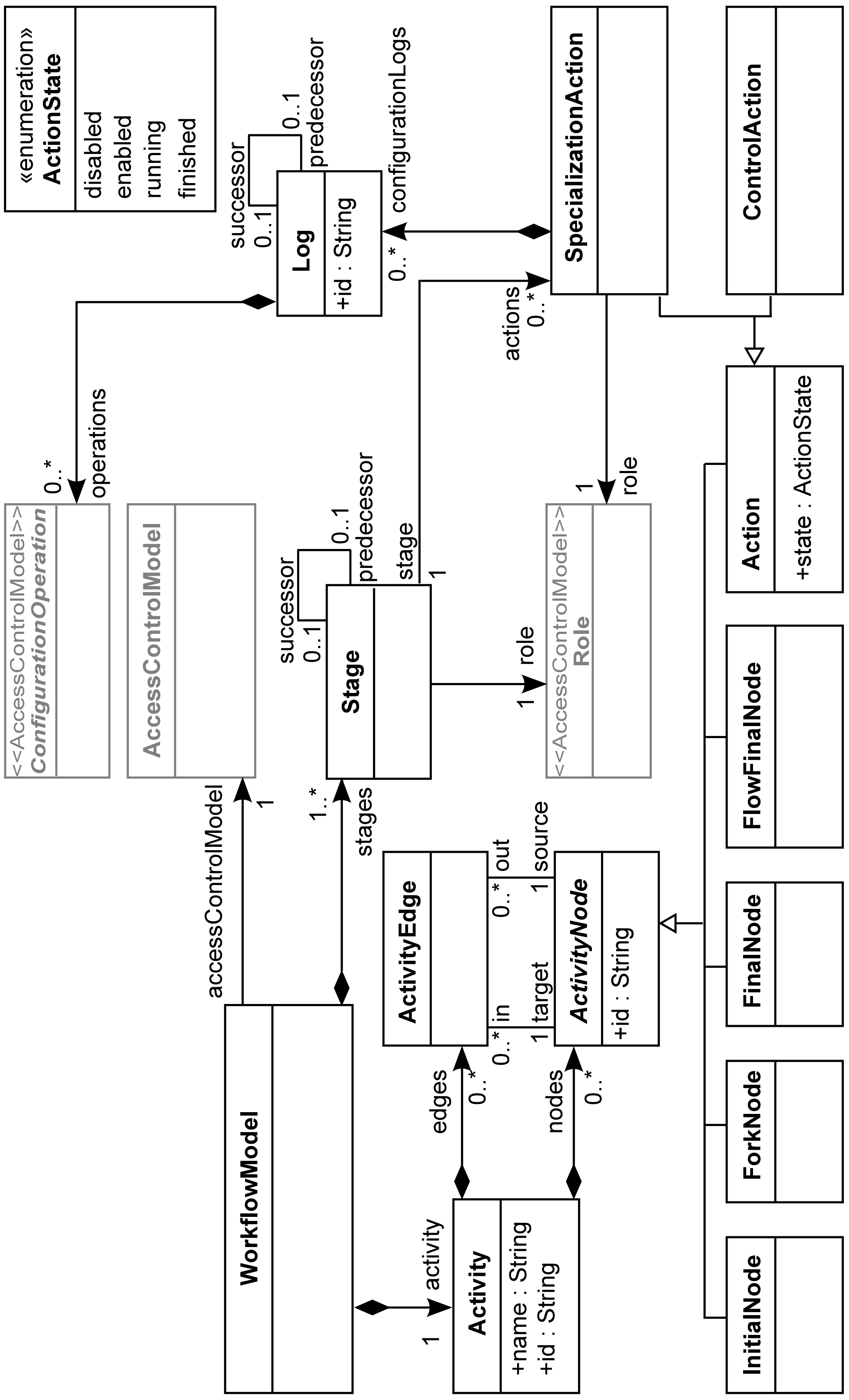


Figure 6.13 Metamodel for staged configuration workflows forming specialization trees.

The metaclasses `Activity`, `ActivitiyEdge`, `ActivityNode`, `InitalNode`, `ForkNode`, `FinalNode`, `FlowFinalNode`, and `Action`, as well as their relations are taken from the UML specification. An `Activity` corresponds to a staged configuration workflow. The `ActivityEdge` metaclass defines directed transitions between a source and a target `ActivityNode`.

An `Action` corresponds to an executable task. To distinguish configuration tasks that refine a feature model from a task to control the termination of the workflow, the two metaclasses `SpecializationAction` and `ControlAction` are introduced, which are derived from the `Action` metaclass.

The `ControlAction` is an accept event action, modeled as a singleton and waiting for a cancel event to terminate the configuration workflow. This action represents an explicit breaking condition to finish the entire configuration workflow. The workflow will only be terminated by triggering the `ControlAction` by an external cancel event. An example for such an external termination event is the notification that the application server shuts down, where the configurable cloud application runs on. The `SpecializationAction` is introduced to execute a configuration task. A configuration task is executed by a stakeholder represented by a concrete `Role` defined in the `AccessControlModel`. The result of a `SpecializationAction` is a partial configuration of the feature model. The `WorkflowModel` metaclass references an `AccessControlModel` to assign `Roles` to `SpecializationActions`. Additionally, a `state` attribute is added to the `Action` metaclass to explicitly model the runtime state of actions.

The `WorkflowModel` comprises a set of ordered `Stage`s partitioning `SpecializationAction`s. A `Stage` subsumes multiple `SpecializationAction`s of the same level in the specialization tree, which are related to concrete stakeholder `Roles` inheriting from the same abstract *Role*. This abstract `Role` is assigned to the `Stage`. In other words, a `Stage` comprises configuration tasks of stakeholders of the same kind. `Succeeding` and `predecessing` relations define an explicit order among `Stage`s.

A specialization tree is *well-formed*, if the underlying directed graph is a rooted tree. Exactly one `InitialNode` must be specified as a root node. Further well-formedness constraints are defined between metaclasses inheriting from `ActivityNode`s and transitions represented by the metaclass `ActivityEdge`. An instance of the `InitialNode` metaclass does not have an incoming transition as this node indicates the start of a workflow. In contrast, the `FinalNode` metaclass representing the termination of the workflow activity and the `FlowFinalNode` representing the end of a single staged configuration process do not have outgoing transitions. Each `Action` has only one incoming and one outgoing transition represented by an `ActivityEdge`. A concrete `Role` contained in the referenced `AccessControlModel` is assigned to every `SpecializationAction`.

A well-formed specialization tree is *valid*, if each path in the tree between an `InitialNode` and a `FlowFinalNode` represents a staged configuration process leading to a complete variant configuration.

An instance of the workflow metamodel models a staged configuration workflow by defining configuration dependencies between stakeholders, and is referred to as *specialization tree definition*. A specialization tree definition is executed as a *specialization tree instance*, where each action has a current runtime state, as explained in the following section.

6.5.2. Specialization Tree Execution

An `Action` has an own lifecycle, which can be expressed by means of states, as depicted in Figure 6.14. Adopted from [RW12], four execution states of an action are distinguished. These states are disabled, enabled, running and finished and modeled in the workflow metamodel as an enumeration type `ActionState`. A configuration task assigned to a `SpecializationAction` can only be conducted if the predecessing `SpecializationAction` is *finished*.

The state of an `Action` changes during the execution of a configuration workflow. Initially, an `Action` is *disabled* and the task represented by the `Action` cannot be executed. Hence, configuration operations cannot be conducted in a disabled `SpecializationAction` accordingly.

A `SpecializationAction` is *enabled*, if the action has no predecessing `SpecializationAction`, or if the predecessing `SpecializationAction` is in a *finished* state. An enabled action is *running*, when a stakeholder starts the configuration task. If a stakeholder completes the configuration task, the `SpecializationAction` is *finished*. If a stakeholder decides to cancel the configuration of a currently running configuration task, the corresponding `SpecializationAction` is *enabled*.

If the state of the `SpecializationAction` changes to `finished`, all `SpecializationActions`, which are direct successors, are implicitly *enabled*. The lifecycle of a `ControlAction` is similar, except that a `ControlAction` is *running* if it receives an external termination event. Changing the state of the `ControlAction` to *finished* terminates the execution of the specialization tree.

A `SpecializationAction` executes a configuration task to reduce feature model variability obeying feature model constraints. Input for the first `SpecializationAction` in the staged configuration workflow is an unconfigured feature model instance. The input for any other `SpecializationAction` is a partial configuration derived from the predecessing `SpecializationAction`. A `SpecializationAction` is executed by a stakeholder represented

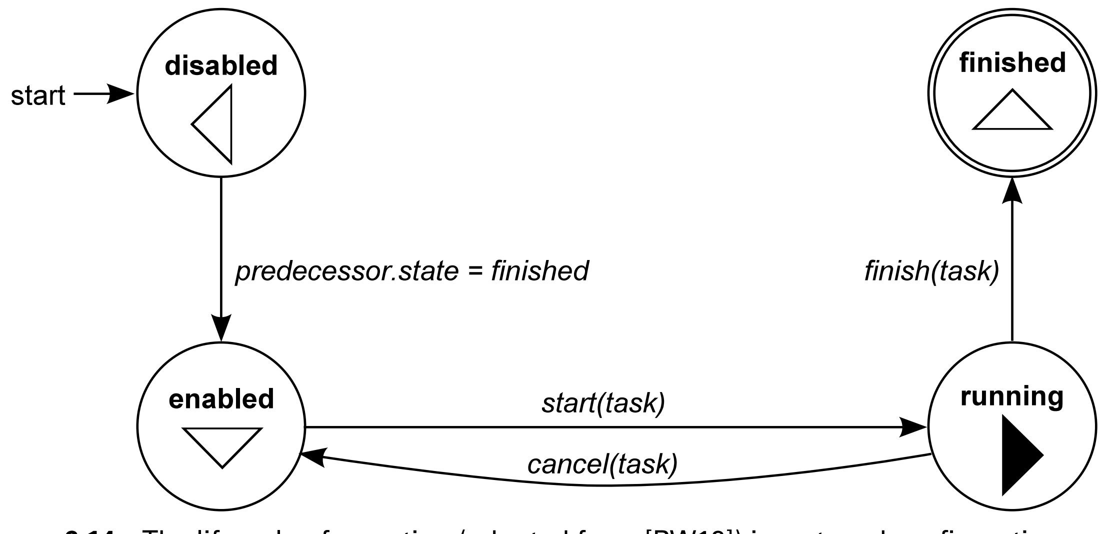


Figure 6.14 The lifecycle of an action (adopted from [RW12]) in a staged configuration workflow.

by the assigned `Role`. The permissions of a `Role` are evaluated to identify allowed configuration operations in terms of RBAC. Allowed operations define a configuration view for the `SpecializationAction`. The configuration view presents all allowed configuration operations to the stakeholder when performing a configuration task. However, not all allowed configuration operations are applicable to the current partial configuration.

Allowed configuration operations are partitioned into a set of applicable and a set of inapplicable configuration operations. The current configuration state of features and attributes is evaluated to identify applicable configuration operations, as explained in Section 6.3.1. Inapplicable configuration operations are highlighted.

Views of stakeholders can overlap due to the overlapping permissions defined for roles in the access control model. Hence, a stakeholder can postpone configuration decisions, if the item is configurable in a succeeding `SpecializationAction`.

To identify postponable configuration operations, the set of applicable configuration operations is compared to the configuration operations allowed in succeeding actions. Furthermore, a stakeholder cannot conduct all allowed operations, as some `ConfigurationOperations` are mutual exclusive, for instance the selection and deselection of features, as explained previously. If a stakeholder finishes the execution of a `SpecializationAction`, conducted configuration operations are persisted in a `Log` and a feature model configuration is derived as output of the action serving as input for potential succeeding `SpecializationActions`.

During executing the workflow, the result of a `SpecializationAction` is either a partial or complete variant configuration. However, the last `SpecializationAction` in a path between the `InitialNode` and a *FlowFinalNode* must define a complete variant configuration. Traversing is defined by following the directed outgoing transition until reaching the descendant node. If the descendant node is a `FlowFinalNode` or a `FinalNode`, the partial configuration of the `SpecializationAction` must be complete, and partial otherwise. In the following the definition and execution of specialization tree definitions are explained by example.

6.5.3. Example Definition and Execution of a Specialization Tree

The definition of specialization trees and their execution is exemplified by the document management system introduced in Section 6.1. The definition of an example specialization tree for the document management system is depicted in Figure 6.15. The specialization tree definition starts with a fork to separate specialization tree from the cancel path comprising an instance of the control action for terminating the workflow. The definition comprises a specialization action node per stakeholder role involved in configuration. The roles *Resource Provider* and *Application Provider* are assigned to the specialization actions of the first configuration stage prioritizing their configuration operations.

A fork after the specialization action related to the *Application Provider* separates control and data flow. The specialization action, related to the two roles *Customer A* and *Customer B*, are modeled in two separated transitions in the *Customer Stage*. Eventually, the *User Stage* is the latest stage to bind remaining variability of the partial configurations, as all specialization actions in this stage are the last of their kind following the directed paths in the specialization tree. Roles *User A.1*, *User A.2*, and *User A.3* in this stage depend on *Customer A* on the prior stage. In addition, the roles *User B.1* and *User B.2* depend on configuration decisions of role *Customer B*. All stakeholder roles of the *User Stage* are allowed to independently perform the same configuration operations specified by the abstract role *User*.

An example execution of the specialization tree depicted in Figure 6.15 is explained in the following. The control action is initially enabled during specialization tree execution, waiting for a terminating event. After executing the control action, the configuration workflow terminates, which is depicted by the activity final node.

The first stakeholder role conducting configuration operations in the example specialization tree is the *Resource Provider*. This role is allowed to deselect features representing data center locations, as defined in the access control model for the document management system depicted in Listing 6.1. Initially, the *Resource Provider* conducts no configuration operations. Hence, the corresponding specialization action is just finished by this stakeholder, which triggers the succeeding specialization action of the *Application Provider* to be enabled. The *Application Provider* is allowed to configure features related to platform services. The *Application Provider* starts the configuration tasks by selecting the root `DocumentManagement` feature. In addition,

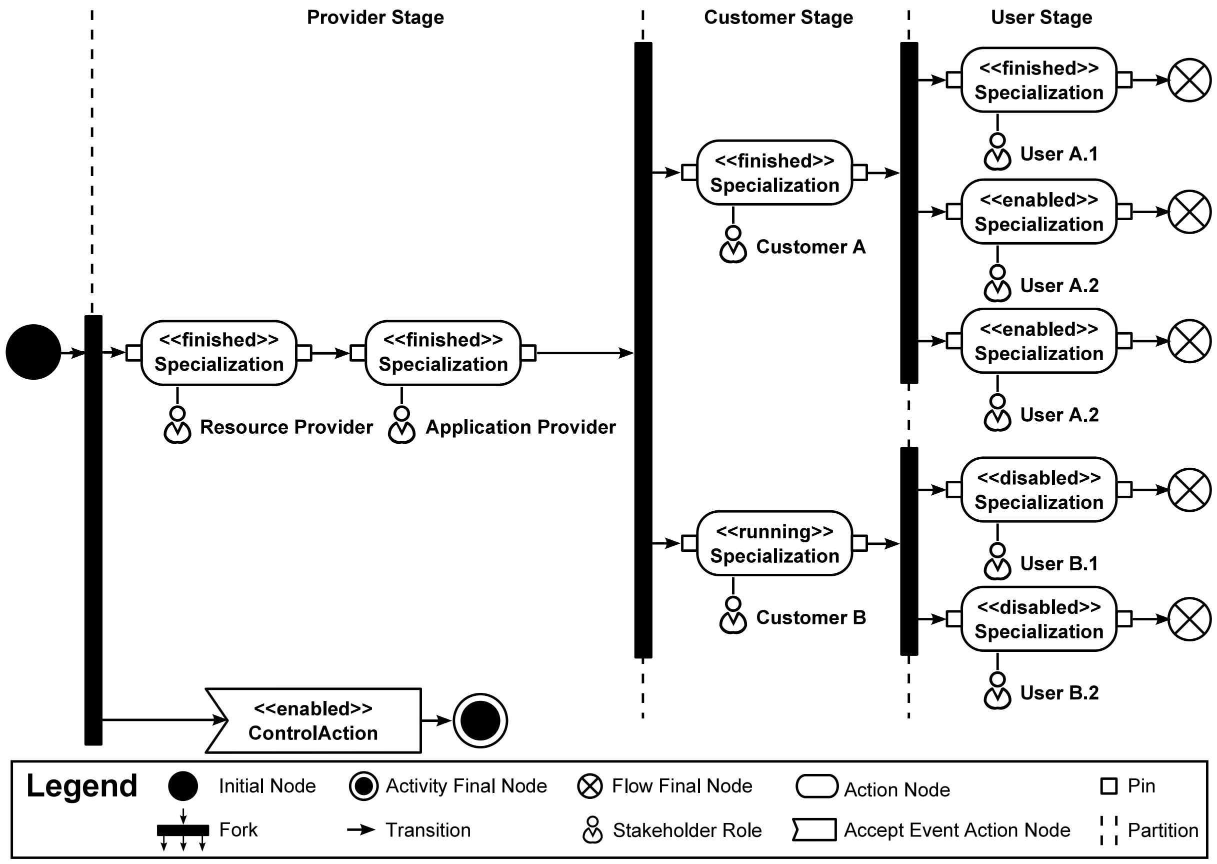

Provider Stage
Customer Stage
User Stage
<<finished>> Specialization
Resource Provider
<<finished>> Specialization
Application Provider
<<finished>> Specialization
Customer A
<<running>> Specialization
Customer B
<<finished>> Specialization
User A.1
<<enabled>> Specialization
User A.2
<<enabled>> Specialization
User A.2
<<disabled>> Specialization
User B.1
<<disabled>> Specialization
User B.2
<<enabled>> ControlAction
Legend
Initial Node
Activity Final Node
Flow Final Node
Action Node
Pin
Fork
Transition
Stakeholder Role
Accept Event Action Node
Partition

Figure 6.15 Specialization tree definition and execution for the document management system example.

```
select "Document Management System" <dms>,
select "Database" <DB>,
select "SAP HANA" <HANA>,
select "Application Server" <AppServer>,
select "SAPA HANA Cloud Platform" <HANACloud>,
select "Server Location" <Location>,
deselect "Eclipse Virgo" <Virgo>,
deselect "Oracle 12c" <Oracle>,
deselect "MongoDB" <Mongo>,
select DB.users.1000
```

Listing 6.2 Executed configuration operations of the role *Application Provider*.

the stakeholder selects the features `Database`, `HANA`, `ApplicationServer`, `HANA Cloud`, and `ServerLocation`, while deselecting `Oracle 12c`, `MongoDB`, and `Virgo`. Furthermore, the stakeholder assigns the attribute value 1000 to the attribute `users` of the `Database` feature. The conducted configuration operations of the *Application Provider* are summarized in Listing 6.2.

In the example specialization tree depicted in Figure 6.15, *Resource Provider* and *Application Provider* have already finished their configuration tasks, which is illustrated by the annotation of the *finished* state at the corresponding specialization action. However, finishing the configuration task of the *Application Provider* triggers the succeeding specialization actions of *Customer A* and *Customer B* to be enabled.

Due to the fork succeeding the specialization action of the *Application Provider*, the resulting partial feature model configuration serves as input for both specialization actions in the customer stage. The configuration operations of the stakeholders *Customer A* and *Customer B* are executed independently without influencing each other. Each specialization action of a customer leads to a single partial configuration, which is further forked for the users of the customers. In this example, only *Customer A* finished the configuration task, while the configuration task of *Customer B* is currently running. Hence, specialization actions related to users of *Customer A* are enabled, while the specialization actions of users related to *Customer B* are disabled, until this stakeholder finishes its configuration task.

Configuration permissions of the *Application Provider* role and the *Customer A* role are disjunct in this example. However, the permissions of the *Customer A* role and the *User A.1* role overlap. Both roles are allowed to conduct configuration operations on features representing application functionality. For instance, both roles are allowed to select and deselect the `OCR` feature. In the example specialization tree, depicted in Figure 6.15, role *Customer A* is higher prioritized than role *User A.1*.

Assuming that *Customer A* deselects the `OCR` feature, *User A.1* and all other subsequent stakeholders cannot conduct a select or deselect operation on this feature. However, *Customer A* is not required to decide on selecting or discarding the `OCR` feature from a configuration. This decision can be left to succeeding specialization actions, as related roles are allowed to configure this feature. Hence, in a configuration task, the applicability of a configuration operation is evaluated, as well as the possibility to discard a configuration decision. All configuration decisions that cannot be discarded must be conducted in the configuration task to derive a valid variant

```
select "Quality of Service" <QoS>,
select QoS.availability.high,
select QoS.encryption.strong,
select QoS.concurrentusers.100,
select "European Union" <EU>,
select "Spain" <ESP>,
select "Germany" <GER>,
select "Norway" <NOR>,
deselect "Ireland" <IRL>,
deselect "United States" <US>,
deselect "California" <CA>,
deselect "Washington" <WA>,
deselect "Alaska" <AK>,
deselect "Texas" <TX>,
deselect "Nebraska" <NE>,
deselect "Asia" <AS>,
deselect "Russia" <RUS>,
deselect "India" <IND>,
deselect "OCR" <OCR>,
deselect "PDF OCR" <PDFOCR>,
deselect "Image OCR" <ImageOCR>
```

Listing 6.3 Executed configuration operations of role *Customer A*.

```
select "Document Type" <DocumentType>,
deselect "Text Type" <TextType>,
select "UnicodeText Type" <UnicodeTextType>,
select "Image Type" <ImageType>,
select "PDF Type" <PDFType>,
select "Indexing" <Indexing>,
select "MetaData Index" <MetaDataIndex>,
select "Author Index" <AuthorIndex>,
select "Title Index" <TitleIndex>,
select "Content Index" <ContentIndex>,
deselect "General Index" <GeneralIndex>,
select "FileName Index" <FileNameIndex>,
select "Search" <Search>,
select "MetaData Search" <MetaDataSearch>,
select "Author Search" <AuthorSearch>,
select "Title Search" <TitleSearch>,
select "Content Search" <ContentSearch>,
deselect "General Search" <GeneralSearch>,
select "FileName Search" <FileNameSearch>
```

Listing 6.4 Executed configuration operations of role *User A.1*.

configuration where all variability is bound. For *Customer A*, the decisions on the `Quality of Service` feature and its attributes, as well as on server location related features must be made, as subsequent stakeholders in the workflow are not allowed to conduct configuration operations on these configuration items.

Assuming that *Customer A* conducts the configuration operations depicted in Listing 6.3, the resulting partial configuration of the document management system allows for deriving 352 different complete variant configurations.

The configuration task of *Customer A* is already finished in this example and the succeeding specialization actions are enabled. In addition, role *User A.1* already finished the corresponding configuration task by conducting the configuration operations depicted in Listing 6.4. Hence, the staged configuration process defined between the initial node and the succeeding flow final node of the specialization action related to *User A.1* is completed.

The conducted configuration operations lead to a complete variant configurations illustrated in Figure 6.16. Other stakeholders depending on *Customer A* may decide to conduct different

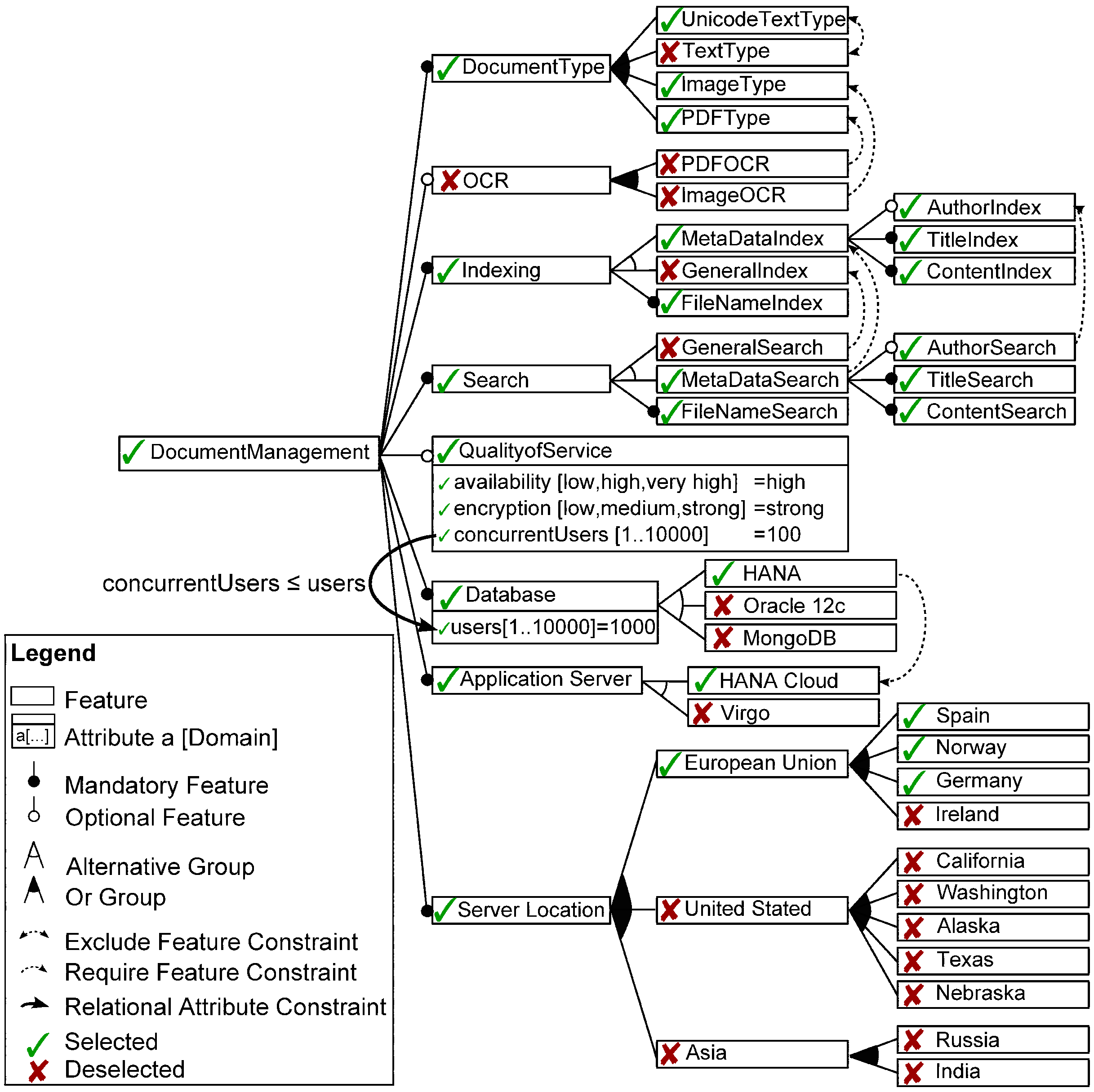


Figure 6.16 Complete variant configuration of *User A.1* of the variable document management system.

configuration operations than *User A.1*. For instance, they may decide to select the `Text Type` feature instead of deselecting it as *User A.1* did, which leads to different complete configurations.

In addition, *Customer B* may conduct different configuration operations than *Customer A* as both roles are independent in the specialization tree. However, both stakeholders have the same permissions as their concrete roles have the same abstract parent role *Customer*. As *Customer B* is currently performing a configuration task indicated by the annotation *running* in Figure 6.15, succeeding roles in the *User Stage* cannot start their configuration tasks yet.

By modeling the dependencies between stakeholders in a specialization tree, partial configurations of stakeholders with high impact are reusable. Another advantage of a specialization tree is the efficient propagating of reconfiguration changes, as discussed in the next section.

6.6. Staged Reconfiguration on Extended Feature Models

Cloud applications that are scalable in terms of functionality and quality are required to support reconfiguration. As multiple stakeholders are involved in the configuration process of a cloud application, as discussed before, reconfiguration changes of a stakeholder are to be propagated to depending stakeholders. For instance, if customers change the tenancy contract signed with a provider to adjust the contract to their current demands, users of these customers are to be notified, and their configurations are changed accordingly. However, these changes are local, as they only affect users of this particular customer. Other reconfiguration decisions have global impact. For example, a resource provider needs to disable data centers due to server maintenance, as explained in the document management example in Section 6.1. Concepts for supporting reconfiguration and propagating changes of feature model configurations in a specialization tree are discussed in the following.

6.6.1. Extending the Action Lifecycle

To support reconfiguration, the lifecycle of actions in the workflow is extended with a further reconfiguration transition, as depicted in Figure 6.17. According to Section 6.5.2 an action is enabled, if the predecessing action in finished. To support reconfiguration an additional transition is added, leading from the *finished* state to the *disabled* state. This transition is triggered if the predecessing action sends a reconfiguration event, or if the stakeholder of this action requests reconfiguration. Subsequently, the reconfiguration task starts and the state of the action changes to *running*. After finishing configuration operations, a reconfiguration event is sent to all direct succeeding `SpecializationActions`. All succeeding `SpecializationActions` that are currently in the *finished* state are revoked to perform reconfiguration.

Succeeding `SpecializationActions` in a *disabled* and *enabled* state are not affected by reconfiguration changes, as no configuration operations are conducted in these actions yet. Therefore, these actions consume the reconfiguration event without further propagating it to their succeeding `SpecializationActions`. A `SpecializationAction` in a *running* state replaces the current input partial configuration with the reconfigured partial configuration and conducted configuration operations are evaluated against the new partial configuration.

The next section discusses how to propagate reconfiguration changes and evaluate configuration operations to preserve feature model integrity.

6.6.2. Reconfiguration Strategies

Three strategies for propagating reconfiguration changes to depending actions are applicable, manual, automated and semi-automated.

In a *manual* strategy, the status of depending specialization actions is changed to *enabled*. Additionally, stakeholders assigned to these actions are notified to restart the specialization. The input for the specialization actions is the changed partial configuration, while already performed configuration operations are discarded. In complex configuration scenarios this strategy is not feasible as stakeholders don't want to execute the same configuration operations multiple times.

Second, an *automated* strategy could be applied. A policy with respect to a certain criteria must be specified therefore. For instance, a policy to change as little as possible in the complete configuration in terms of total number of features, neighboring features, or child features. To define a fully automated approach, the meaning of *similar* features must be specified. Additionally, a metric for analyzing feature similarity must be defined. For instance, a metric based on the structure of the feature model could define features contained in the same group to be similar. However, the notion of similarity depends on the application area of the feature model and is subjective. Thus, it is hard to identify a fully automatic strategy that is applicable in general. Furthermore, stakeholders may not accept a completely automated reconfiguration, as they will get a feeling of loosing control of their configuration operations.

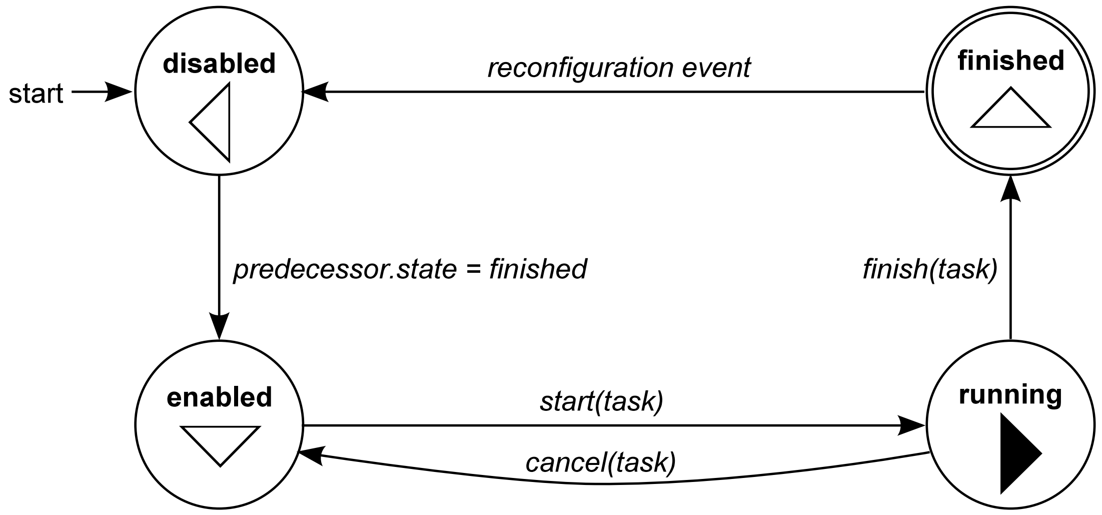


Figure 6.17 The lifecycle of an action is extended with a reconfiguration transition.

Third, a *semi-automated* strategy combines automated and manual concepts. When a reconfiguration event occurs in a specialization action, the operations previously conducted in this action are to be evaluated for compatibility with the new input partial configuration. This is done iteratively by applying each configuration operation in the order they were conducted previously. Configuration operations are inapplicable if they are in conflict with previously conducted configuration operations on the same configuration item or if they violate constraints of the current feature model configuration. To determine if a configuration operation is previously applied on the same configuration item, the current state of the item is analyzed. To determine if a constraint is violated in the current feature model configuration, a CSP solver is applicable [MSD+12]. Inapplicable configuration operations are discarded, where the others are pre-selected in the configuration view of the configuration stakeholder and a notification is displayed accordingly to the configuring stakeholder. In the best case, all configuration operations are applicable, while in the worst case none of them are applicable.

Further input by means of a `Log` is provided to a `SpecializationAction` to support semi-automated reconfiguration depicted in Figure 6.18. A `Log` contains information about previously conducted configuration operations. A metaclass `Log` is defined in the metamodel for staged configuration workflows depicted in Figure 6.13 to store information about conducted configuration operations. Each specialization action references a set of *Log*s. Hence, a `Log` comprises a set of `ConfigurationOperations` defined in the `AccessControlModel`. They represent the configuration decisions made in a `SpecializationAction` during executing the configuration task of a stakeholder. When starting the task of the `SpecializationAction`, a `Log` is instantiated. Every conducted `ConfigurationOperation` is recorded in the `Log`.

A `Log` equals a configuration memento as it records all configuration operations conducted in a configuration task. The relation between successor and predecessor `logs` allows for storing different configuration mementos of the same `SpecializationAction`. Thus, a stakeholder can follow

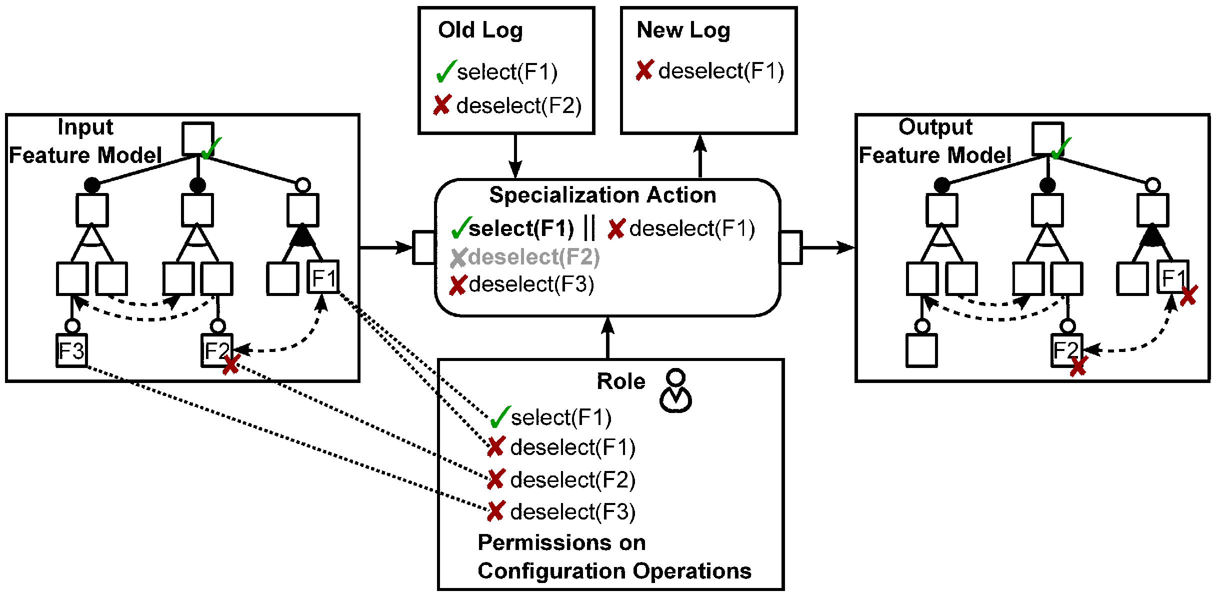


Figure 6.18 Input for a specialization action to support reconfiguration are a feature model, a role defined in the access control model, and a log.

up previously conducted configuration operations. Hence, input of a `SpecializationAction` is the recent `Log` and the partial configuration of the predecessing `SpecializationAction`. Hence, if a recent `Log` exists, each `ConfigurationOperation` contained in the `Log` is checked for its applicability on the input partial configuration of the `SpecializationAction`. In Figure 6.18, previously conducted configuration operations *select(F1)* and *deselect(F2)* are displayed in bold face in the configuration view of the `SpecializationAction`. Inapplicable configuration actions are visualized in gray. In this example, the configuration action *deselect(F2)* was conducted in the previous configuration task, but is no more applicable to the new input feature model as the feature *F2* is already configured. Therefore, this operation is displayed in gray and bold face.

Every `SpecializationAction` can perform a configuration task multiple times while different applied configuration operations lead to different partial configurations as outputs.

6.6.3. Example Reconfiguration

To exemplify reconfiguration, the configuration scenario depicted in Figure 6.15 is assumed. If `Customer A` requests reconfiguration, this reconfiguration event changes the state of the corresponding specialization action from *finished* to *disabled*. As the predecessing specialization action of the *Application Provider* is *finished*, the state of the action assigned to *Customer A* changes to *enabled*. Hence, the stakeholder `Customer A` can start the configuration task, which changes the state of the action to `running`. Input for this task is the partial feature model configuration of the predecessing stakeholder *Application Provider*, a `Log` comprising previously conducted configuration operations, and the permission of the corresponding role *Customer A*. A configuration view is presented to the stakeholder comprising all allowed configuration operations defined by permissions of the role. Figure 6.19 shows an excerpt of the configuration view of *Customer A*. Previously conducted configuration operations are highlighted in bold face. Configuration decisions that cannot be postponed to succeeding specialization actions are visualized by a gray background.

In this example, stakeholder *Customer A* decides to no longer deselect the features *OCR*, *PDF OCR*, and *Image OCR*, leaving the decision whether to include or discard those features from a complete configuration to the succeeding stakeholders. Furthermore, both previously deselected features *United States* and *California* representing server locations are now selected. In addition, the feature *Image Type* is selected. Thus, 576 complete variant configurations are derivable from the output partial configuration of *Customer A*.

In contrast, the prior partial configuration of this stakeholder allowed to derive 352 complete variant configurations only. The reconfiguration of *Customer A* triggers reconfiguration of *User A.1* in this scenario, as this stakeholder already conducted a configuration task. However, this stakeholder is required to decide on the yet unconfigured three features *OCR*, *PDF OCR*, and *Image OCR*, while most of the previously conducted configuration operations of this stakeholder remain applicable to the new input partial configuration. Only the previously conducted select operation of feature *Image Type* is not applicable, as the predecessing stakeholder made a decision on this feature. However, as *Customer A* selected this feature as well, the change did not influence the configuration of *User A.1*. The same configuration operation was conducted at an earlier point in time. Hence, reconfiguration changes vary in their influence on succeeding specialization actions and are classified therefore in the next section.

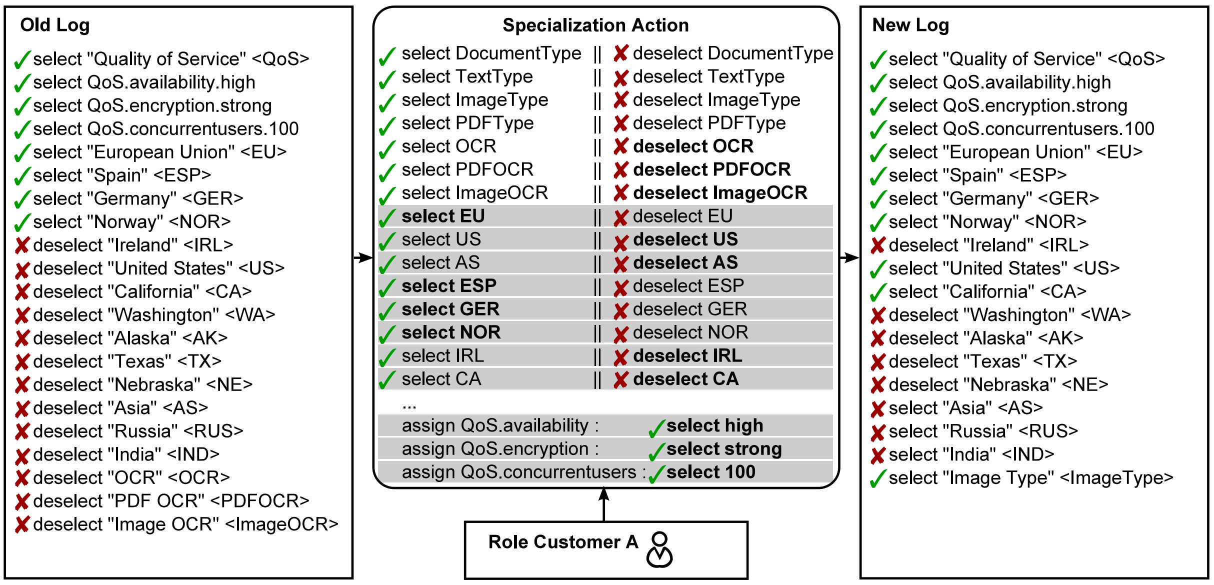
6.6.4. Classification of Reconfiguration Changes

Different types of reconfiguration changes are distinguished with different influence on change propagation. Reconfiguration changes are described in terms of generalization and specialization, as classified in Figure 2.8. Hence, *generalization changes*, *specialization changes* and *switch changes* are to be distinguished. A *generalization change* describes the transition from a final state back to the undecided state of a feature or attribute. In the reconfiguration example explained in Section 6.6.3, generalization changes occurred on the features *OCR*, *PDF OCR*, and *Image OCR*. The state of these features is changed from *deselected* to the *undecided* in the resulting partial configuration.

For a feature, a *specialization change* leads from an undecided state to a final state. In the previous reconfiguration example, a specialization change occurred on feature *Image Type*. The state of this feature is changed from *undecided* to *selected* state.

In addition, a *switch change* defines a transition between the final selected and deselected states of a feature, such that a previously selected feature will be deselected after this change for instance. In the example explained in Section 6.6.3, a switch change occurred on the features *United States* and *California* leading from a *deselected* to the *selected* state.

For an attribute, a *specialization change* occurs, if the attribute is in the *undecided* state and an attribute value is *selected* or *deselected*. If an attribute value is selected, the attribute state



Old Log

✓ select "Quality of Service" <QoS>
✓ select QoS.availability.high
✓ select QoS.encryption.strong
✓ select QoS.concurrentusers.100
✓ select "European Union" <EU>
✓ select "Spain" <ESP>
✓ select "Germany" <GER>
✓ select "Norway" <NOR>
✗ deselect "Ireland" <IRL>
✗ deselect "United States" <US>
✗ deselect "California" <CA>
✗ deselect "Washington" <WA>
✗ deselect "Alaska" <AK>
✗ deselect "Texas" <TX>
✗ deselect "Nebraska" <NE>
✗ deselect "Asia" <AS>
✗ deselect "Russia" <RUS>
✗ deselect "India" <IND>
✗ deselect "OCR" <OCR>
✗ deselect "PDF OCR" <PDFOCR>
✗ deselect "Image OCR" <ImageOCR>

Specialization Action

✓ select DocumentType	||	✗ deselect DocumentType
✓ select TextType	||	✗ deselect TextType
✓ select ImageType	||	✗ deselect ImageType
✓ select PDFType	||	✗ deselect PDFType
✓ select OCR	||	✗ **deselect OCR**
✓ select PDFOCR	||	✗ **deselect PDFOCR**
✓ select ImageOCR	||	✗ **deselect ImageOCR**
✓ **select EU**	||	✗ deselect EU
✓ select US	||	✗ **deselect US**
✓ select AS	||	✗ **deselect AS**
✓ **select ESP**	||	✗ deselect ESP
✓ **select GER**	||	✗ deselect GER
✓ **select NOR**	||	✗ deselect NOR
✓ select IRL	||	✗ **deselect IRL**
✓ select CA	||	✗ **deselect CA**
...		
assign QoS.availability :		✓ **select high**
assign QoS.encryption :		✓ **select strong**
assign QoS.concurrentusers :		✓ **select 100**

New Log

✓ select "Quality of Service" <QoS>
✓ select QoS.availability.high
✓ select QoS.encryption.strong
✓ select QoS.concurrentusers.100
✓ select "European Union" <EU>
✓ select "Spain" <ESP>
✓ select "Germany" <GER>
✓ select "Norway" <NOR>
✗ deselect "Ireland" <IRL>
✓ select "United States" <US>
✓ select "California" <CA>
✗ deselect "Washington" <WA>
✗ deselect "Alaska" <AK>
✗ deselect "Texas" <TX>
✗ deselect "Nebraska" <NE>
✗ select "Asia" <AS>
✗ select "Russia" <RUS>
✗ select "India" <IND>
✓ select "Image Type" <ImageType>

Figure 6.19 Reconfiguration of stakeholder *Customer A* of the document management example.

changes to the assigned state. In contrast, if an attribute value is deselected, the value is added to the set of deselected values without changing the attribute state. A *switch change* applies on attributes in an assigned state. The selected value of an attribute is replaced by a different not yet deselected value.

Summarizing, a generalizing change induces variability, where a specialization change reduces variability. In addition, a switch change maintains the same amount of variability. Generalization changes do not influence the applicability of configuration operations already made in subsequent actions, whereas specialization and switch changes do. However, a generalization change requires to decide on the configuration item at a later time in the workflow. Hence, a generalization change is only possible, if the decision can be made by a succeeding stakeholder.

As exemplified in section 6.1, not all stakeholders are known beforehand and can be modeled in the staged configuration workflow prior execution. Thus, a method to support a dynamic stakeholder management in staged configuration workflows is explained in the next section.

6.7. Workflow Adaptation for Dynamic Stakeholder Management

Managing stakeholders involved in configuring a cloud application requires adaptation of the configuration workflow. Hence, a specialization tree definition, as exemplified in Figure 6.15 for the document management system, cannot always be modeled prior execution. For instance, customers and users subscribe for the cloud application at application runtime. Hence, corresponding roles are to be added to the access control model to define access restrictions on configuration operations, and related specialization steps are to be added in the specialization tree to classify the priority of their configuration decisions. These modifications are to be automated to retain model consistency. An event-based adaptation method by applying graph transformation techniques is introduced in this section to enable dynamic stakeholder management.

Graph transformation approaches are frequently applied in software engineering, especially in MDSD [EEKR99]. The metamodel for specialization trees depicted in Figure 6.13 and the access control metamodel depicted in Figure 6.8 are type graphs, where each specialization tree definition and access control model are instance graphs [GLR$^+$02]. Hence, rewrite rules can be defined on the structure of a type graph to transform an instance graph.

A graph rewrite rule $r \in R$ consists of a left-hand side LHS and a right-hand side RHS pattern [Roz97]. The LHS specifies a search graph pattern and therefore pre-conditions to apply the rule r. The RHS specifies a replacement of this pattern. Hence, applying a rule $r : \text{LHS} \rightarrow \text{RHS}$ on a host graph G replaces the occurrence of LHS in G with RHS yielding a transformed host graph G. The host graph G in this work is a specialization tree referencing an access control model. The adaptation of the specialization tree during runtime is depicted as an UML sequence diagram in Figure 6.20.

A specialization tree instance is adapted during execution according to an *adaptation event*. An adaptation event for a dynamic stakeholder management is the *insertion* and *removal* request of a concrete role into the access control model. An event monitor observes the access control

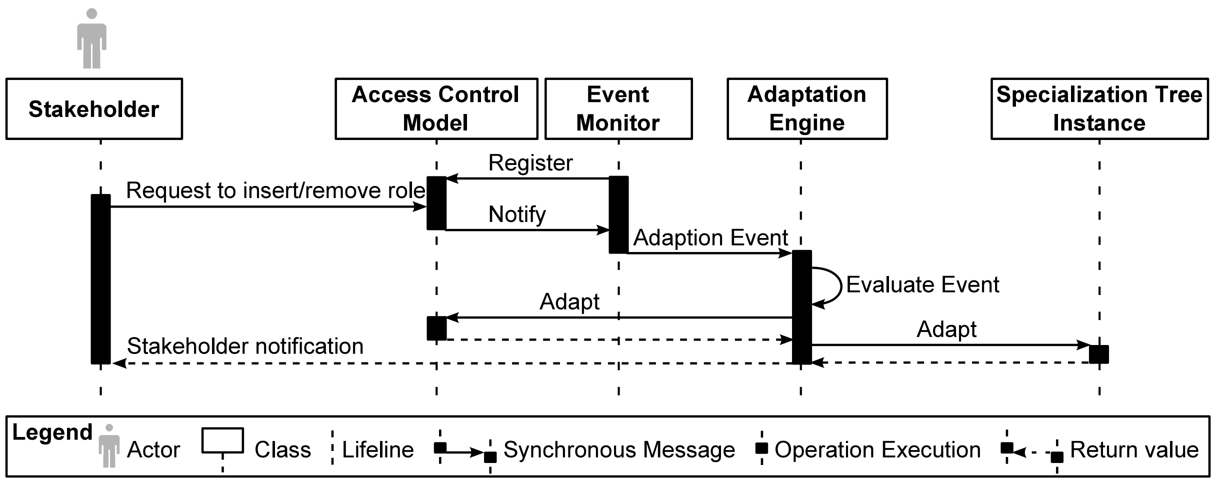


Figure 6.20 Changing a role in the access control model causes the workflow engine to adapt the configuration workflow by applying a corresponding rewrite rule.

model waiting for adaptation events. If an adaptation event occurs, the monitor forwards this event to the adaptation engine, which evaluates the event to apply a sequence of rewrite rules to adapt the structure of the access control model and specialization tree.

6.7.1. Adaptive Specialization Tree Definition

Adaptive specialization trees are defined by means of an initial specialization tree definition, a corresponding access control model, and an adaptation strategy comprising a set of rewrite rules. An initial specialization tree definition specifies the initial part of the specialization tree, while further parts are dynamically generated by applying rewrite rules. A minimal adaptive specialization tree definition comprises the following nodes and edges,

- an *initial node* for defining the start of the workflow,
- a *fork* for separating the control flow between the termination path and the paths related to staged configuration prcoesses,
- a *control action* waiting for an external event for terminating the workflow,
- an *activity final node* specifies a global end of the workflow,
- transitions between the nodes, and
- all sequentially ordered *stages*.

Figure 6.21 shows a minimal adaptive specialization tree definition. In a specialization tree the order of stages is fixed, as well as their relation to abstract roles in the access control model.

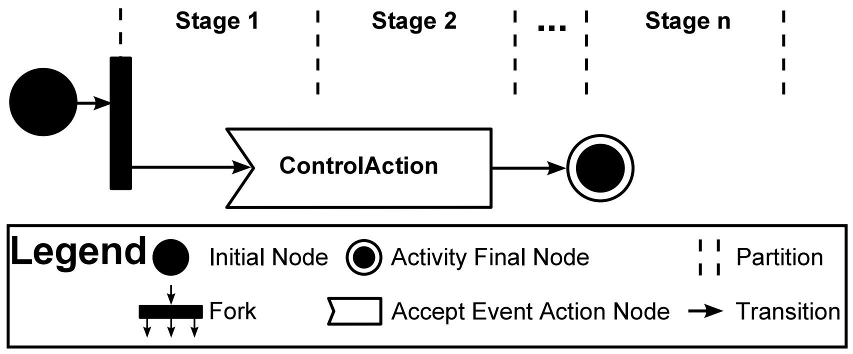


Figure 6.21 Initial specialization tree definition used for adaptation.

Hence, an initial access control model must specify abstract roles related to stages and their permissions, while concrete roles are added through graph transformations.

According to the well-formedness criteria of a specialization tree, described in Section 6.5.1, a minimal specialization tree is well-formed. The specialization tree does not yet explicitly represent staged configuration processes. Further specialization actions are integrated by applying rewrite rules as explained in the following.

Graph rewrite rules specify the transformation of an access control model to add and remove stakeholders and to adapt the specialization tree accordingly. Rewrite rules are defined on a set of nodes N and edges E of a graph $G = (N, E)$. A specialization tree and a referenced access control model has a graph structure comprising attributed nodes N and directed labeled edges E to interconnect nodes. Nodes refer to object instances of metaclasses containing attributes according to their metamodel specifications. In addition, edges corresponds to references between object instances. Rewrite rules define the addition and removal of object instances and references, as well as attribute value assignments in the specialization tree and a referenced access control model. In the next section, rewrite rules for integrating stakeholders are explained.

6.7.2. Rewrite Rules for Integrating Stakeholders

To integrate a new stakeholder, new objects are instantiated in the access control model and the specialization tree. In the access control model, concrete roles, their relation to other roles, groups and the relations to a group are added. In the specialization tree, new control and data flow structures and the new specialization action related to the new stakeholder are added. Forks represent adaptation points in the specialization tree.

First, a concrete role is instantiated in the access control model representing the new stakeholder. Further references are to be introduced modeling the relation of this role to other roles. A specialization action and related control and data flow structures are to be created in the specialization tree accordingly. Therefore, a name and id of the new stakeholder role are given, an abstract role representing a stage is selected, and a concrete role which is the owner of a role group. In addition, constraints occur on selectable input parameters. Hence, input parameters are specified in an explicit order as they depend on each other. The input order is as follows.

1. *Specifying stakeholder name and identifier.* The name and identifier of the new stakeholder must be defined to instantiate a new role with these values.

2. *Selecting an abstract role.* The selected abstract role corresponds to a stage where to integrate the new stakeholder. Prior to selecting an abstract role, the set of selectable abstract roles is filtered. Only abstract roles that are assigned to stages are available, where the assigned stage is either the first stage that has no predecessor or a stage where the predecessor stage comprises at least one specialization action.

3. *Selecting a concrete role.* The selected concrete role represents a depending configuration stakeholder which will be the direct predecessor in the configuration process. The previously selected abstract parent role implies selectable concrete roles. All concrete roles assigned to specialization actions of the predecessing stage represented by the chosen abstract role which are owner of a role group are available for selection. However, selecting a concrete role requires the existence of concrete roles in the predecessing stage. The selection of a concrete role is omitted, if the new role will be added to the first stage.

The filter constraints implicitly define which specialization actions and control flow structures can be added in the current structure of the specialization tree. For example, in the specialization tree of the document management system and depicted in Figure 6.30, only concrete roles inheriting from the abstract *Customer* role can be added. Concrete roles inheriting from the abstract *User* role can currently not be added, as this would lead to an unconnected specialization tree in the workflow model.

Rewrite rules to integrate a new stakeholder in the specialization tree are visualized in Figures 6.22 to 6.29. Figures depicting the graph rewrite rules apply a graphical notation of UML object diagrams to specify the LHS and RHS patterns of the access control and workflow models.

Input parameters of a rewrite rule are highlighted applying an italic font. In addition, a red cross in the search pattern in the LHS represents a non-existing reference. The dotted line on the left side frames the search pattern in the instance graph, while the dotted line on the right side frames the replacement pattern and is similar to the notation applied in *PROGRES* [RW08].

The following rewrite rules *A1* to *A8* are specified for stakeholder integration and applied as a sequence in the given order.

A1 Add a concrete role with given name and identifier to the access control model and insert an inheritance relation to the specified abstract parent role, as depicted in Figure 6.22.

A2 Define group membership relation between the new concrete role and a different specified concrete role, as depicted in Figure 6.23.

A3 Create specialization action and relation to concrete role, add the specialization action to the stage assigned to related abstract parent role, as depicted in Figure 6.24.

A4 Add predecessing control and data flow transitions, if new role assigned to the first stage, as depicted in Figure 6.25. The stage related to the specialization action of the new role does not have a predecessor relation to a different stage, which is visualized in the figure by a red cross on the predecessor reference.

A5 Add predecessing control and data flow transitions to the new specialization action by evaluating group membership, as depicted in Figure 6.26.

A6 If the specialization action is not added to the last stage, create a new group in the access control model and define a group ownership relation between the new concrete role and the new role group, as depicted in Figure 6.27.

A7 Add succeeding control and data flow transitions by evaluating group ownership, as depicted in Figure 6.28.

A8 If the specialization action is added to the last stage, insert a flow final node, as depicted in Figure 6.29. The stage related to the specialization action of the new role must not have a successor relation to a different stage, which is visualized in the figure by a red cross on the successor reference.

The adaptation engine applies the rewrite rules *A1* to *A8* as a sequence in the given order. The first rule returns a reference to the instantiated role, while this role provides input for further rules. The defined sequence ensures a connected specialization tree. Hence, the specialization tree grows and shrinks, but always remains connected.

The runtime state of an inserted specialization action depends on the runtime state of the predecessor specialization action. The runtime state is determined according to the action lifecycle described in Section 6.5.2.

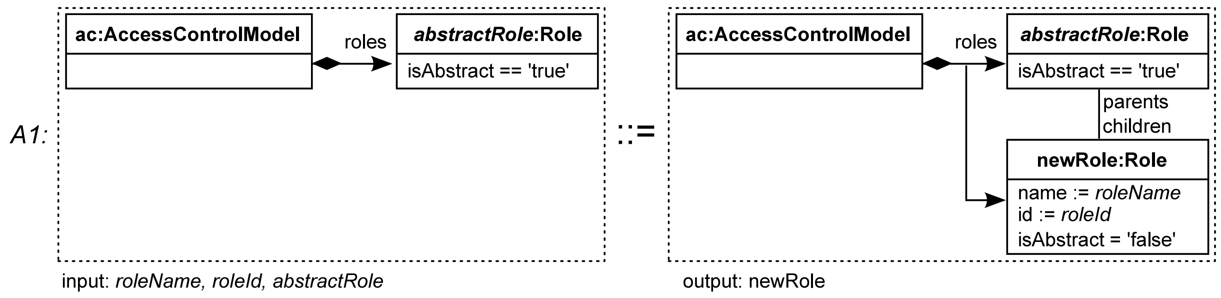


Figure 6.22 Rewrite rule for inserting a concrete role into the access control model.

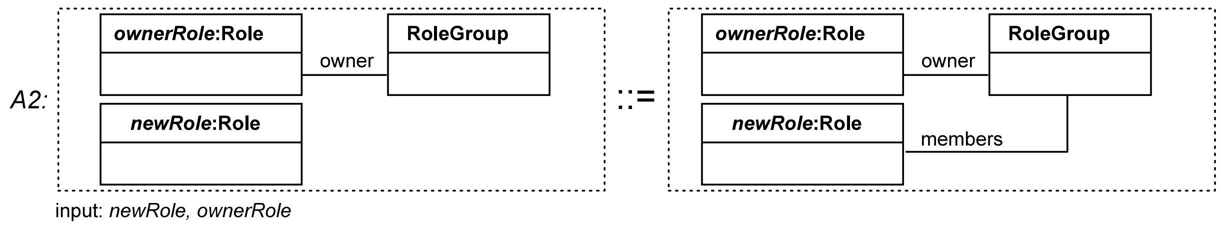


Figure 6.23 Rewrite rule for assigning a concrete role as a member to a role group in the access control model.

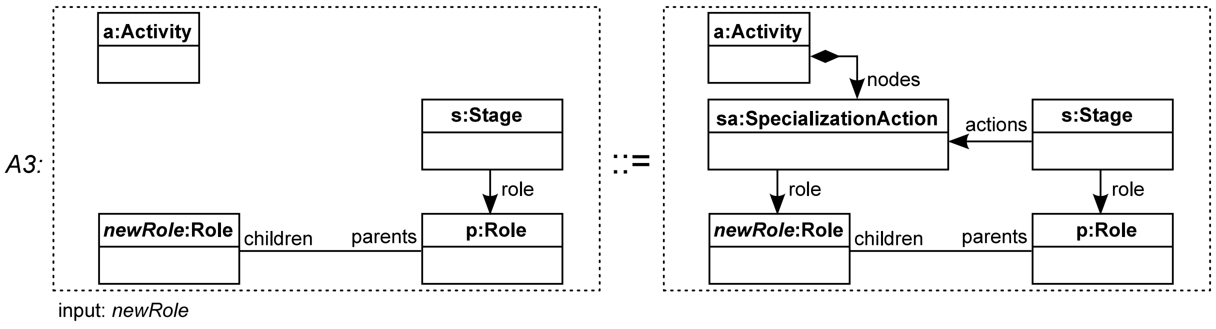


Figure 6.24 Rewrite rule for inserting a specialization action in the workflow model and instantiating required relations.

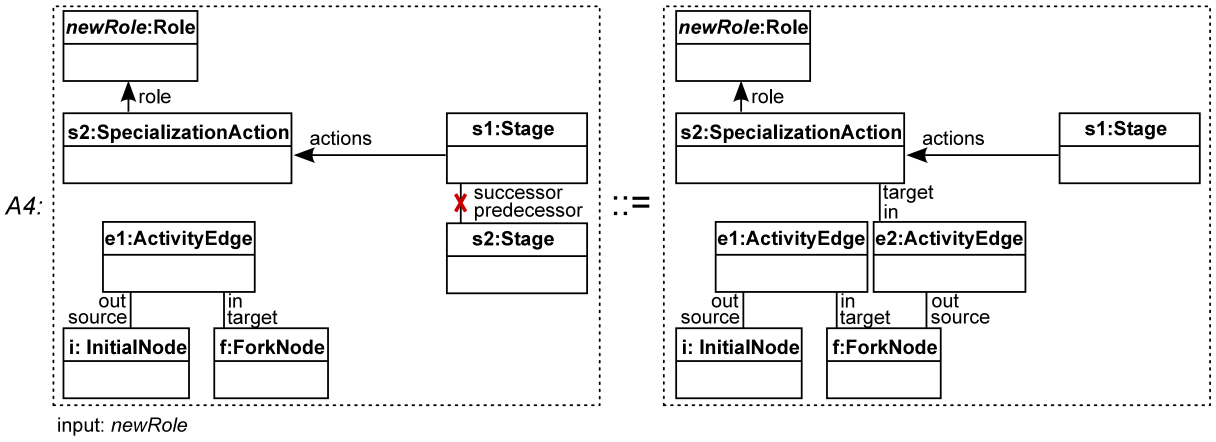


Figure 6.25 Rewrite rule for inserting a transition to a predecessing fork node in the workflow model if role belongs to first stage.

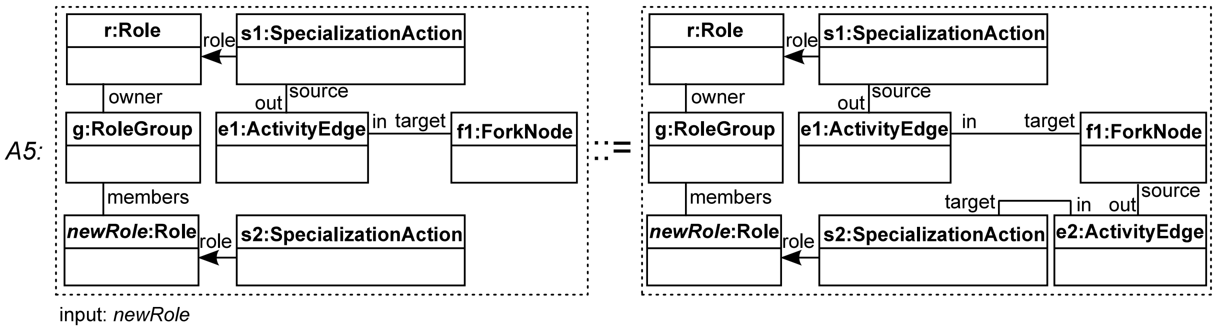


Figure 6.26 Rewrite rule for inserting a transition to a predecessing fork node in the workflow model if new role belongs to a role group in the access control model.

A6:

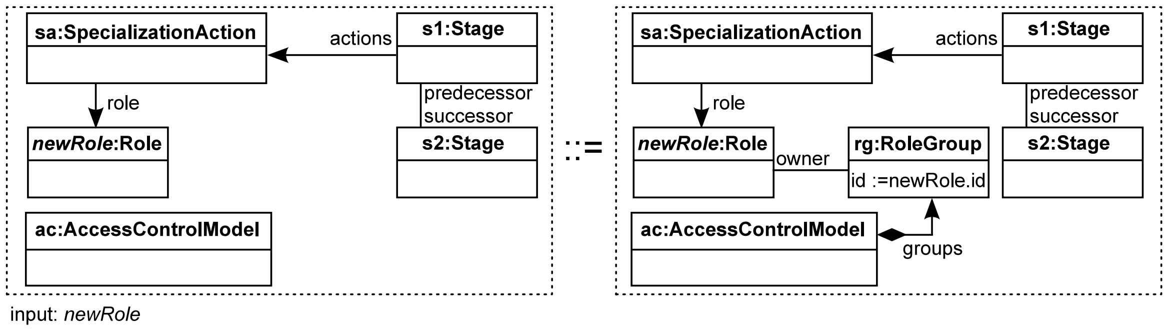


Figure 6.27 Rewrite rule for inserting a group and adding ownership to the new role in the access control model.

A7:

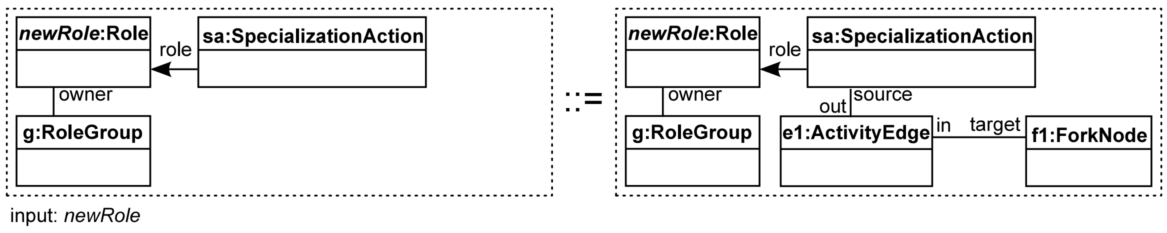


Figure 6.28 Rewrite rule for inserting a transition and a successor fork node in the workflow model if the role is owner of a role group.

A8:

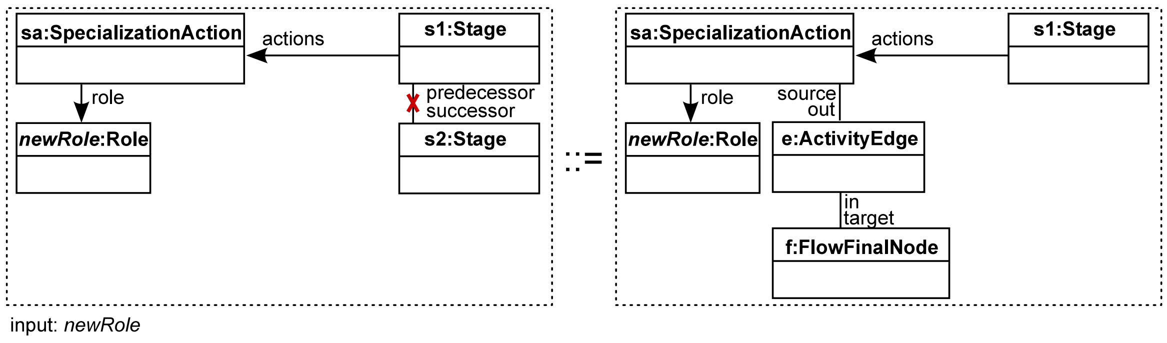


Figure 6.29 Rewrite rule for inserting transition and successor flow final node if role added to last stage in the workflow model.

6.7.3. Example Application of Rewrite Rules for Integrating Stakeholders

A specialization tree for the document management example, as visualized in Figure 6.15, can be generated from the initial specialization tree instance depicted in Figure 6.30. In this initial specialization tree, the order of stages and their relation to abstract roles is specified, as well as initial nodes, transitions and initially assigned concrete roles. In this example, the sequentially ordered specialization actions related to the concrete roles *Resource Provider* and *Application Provider* in the *Provider Stage* are specified. Hence, these concrete roles and all abstract roles assigned to stages are defined in the initial access control model, which is depicted in Listing 6.5. The initial access control model further specifies that the application provider is owner of a group representing the application and an entry point for forking the specialization tree.

Assuming that stakeholder *Customer A* subscribes for the service of the document management system, the configuration workflow is adapted. First, a stakeholder role is instantiated in the access control model and an inheritance relation is added to define that this role inherits from the

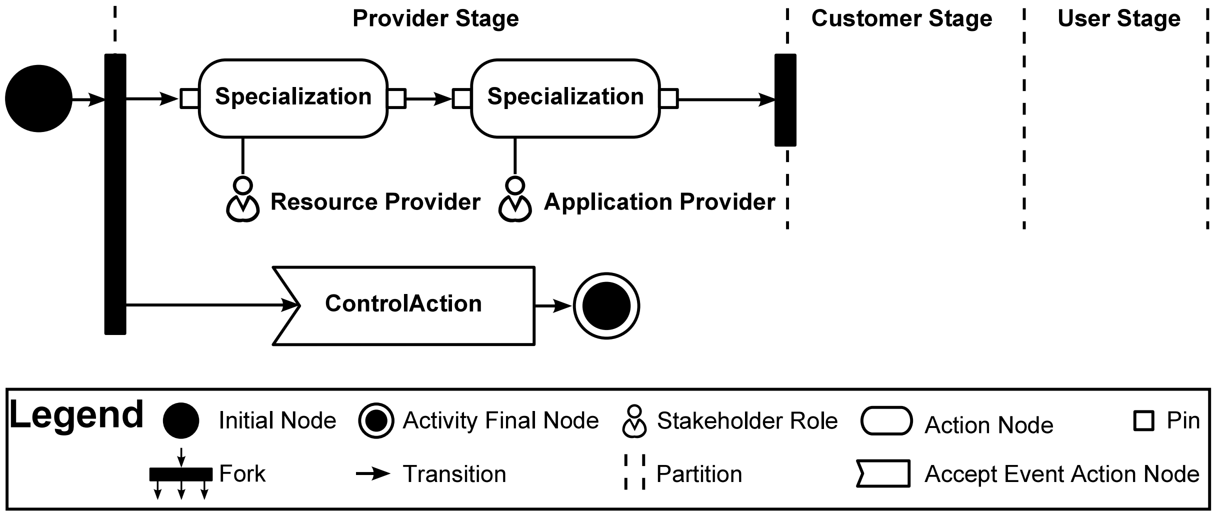


Figure 6.30 Initial specialization tree definition of the document management example used for adaptation.

```
access control on <extended_dms.eft>

abstract role "Functionality Configuration" <FunctionalityConfiguration> {...}

abstract role "Provider" <Provider>
abstract role "Customer" <Customer> extends FunctionalityConfiguration {...}
abstract role "User" <User> extends FunctionalityConfiguration

role "Resource Provider" <ResourceProvier> extends Provider {...}
role "Application Provider" <ApplicationProvider> extends Provider {...}

group <Application> of ApplicationProvider
```

Listing 6.5 Example of an initial adaptive access control model of the document management system.

abstract role *Customer* by applying the first rule *A1*. The role is added to a member relation with the role *Application Provider* by applying the second rule *A2*. A corresponding specialization action is instantiated in the workflow model and assigned to the *Customer Stage* by applying the rule *A3*.

The LHS of rule *A4* does not match, as the role belongs to the second stage in the specialization tree. However, rule *A5* is applied, as *Customer A* is a member of the role group of the *Application Provider*. Hence, a predecessing transition of the specialization action of *Customer A* to the fork is inserted. The fork splits the control and data flow after the specialization action of the *Application Provider*.

Subsequently, the application of rule *A6* instantiates a new role group for *Customer A* in the access control model. The role group represents configuration dependencies to other roles, which are instantiated during different applications of the rewrite rule sequence. Due to the instantiated

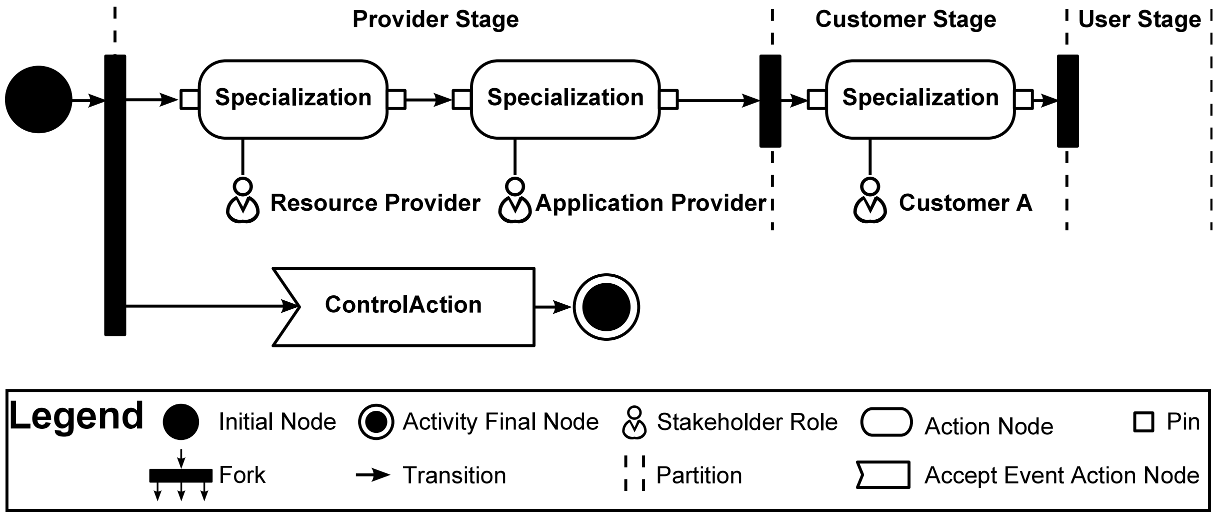


Figure 6.31 Resulting specialization tree after integrating stakeholder *Customer A*.

```
access control on <extended_dms.eft>

abstract role "Functionality Configuration" <FunctionalityConfiguration> {...}

abstract role "Provider" <Provider>
abstract role "Customer" <Customer> extends FunctionalityConfiguration {...}
abstract role "User" <User> extends FunctionalityConfiguration

role "Resource Provider" <ResourceProvier> extends Provider {...}
role "Application Provider" <ApplicationProvider> extends Provider {...}
role "Customer A" <CustomerA> extends Customer

group <Application> of ApplicationProvider
group <CustomerA> of CustomerA
```

Listing 6.6 Access control model after integrating stakeholder *Customer A*.

role group, rule *A7* is applied to add further control structures to the workflow model. The last rule *A8* of this sequence does not transform any model as the LHS of this rule is not matched.

The resulting specialization tree is shown in Figure 6.31, while the resulting access control model is depicted in Listing 6.6. The group relations between roles in the access control model is translated into a forking in the specialization tree. The role group with the owner role *Application Provider* defined in line 13 in Listing 6.6 has currently one member role *Customer A*. This role group is represented in the specialization tree by a fork structure, where the specialization action related to the owner *Application Provider* is predecessor of all specialization actions of member roles.

A fork node is inserted in the specialization tree between the specialization actions of the *Application Provider* and *Customer A*, as depicted in Figure 6.31. Fork nodes in the specialization tree are adaptation points. For instance, if a second stakeholder role *Customer B* is instantiated as a member of this role group, the corresponding specialization action of this stakeholder is integrated as a successor of the fork node of the specialization action related to role *Application Provider*.

Applying the sequence of rewrite rules *A1* to *A8* to integrate a stakeholder does not interfere with executing specialization actions in the specialization tree. Hence, the application of integration sequences is non-destructive. In contrast, applying the sequence of graph rewrite rules to remove a stakeholder interferes with existing specialization actions. The application of this sequence causes depending specialization actions to terminate instantly, and to be removed accordingly, which is referred to as *termination by subtraction* [Ass00]. The rewrite rules for removing a stakeholder are explained in the next section.

6.7.4. Rules for Removing Stakeholders

To remove a stakeholder from the staged configuration workflow, the role assigned to this stakeholder has to be removed, as well as the specialization action in the specialization tree. The specialization action assigned to the removable role in the specialization tree is observed. All transitions and nodes in the specialization tree succeeding this specialization action by following the directed transitions are to be removed. In addition, the reference to the role is removed from the specialization action, as well as all logs assigned to this action. In the access control model, all relations to groups and parent roles are to be removed. If the role is owner of a role group, this role group is deleted accordingly.

Rewrite rules for removing stakeholders roughly correspond to the rewrite rules for integrating stakeholders but with interchanging LHS and RHS of each rule and applying them in reversed order. The following rules *R1* to *R10* for removing a stakeholder from the staged configuration workflow are specified, and applied as a sequence in the given order. Input for each graph rewrite rule is the role to be removed.

R1 Instantly terminate the execution of the specialization action referring to the removable role by setting the state of the action to *disabled*, as depicted in Figure 6.32.

R2 Logs assigned to the specialization action are to be removed, as depicted in Figure 6.33.

R3 If the specialization action of the removable role is contained in the last stage, the flow final node and the related transition are removed, as depicted in Figure 6.34

R4 Succeeding control and data flow transitions are removed by evaluating group ownership, as depicted in Figure 6.35.

R5 If the specialization action assigned to the role is not contained in the last stage, the role group related to this role is removed from the access control model. In addition, the group ownership relation between the removable concrete role and the role group is removed together with the role group. This rule is depicted in Figure 6.36.

R6 Remove predecessing control and data flow transitions from the specialization action by evaluating group membership, as depicted in Figure 6.37.

R7 Remove predecessing control and data flow transitions, if the removable role belongs to the first stage, as depicted in Figure 6.38.

R8 Delete specialization action and relation to removable concrete role, and remove the specialization action from the stage assigned to related abstract parent role, as depicted in Figure 6.39.

R9 Remove the group membership relation between the removable concrete role and an owner concrete role of this role group, as depicted in Figure 6.40.

R10 Remove the inheritance relation to the specified abstract parent role and remove concrete role from the access control model, as depicted in Figure 6.41.

The defined rewrite rules *R1* to *R10* are applied in the specified order on a stakeholder role to remove this stakeholder and associated objects from the specialization tree and access control model. However, removing a stakeholder requires the removal of depending stakeholders. A stakeholder s_2 depends on another stakeholder s_1 if the related roles are assigned to specialization actions comprised in the same directed path and the specialization action of s_2 is a descendant of the specialization action of s_1.

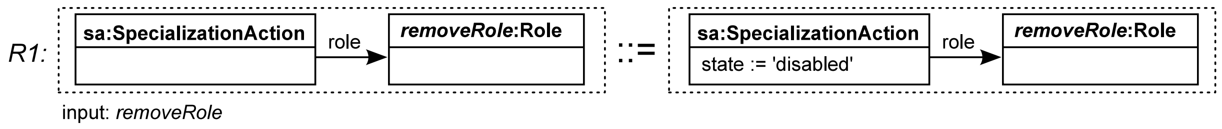


Figure 6.32 Rewrite rule for disabling an action in the workflow model.

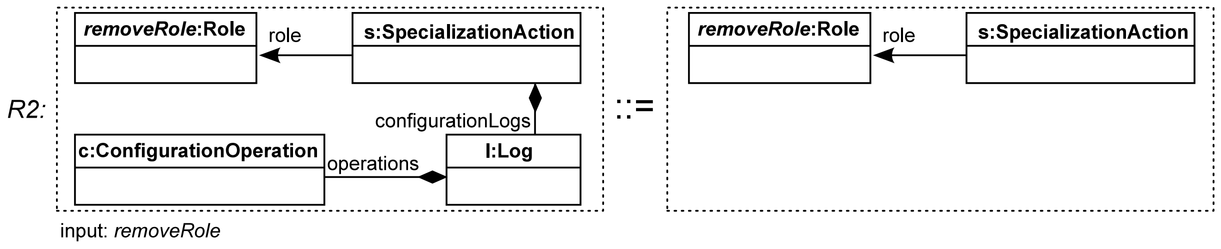


Figure 6.33 Rewrite rule for removing logs indirectly related to a removable role.

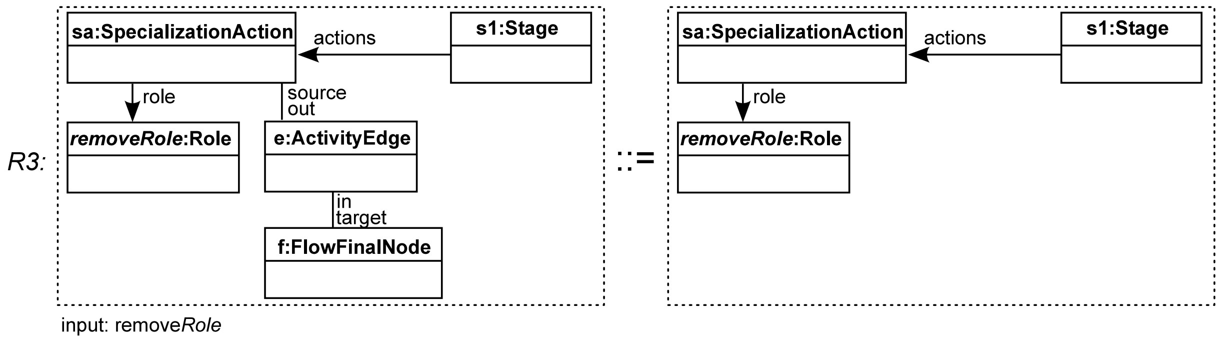


Figure 6.34 Rewrite rule to remove the flow final node and transitions in the last stage.

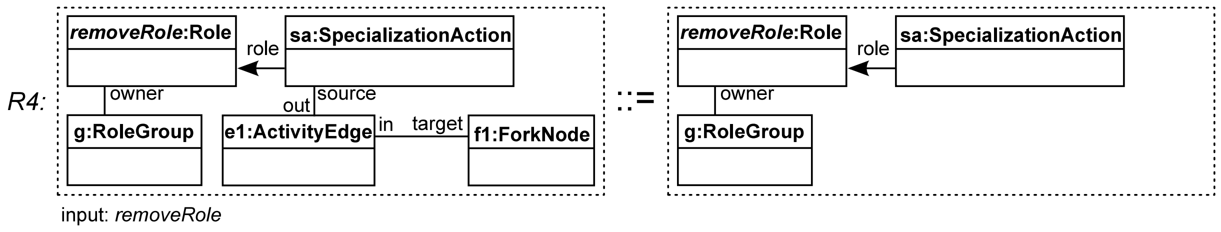


Figure 6.35 Rewrite rule for removing succeeding control and data flow transitions.

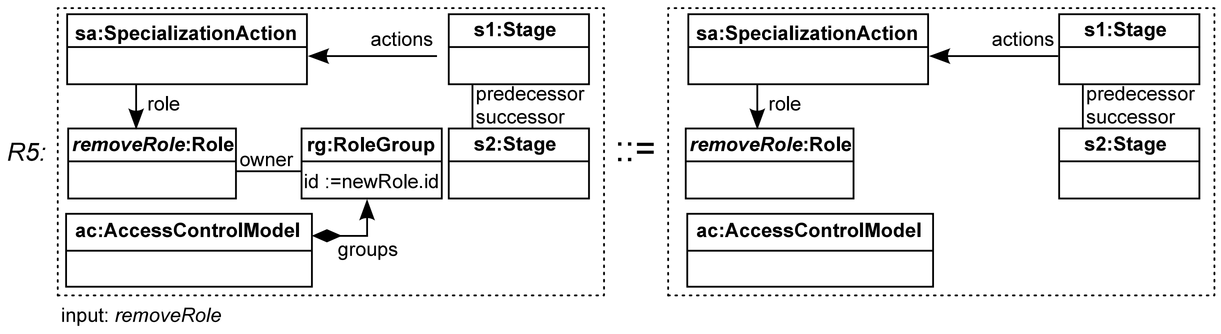


Figure 6.36 Rewrite rule for removing the role group where the removable role is owner.

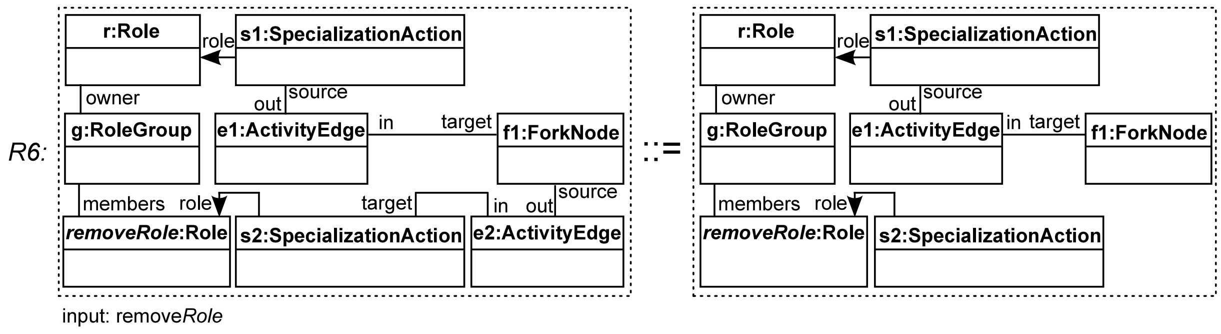


Figure 6.37 Rewrite rule for removing predecessing control and data flow transitions by evaluating role group membership.

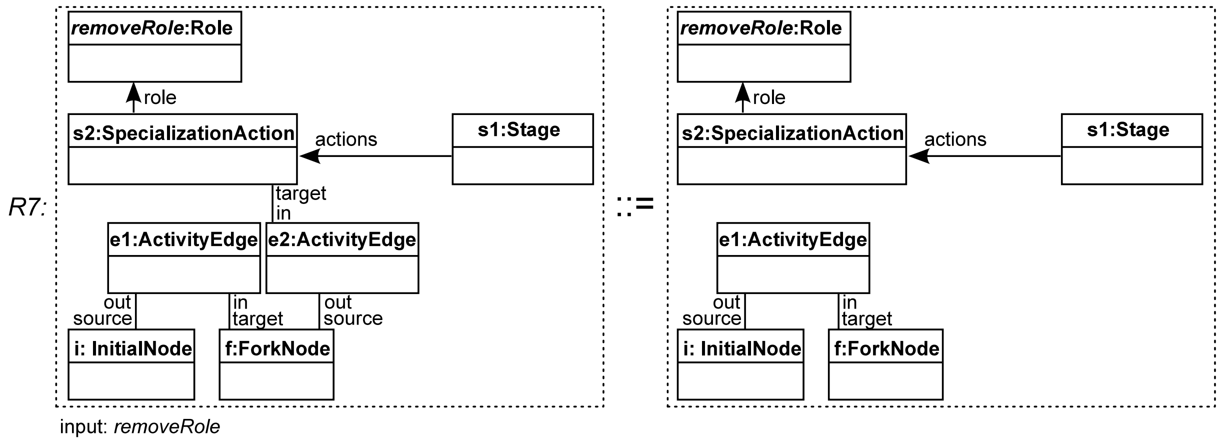


Figure 6.38 Rewrite rule for removing predecessing control and data flow transitions in the first stage.

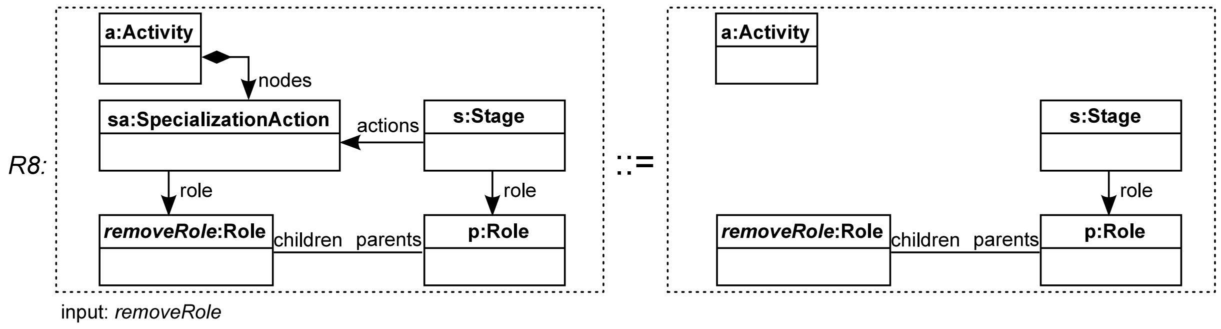


Figure 6.39 Rewrite rule for removing specialization action and relation to removable concrete role.

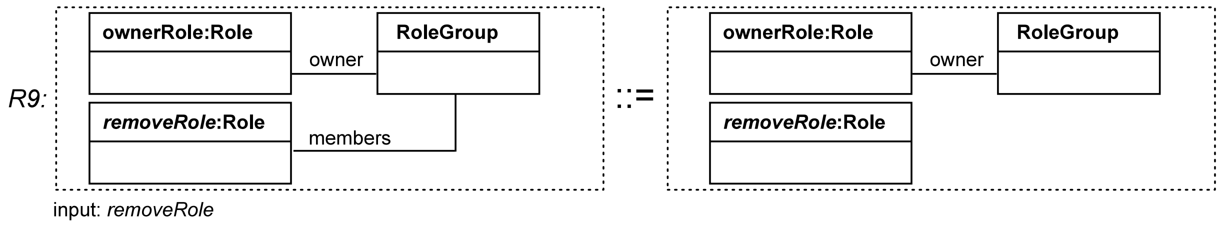


Figure 6.40 Rewrite rule for removing the group membership relation.

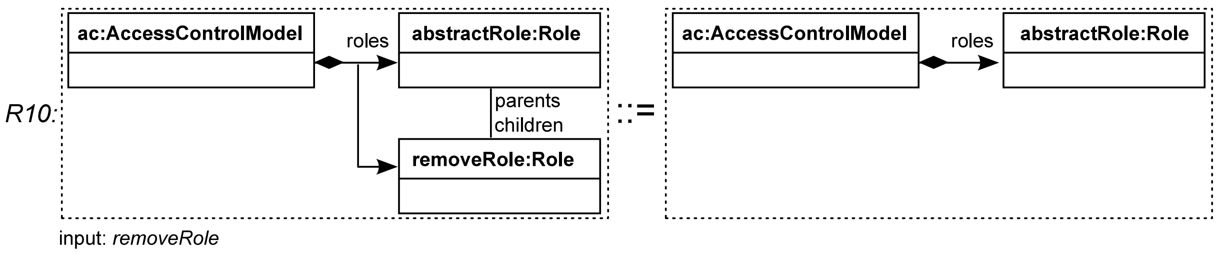


Figure 6.41 Rewrite rule for removing inheritance relation to the parent role and delete the role.

Algorithm 3 Algorithm to recursively remove roles *RemoveRoleRecursively(role r)*

```
1: Input:  role r
2: a := getSpecializationAction(r)
3: SA := getSuccessorActions(a)
4: for all $s \in$ SA do
5:     $r_s$ = getRole(s)
6:     removeRoleRecursively($r_s$)
7: end for
8: applySequenceRemoveRewriteRules(r)
```

The adaptation engine applies Algorithm 3 to recursively remove roles. The algorithm applies the sequence of graph rewrite rules *R1* to *R10* on all depending stakeholders starting with roles assigned to most recent specialization actions in the specialization tree.

A depth-first search on the specialization tree is applied to determine depending stakeholder roles and remove them. Hence, the algorithm takes advantage of the staged configuration workflow's underlying tree-structure. In addition, the sequence of rewrite rules *R1* to *R10* is applied on each depending role in the same order.

An example of how to apply the algorithm and the defined rewrite rules to recursively remove a stakeholder is explained in the following section.

6.7.5. Example Application of Rewrite Rules for Removing Stakeholders

Further stakeholders are integrated into the workflow model depicted in Figure 6.31, and related access control model presented in Listing 6.6. The stakeholders *User A1* and *User A2* represent users of *Customer A* and are therefore inserted into the *User Stage*. A further stakeholder *Customer B* is added to the *Customer Stage*. Figure 6.42 and Listing 6.7 represent the related workflow model and access control model instances.

In this example, different stakeholders already finished their configuration tasks depicted by corresponding annotations in the action nodes, while stakeholder *User A1* is currently configuring. The removal of *Customer A* implies the removal of depending stakeholders *User A1* and *User*

A2 while the stakeholder *Customer B* is not affected. Hence, the request to remove *Customer A* triggers Algorithm 3.

The algorithm determines succeeding stakeholder roles, which are *User A1* and *User A2* in this case. The algorithm is recursively applied on both succeeding roles. The application of the algorithm on role *User A1* first reveals that this role does not have successor roles. Hence, the recursion stops and the sequence of rewrite rules is applied on role *User A1*.

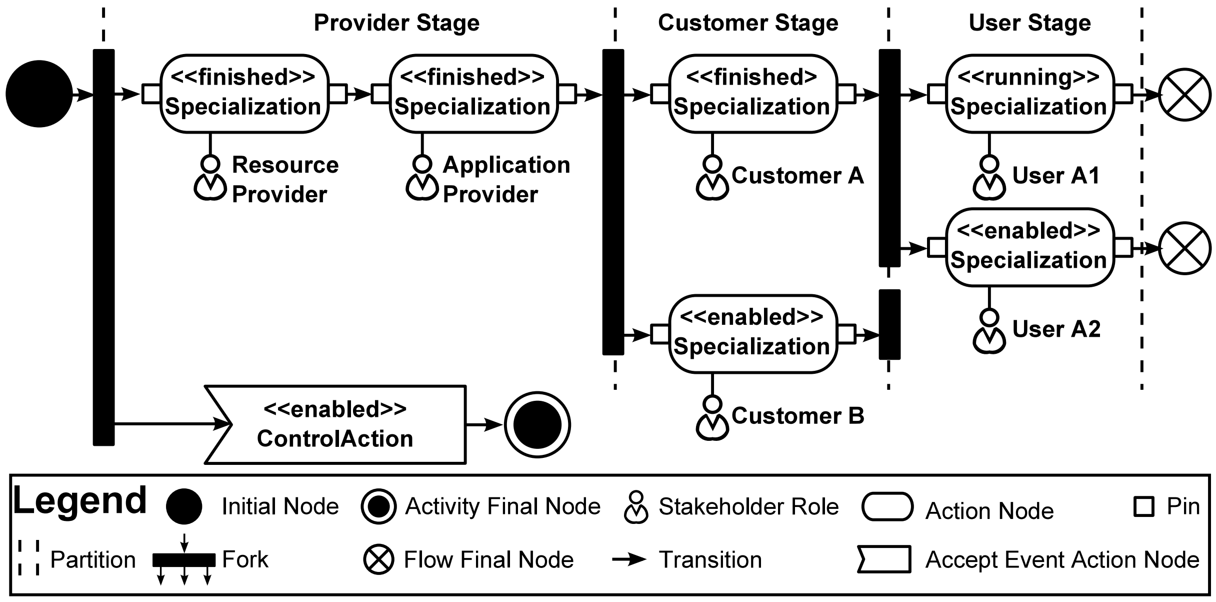


Figure 6.42 Specialization tree of the document management example before removing stakeholder *Customer A*.

```
access control on <extended_dms.eft>

abstract role "Functionality Configuration" <FunctionalityConfiguration> {...}

abstract role "Provider" <Provider>
abstract role "Customer" <Customer> extends FunctionalityConfiguration {...}
abstract role "User" <User> extends FunctionalityConfiguration

role "Resource Provider" <ResourceProvier> extends Provider {...}
role "Application Provider" <ApplicationProvider> extends Provider {...}
role "Customer A" <CustomerA> extends Customer
role "Customer B" <CustomerB> extends Customer
role "User A1" <UserA1> extends User
role "User A2" <UserA2> extends User

group <Application> of ApplicationProvider
group <CustomerA> of CustomerA has members UserA1, UserA2
group <CustomerB> of CustomerB
```

Listing 6.7 Access control model of the document management example before removing stakeholder *Customer A*.

Applying the first rule *R1* on this role terminates the configuration task of this role by changing the state of the specialization action back to *disabled*. Hence, currently conducted configuration operations in this task are reverted. The second rule *R2* deletes potential logs of the role's specialization action.

The application of rule *R3* on role *User A1* removes outgoing edges and the flow final node from the related specialization action in the workflow model. The application of rules *R4* and *R5* does not affect the models as the role is not owner of a role group and is related to the last stage. Subsequently, rule *R6* is applied to remove predecessing transitions, while rule *R7* does not remove predecessing transitions as the role *User A1* does not belong to the first stage.

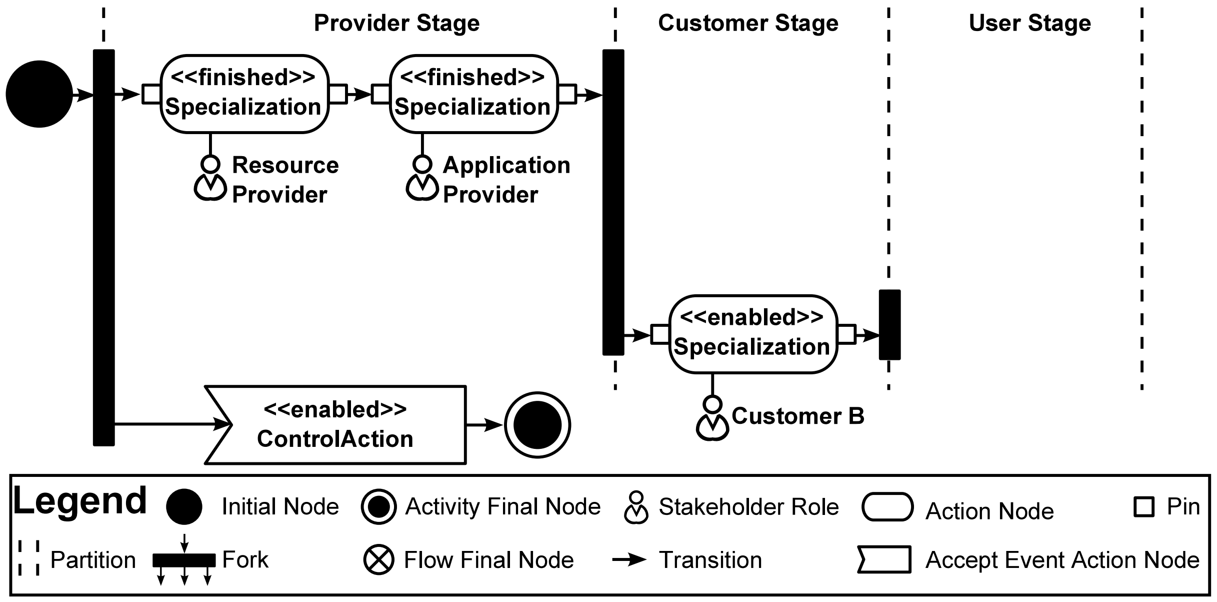


Figure 6.43 Specialization tree of the document management example after removing stakeholder *Customer A*.

```
access control on <extended_dms.eft>

abstract role "Functionality Configuration" <FunctionalityConfiguration> {...}

abstract role "Provider" <Provider>
abstract role "Customer" <Customer> extends FunctionalityConfiguration {...}
abstract role "User" <User> extends FunctionalityConfiguration

role "Resource Provider" <ResourceProvier> extends Provider {...}
role "Application Provider" <ApplicationProvider> extends Provider {...}
role "Customer B" <CustomerB> extends Customer

group <Application> of ApplicationProvider
group <CustomerB> of CustomerB
```

Listing 6.8 Access control model of the document management example after removing stakeholder *Customer A*.

The application of rule *R8* removes the specialization action assigned to this role from the workflow. Rule *R9* removes the role *User A1* from the role group of *Customer A*. Eventually, rule *R10* removes the inheritance relation of the role *User A1* from the abstract role *User* and deletes the role *User A1*. After finishing the sequence of rewrite rules, Algorithm 3 applies the sequence on role *User A2* and finally on role *Customer A*.

After removing role *Customer A*, the algorithm terminates. The resulting workflow model is depicted in Figure 6.43 and the resulting access control model after applying the rewrite rules is shown in Listing 6.8.

Only stakeholders *Application Provider*, *Resource Provider*, and *Customer B* remain, while the stakeholders *Customer A*, *User A1*, and *User A2* are removed. The stakeholder *Customer B* is not affected from removing stakeholder *Customer A*, as their control and data flow are split by a fork node.

6.8. Adaptive Staged Reconfiguration Workflows in Software Product Line Engineering

The processes of domain and application engineering are distinguished in SPL engineering, as explained in Section 2.2. Adaptive staged reconfiguration workflows are integrated in both. During domain engineering, an initial specialization tree definition is defined according to Section 6.7.1. Furthermore, abstract roles and related permissions are defined in a corresponding access control model referencing a feature model. Abstract roles from the access control model are related to stages in the specialization tree definition.

During application engineering the specialization tree is executed as a workflow instance to derive complete variant configurations of the referenced feature model by configuring and reconfiguring partial configurations. However, the workflow does not terminate after deriving a single complete configuration. The workflow only terminates if the control action is explicitly invoked by an external termination event. Furthermore, the specialization tree and the according access control model are adapted to integrate or remove stakeholders during workflow execution.

6.9. Demarcation from Related Work

In this section, related work in the areas of configuration of cloud applications, staged configuration workflows, evaluation of partial feature model configurations, and reconfiguration is discussed.

Lytra and colleagues present a case study about the decision-making process of service-based platforms [LSZ12]. Configuration decisions are clustered into architecture, platform, integration, and application level, where each level comprises various refinement stages. The focus of this work is on extracting decision patterns of these levels and modeling their configuration dependencies in a pattern language. A decision-making process proposed in this work is inspired by staged configuration comprising a further dimension. However, configuration decisions are expressed in a decision model, and the result of the decision-making process is a single complete configuration. Reuse of pre-configurations on particular levels and reconfiguration is out of scope of the decision-making approach.

An approach for applying staged configuration to configure multiple heterogenous SPLs is proposed in [Els12]. The author extends staged configuration by introducing build steps to persist configuration artifacts during the configuration process. Thus, configuration artifacts are already created without finalizing the configuration process. Moreover, the approach allows for defining further constraints in each configuration step, which are evaluated in subsequent steps in conjunction with feature model constraints. Reconfiguration and adaptation of a workflow instance are not considered in this work.

Other authors introduce a special variability representation based on feature models and apply it in a cloud service selection process [WKM12]. The authors propose *cloud feature models* as a representation form to define selection criteria in cloud services. However, these feature models are attributed and enriched with abstract features, instance features, and further model elements, to name but a few. During a cloud selection process, a domain model is extended by adding further instance features representing instantiations of the abstract features. Furthermore, a requirements model is defined to make stakeholder requirements explicit by means of a selection of abstract and instance features, and a restriction of attribute domain values. Applying a requirements model on a service model yields a variant configuration referred to alternative model. The proposed approach applies a similar representation of variability compared to the feature model definition provided in this work. However, the way of deriving a variant configuration differs. In this approach, a stakeholder can define a requirements model that is matched against a service model. Furthermore, the process does not support feature model specialization of multiple stakeholders by means of staged configuration processes and is not intended for reconfiguration of partial configurations.

Krueger proposes multistage configuration trees to define pre-configurations for certain domains [Kru13]. This approach is similar to the definition of specialization trees in this thesis. In this approach, feature models are partially configured for certain application domains. The dependencies between partial configurations are expressed as configuration tree with an arbitrary depth. A configuration tree is a structural model, whereas a specialization tree is a behavioral model. Reconfiguration and configuration processes are not defined for configuration trees.

Lee and Kang describe an approach for developing dynamically reconfigurable core assets in an SPL [LK06]. A feature binding unit comprises features that are related to each other in a feature model. Hence, features in a binding unit are to be reconfigured together. In addition, a conceptual model of a reconfigurator is provided to monitor and manage product reconfiguration at runtime. Strategies for how to perform a reconfiguration in the solution space are provided according to the relations in a feature binding unit. Furthermore, binding units have different binding times, such as design time, runtime, and deploy time. These binding times are analyzed to define reconfiguration strategies for when, how, and which parts of the application should be reconfigured. The scope of this approach is on how to concisely specify and apply reconfiguration in the solution space. Hence, this approach can be combined with reconfiguration in staged configuration workflows in the problem space, as proposed in this thesis.

6.10. Summary

The adaptive reconfiguration approach proposed in this chapter is based on workflows represented as UML activity diagrams and incorporates RBAC to authorize configuration decisions on the feature model as RBAC is a common standard in cloud applications, as explained in Section 1.7.1. The presented approach allows for defining and executing structured multiple staged reconfiguration processes by means of a single workflow. Staged configuration concepts are extended to define specialization trees for modeling configuration dependencies between stakeholders and reusing partial configurations of higher prioritized stakeholders. Defining a staged configuration workflow as a specialization tree omits configuration redundancies of staged configuration processes with the same structure.

The reuse of partial configurations in a specialization tree allows for deriving multiple complete variant configurations in a single staged configuration workflow. In addition, a specialization tree allows for propagating reconfiguration changes along directed paths. Hence, the impact of a reconfiguration change is made explicit in the activity diagram representation. Reconfiguration changes in a specialization tree have global influence, if induced by high prioritized stakeholders. In contrast, reconfiguration changes have local influence, if induced by stakeholders with a low priority in the staged configuration workflow.

An adaptation engine is introduced in this chapter for dynamic stakeholder management. The adaptation engine applies a sequence of rewrite rules on a specialization tree, and referenced access control model, to integrate or remove a stakeholder from the staged configuration workflow. The adaptation is applicable during executing the staged configuration workflow. The concepts of specialization trees, reconfiguration and adaptation are exemplified in this chapter on a reconfigurable document management system offered as a cloud application.

Summarizing, the proposed concept of adaptive staged reconfiguration workflows is a generic foundation for configuration processes, where different stakeholder are involved that cannot be modeled before executing the configuration process. In addition, a specialization tree reduces configuration redundancies by reusing partial configurations and propagating reconfiguration changes following the tree structure.

7. Configuration Management Tool Suite PUMA

Before software can be reusable it first has to be usable.

— *Ralph Johnson*

Configuration management concepts related to the problem space and presented in the chapters before, are implemented in the tool suite Product Line Utilities for Multi-Tenant Aware Applications (PUMA).[1] The tool suite provides techniques to model and analyze extended feature models, to define and evaluate perspectives, and model and execute adaptive staged configuration processes comprising access restrictions on feature model configuration operations. Figure 7.1 gives an overview of the tools implemented in Java and integrated as plug-ins in the Eclipse[2] Integrated Development Environment (IDE). The tool suite follows a model-view-controller approach to enable reuse of tool components and interchange of models [GHJV95]. Additionally, a layered software architecture enables SoC and reuse and supports evolving functionality, system scalability and potential new technologies can be integrated. The tool suite comprises the tools *Conper*, *DyscoGraph*, and *FMAnalysis*.

Conper implements the concepts of views and perspectives presented in Chapter 5, and DyscoGraph implements adaptive staged configuration workflows, as introduced in Chapter 6. In addition, FMAnalysis is a tool for analyzing extended feature models utilized by Conper and DyscoGraph. The tools are integrated and accessible via the Eclipse platform and can be used independently. Interoperability between the tools is given by operating on the same feature model structure, sharing feature model information, and by providing public interfaces.

The core functionality of the PUMA tool suite is to define, modify, analyze and execute concepts of multi-perspectives and adaptive staged configuration. Hence, PUMA implements configuration management concepts related to the problem space. However, the concepts of a flexible application architecture explained in Chapter 4 are not implemented in PUMA and therefore not evaluated in this chapter.

The tool suite is implemented as plug-ins to realize low-coupled modules that are easier to maintain. Hence, each tool comprises several plug-ins to separate model, view and controller concepts. The application logic on *Layer 2* is separated from the presentation on *Layer 3* to enable reuse. On *Layer 3* Eclipse offers interfaces to define views and editors integrated into the Eclipse User Interface (UI) for visualizing and manipulating data. Hence, these multi-perspective concepts and adaptive staged configuration concepts are visualized on this layer. Furthermore, editors to model and manipulate these concepts are provided implementing Eclipse interfaces. Additionally, on *Layer 1* an internal tool repository comprises structural models representing

[1] `https://github.com/extFM/extFM-Tooling`
[2] `http://www.eclipse.org/`

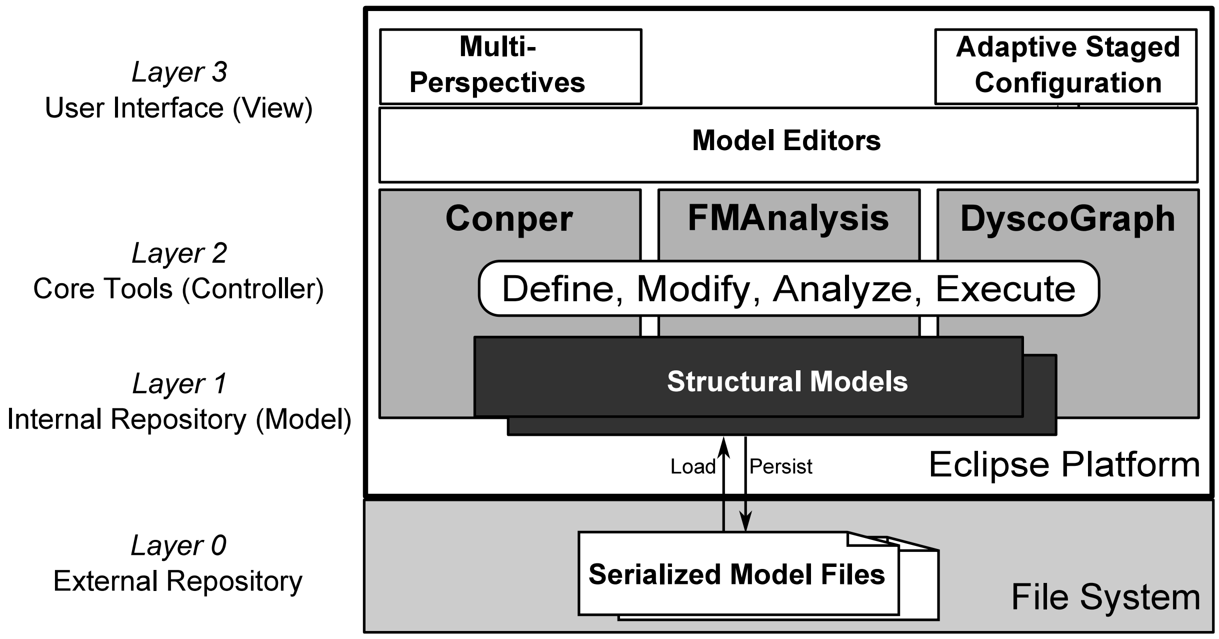


Figure 7.1 The PUMA tool suite comprises the tools *Conper*, *FMAnalysis*, and *DyscoGraph* integrated in Eclipse.

data to specify feature models, multi-perspectives, and adaptive staged configuration. Structural models are loaded into the internal model repository by parsing serialized files on *Layer 0*. In addition, technical services such as loading and persisting model files are separated from application logic to enable reuse and to potentially replace the underlying repository. The physical representation of the models in the file system on *Layer 0* is separated from the logical representation in the model repository [HL93].

This chapter presents tools and languages comprised in the configuration management suite PUMA and shows empirical evaluation results.

7.1. Model Based Domain-Specific Languages

PUMA applies the frameworks Eclipse Modeling Framework (EMF)[3] and EMFText[4] to uniformly model the concepts of multi-perspectives and adaptive staged configurations. The specific domain concepts are precisely defined in a model-based Domain-Specific Language (DSL). A DSL is a formal language that is tailored to a certain domain [Nei84]. In contrast to a General Purpose Language (GPL), its expressiveness is reduced. Such languages encapsulate domain knowledge and enable domain experts to understand and express concepts, as well as constraints in a concise and self-documenting way. Additionally, DSLs enhance reliability and portability of the modeled information on a higher level of abstraction than source code [VDKV00]. A DSL consists of a syntactical and a semantical domain and a mapping between both [HR04, SK95]. A syntactical

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

domain defines the structure and essential tokens of the language, whereas a semantic domain specifies the meaning of the concepts. The syntactical domain comprises an abstract syntax represented by a metamodel following the MDSD and a concrete syntax defined by a grammar. In the following, the focus is on the definition of metamodels and the application of models in PUMA to concisely specify configuration management concepts presented in the previous chapters.

Model-Driven Architecture (MDA) is a standard for MDSD defined by Object Management Group (OMG) for developing software based on models [Obj03]. The MDA proposes the metalanguage Meta Object Facility (MOF)[5] for specifying models. The MOF metalanguage is applicable to define DSLs. A DSL comprises an *abstract syntax* describing the structure of the language and a *concrete syntax* defining a language notation, either graphical or textual. Essential Meta Object Facility (EMOF) is a subset of MOF narrowed down to contain essential concepts for object-oriented modeling by means of concepts known from UML class diagrams [OMG2011b]. A reference implementation and a de-facto standard of EMOF is Ecore [KK02]. Ecore is a part of EMF and integrated in the technological space of Eclipse and Java [SBPM09]. EMF provides a graphical modeling notation similar to the notation of UML class diagrams to define the abstract syntax of a DSL. Well known concepts, such as packages, classes, operations, references, and attributes, are applied to specify a metamodel in EMF. Hence, the EMF framework provides convenient tooling to define various model-based DSLs combined with a comprehensive generation of methods to instantiate models.

The tool suite PUMA comprises different EMOF-based models for defining configuration management concepts, as depicted in Figure 7.2. All tools of the tool suite rely on *feature models* as central models. The feature models applied in the tool suite conform to the metamodel definition explained in Section 6.3 and depicted in Figure 6.3. Instances of the metamodel represent unconfigured feature models, partial configurations, or complete configurations. Modeling feature model configurations similar to unconfigured feature models allows for a unified processing in the PUMA tool suite. The tool *FMAnalysis* provides logic-based analysis methods of extended feature models. For instance, FMAnalysis checks satisfiability of a given feature model. The tool *Conper* enhances feature models with a definition of views in a *view model* hierarchically structuring concerns. In addition, Conper defines a mapping between features in a feature model and viewgroups in a view model in a *feature-view mapping model.* The tool *DyscoGraph* defines an *access control model* that references a feature model. The access control model applies RBAC concepts to restrict feature model configuration operations to specific roles. Furthermore, DyscoGraph defines a *workflow model* that implements staged configuration concepts by defining serialized specialization actions to configure feature models. The *action-role mapping model* defines a mapping between workflow action nodes and roles in the access control model to assign configuration operations to actions. In addition, this tool comprises a *stage model* to define a sequence of stages and to assign specialization actions to these stages. These models comply to the structure defined by corresponding EMOF metamodels.

[5] `http://www.omg.org/mof/`

7.1.1. Metalevel Hierarchy of Tool Suite Languages

The MDSD specifies a metalevel hierarchy of model-based DSLs. In the sense of MDSD, the abstract syntax of a language is defined in EMF by a metamodel, as explained before. Based on this metamodel potentially many concrete syntaxes can be defined. Applying the frameworks EMF and EMFText seize this idea and allows for specifying language instances in both graphical and textual notation, and automatically convert the notations into each other.

Figure 7.3 illustrates the metalevels of the DSLs contained in PUMA and defined in EMF and EMFText. EMOF is applied on Level *M3* as a metametamodel by means of the Ecore reference implementation to define the structure of metamodels. On level *M2*, an instance of EMOF defines a metamodel of the language. Subsequently, level *M1* represents a model instance following the structure defined by the metamodel on level *M2*. However, to visualize a model on level *M1*, a

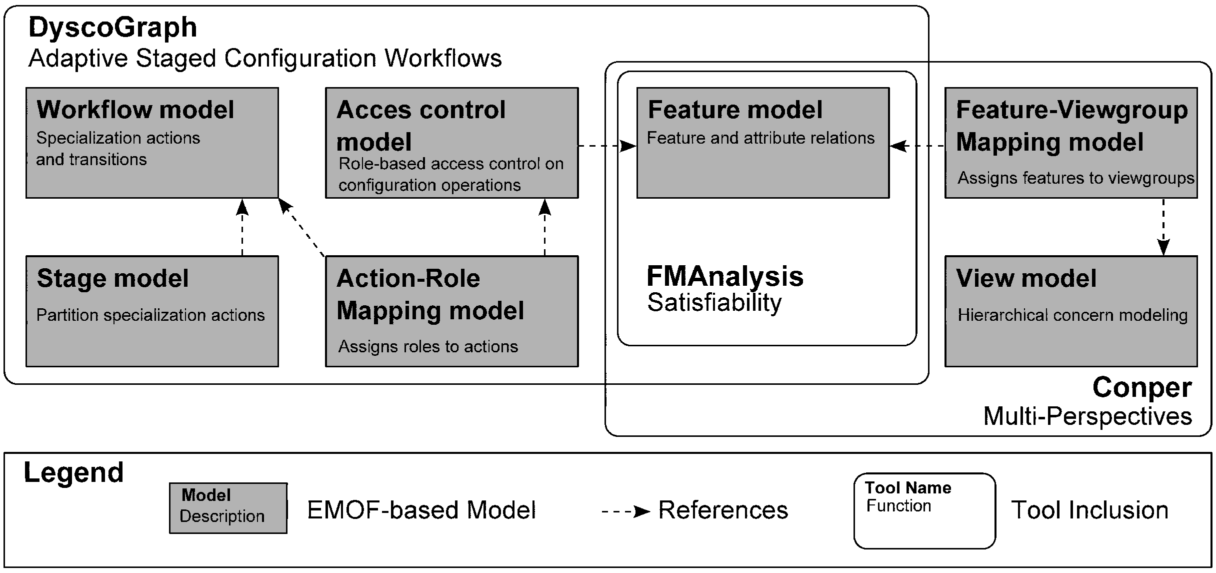

DyscoGraph
Adaptive Staged Configuration Workflows
Workflow model
Specialization actions and transitions
Acces control model
Role-based access control on configuration operations
Feature model
Feature and attribute relations
Feature-Viewgroup Mapping model
Assigns features to viewgroups
Stage model
Partition specialization actions
Action-Role Mapping model
Assigns roles to actions
FMAnalysis
Satisfiability
View model
Hierarchical concern modeling
Conper
Multi-Perspectives
Legend
Model
Description
EMOF-based Model
References
Tool Name
Function
Tool Inclusion

Figure 7.2 The tool suite PUMA comprises various EMOF-based domain-specific languages to model configuration management concepts.

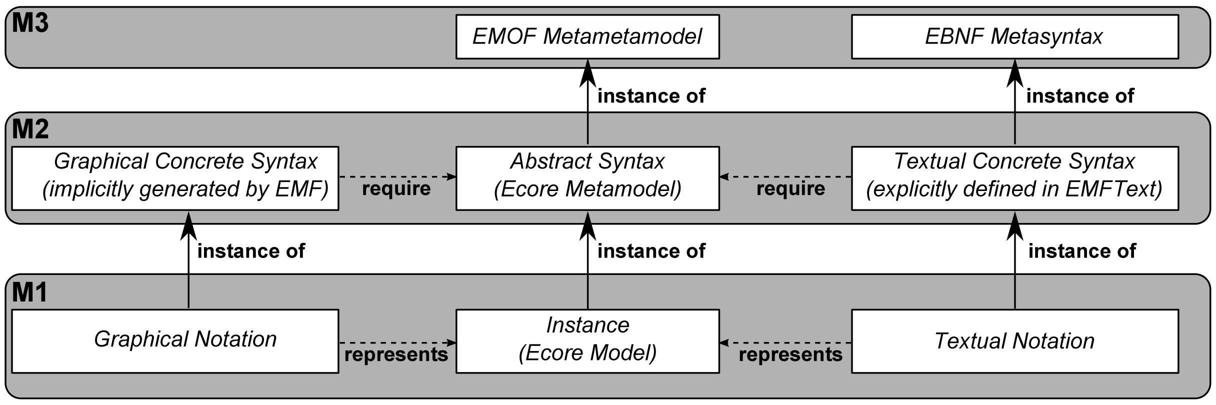


Figure 7.3 The metalevels of a language specification applying *EMF* and *EMFText*.

concrete syntax is required, as the metamodel on *M2* only defines the structure and thus the *abstract syntax* of a language. The EMF framework implicitly generates a tree-like graphical concrete syntax for each metamodel defined on level *M2*. This graphical notation is then applied on level *M1* to present Ecore models graphically in a tree-like notation. The graphical notation generated by EMF can further be customized to apply different icons per syntax element, for instance. An additional textual concrete syntax is specified according to the metamodel definition on level *M2* in EMFText. EMFText defines textual syntax complying to Human-Usable Textual Notation (HUTN), since HUTN is an OMG standard for textual notations conforming to abstract syntax specifications in MOF-based metamodels [OMG2004]. In EMFText the concrete syntax of a language is defined as a context-free grammar in Extended Backus-Naur Form (EBNF) notation [HJK$^+$09]. Each textual concrete syntax explicitly defined by EMFText on level *M2* is an instance of the EBNF metasyntax on level *M3*. EMFText defines a language mapping between EBNF and EMOF. A model instance corresponding to the abstract syntax of the metamodel can be instantiated on level *M1* in textual notation following the grammar defined in EMFText. Hence, model instances on level *M1* can be represented both, in a graphical tree-like notation and in a textual notation. To be conceptually complete, there is a further level *M0* in the MDA definition representing a particular system in terms of software objects describing real world objects abstracted by the instance model on level *M1*.

Based on the abstract syntax definition, EMF generates a Java representation of a language and generic methods to traverse model instances [Gro09]. Both EMF and EMFText generate convenient language editors to instantiate and manipulate models. These editors check the well-formedness of models with respect to the abstract syntax definition. Further restrictions on a model to be valid are defined by means of static semantics. For instance, the names of referenced elements must be already defined. EMFText generates algorithms for evaluating static semantics by means of static name and type analysis [HJK$^+$13]. Hence, default reference resolution mechanism is generated by EMFText comprising generated methods to resolve names that are unique in the model. In PUMA not all reference names are unique, and thus, the generated methods are further customized in EFeatureText and RBACText to resolve the correct references. Additionally, the dynamic semantics of the generated languages are implemented operationally by language interpreters in PUMA.

7.1.2. Serialization of Models in the Tool Suite

A generic exchangeable document format for models is defined in EMF. Models are persisted in XML Metadata Interchange (XMI) files in the file system. XMI is an interchange format defined by an Extensible Markup Language (XML) schema [OMG2011a]. EMF generates Java interfaces from the metamodel definition in Ecore to parse information stored as XMI and to access model elements. The XMI files are loaded via generated EMF parsers into a model repository on *Layer 1* of the tool suite, as visualized in Figure 7.1. Tools access the model repository via generated interfaces to process and manipulate models. In turn, generated EMF printers serialize models as XMI files in the file system. Furthermore, EMFText generates parsers and a printers to load and persist these models textually using the Another Tool for Language Recognition (ANTLR)[6] parser generator. As EMFText is an extension to EMF, models with

[6]`http://www.antlr.org/`

textual notation defined in EMFText are serializable as structured text, as well as XMI files, while their logical representation in the model repository of the tools equals. In the tool suite, feature models, multi-perspective mapping models and access control models are also serializable as text files, since their textual notations are defined in EMFText.

7.1.3. Graphical versus Textual Notation

In general, a graphical notation makes it is easy to comprehend relationships and types in a model, while a textual notation is compact, and computable by both humans and machines. Domain experts prefer a textual notation over a graphical representation as they overlook the described concepts better and are faster in typing than modeling graphically [GP92]. Hence, providing a textual and a graphical representation for a model is convenient. As discussed above, the combination of EMF and EMFText allows for both.

The tool suite PUMA comprises three model-based textual DSLs, (1) a DSL to specify extended feature models with attributes over finite domains, their configuration states, and cross-tree constraints, (2) a DSL for applying RBAC to define restrictions on feature model configuration operations, and (3) a DSL to define perspectives by mapping feature models to view models. The definitions of these DSLs in EMFText are explained in the following.

7.2. EFeatureText – Textual Language for Extended Feature Models

Various textual DSLs to describe variability have been proposed and a standardization process for a common textual variability language is conducted. However, the feature model DSLs found in literature are either not realized in the technology space of Eclipse and EMOF or do nor cover all feature model constructs considered in this thesis.

The intention of the textual DSL *EFeatureText* is to provide a comprehensive human and machine readable self-documenting notation for group-cardinality based feature models with attributes over finite domains, as introduced in Section 6.3. Furthermore, the textual DSL is integrated in EMF and Eclipse to easily specify, access and exchange feature models within the PUMA tool suite. The abstract syntax of the language is implemented in EMF and expressed by an Ecore metamodel complying to the structure of extended feature models shown in Figure 6.3. The source code and documentation of EFeatureText comprising examples are provided online in a Github repository and the related wiki[7]. EMFText is applied to define the concrete syntax for the textual notation complying to the metamodel concepts. The textual concrete syntax of EFeatureText is explained in the following.

[7]`https://github.com/extFM/extFM-Tooling/wiki/Extended-Featuremodelling`

7.2.1. Concrete Syntax Rules of EFeatureText

For each metaclass in the feature model metamodel a concrete syntax rule is specified to textually represent the syntax of this metaclass, its attributes and references to other metaclasses. Keywords are used in these rules as syntactic elements to markup and structure expressions. The following rules are specified for extended feature models. The root container of a feature model instance is the `FeatureModel`. The concrete syntax rule for this metaclass is presented in Listing 7.1. The left side of the rule specifies the defined metaclass by name and the right side specifies defined syntax elements.

```
FeatureModel ::= "featuremodel" name['"','"'] domains* root constraints*;
```

Listing 7.1 Concrete syntax rule for the `FeatureModel` metaclass.

A `FeatureModel` metaclass is defined by the keyword *featuremodel* followed by the `name` attribute defined in apostrophes, containing potentially many `Domain` classes, a single obligatory root `Feature` class and potentially many cross-tree `Constraint` classes. The star behind referencing metaclasses denotes that referenced elements occur zero or multiple times. A minimal example of the definition of a `FeatureModel` is depicted in Listing 7.2. Note that comments can be inserted in concrete syntax definitions in a Java-like syntax starting with two slashes.

```
featuremodel "Example Featuremodel"
  //definition of domains, root feature, and cross-tree constraints
  ...
```

Listing 7.2 Example for the application of the concrete syntax rule for the `FeatureModel` metaclass.

Another important element in the feature model metamodel is the `Feature`. The concrete syntax rule for `Feature` metaclasses is presented in Listing 7.3.

```
Feature ::= configurationState[selected:"selected", deselected:"deselected",
    unbound:""] "feature" name['"','"'] "<" id[] ">" (attributes | groups)*;

TOKENSTYLES {
  "selected" COLOR #009E0F, BOLD;
  "deselected" COLOR #CE0000, BOLD;
}
```

Listing 7.3 Concrete syntax rule for the `Feature` metaclass.

The concrete syntax rule of a `Feature` starts with a specification of its enumeration attribute `configurationState` subsequently followed by the keyword *feature*, its obligatory attributes `name` and `id` and potentially many containing `Attribute` and `Group` classes. As explained in Section 6.3.1 the configuration state of a feature is either *undecided*, *selected* or *deselected* modeled by corresponding enumeration literals in the metamodel. For each literal a corresponding textual representation is given in the concrete syntax to identify the literals by this keyword. The literals *selected* and *deselected* are defined by corresponding keywords. The default configuration state for features is *undecided*, and hence represented by an empty string in the concrete syntax rule. For a better readability, configuration states are highlighted in the textual notation. Hence, colored token styles are defined in the concrete syntax specification of EFeatureText causing the keyword *deselected* to be displayed in red and the keyword *selected* displayed in green. The example in Listing 7.4 depicts example EFEatureText definitions of features with different configuration states. `Feature 1` is undecided, `Feature 2` is selected, and `Feature 3` is deselected.

```
feature "Feature 1" <f1>
selected feature "Feature 2" <f2>
deselected feature "Feature 3" <f3>
```

Listing 7.4 Example for the application of the concrete syntax rule for the `Feature` metaclass.

For the metaclass `Group` the concrete syntax rule is depicted in Listing 7.5. The `Group` represents groups of features having a cardinality to uniformly model optional, mandatory, alternative, and or feature groups. The definition of the `Group` metaclass starts with the keyword *group* followed by the specification of the attribute `id` in angle brackets, the lower and upper boundaries of the group in round brackets and the reference of potentially multiple, but at least one containing child `Feature` classes in curly brackets. Lower and upper boundaries of the group are assigned to the corresponding `minCardinality` and `maxCardinality` attributes. As these attributes are of type integer in the metamodel, the specification of the token `INTEGER` in square brackets guarantees that these elements are parsed as numbers and not as strings as it is default for elements without specified tokens.

```
Group ::= "group" "<" id[] ">" "(" minCardinality[INTEGER] ".." maxCardinality[
    INTEGER] ")" "{" childFeatures+ "}";
}
```

Listing 7.5 Concrete syntax rule for the `Group` metaclass.

An example of the definition of the metaclass `Group` is provided in Listing 7.6 representing an alternative feature group with two features.

```
group <g1> (0..1) {
  feature "Feature 1" <f1>
  feature "Feature 2" <f2>
}
```

Listing 7.6 Example of the application of the concrete syntax rule for the `Group` metaclass.

An attribute domain is the set of values allowed in an attribute. Attribute domains are either specified as numerical domains containing intervals of integers or as discrete domains specifying a set of domain values each having an integer representation to make domain values of both domain types comparable. Concrete syntax rules for defining the metaclass `NumericalDomain` and the referenced metaclass `Interval` are shown in Listing 7.7.

```
NumericalDomain ::= "domain" "<" id[] ">" "[" intervals ("," intervals)* "]";
Interval ::= lowerBound[INTEGER] ".." upperBound[INTEGER];
```

Listing 7.7 Concrete syntax rule for the `NumericalDomain` and the `Interval` metaclasses.

A `NumericalDomain` is defined by the keyword *domain* followed by its attribute id in angle brackets and potentially multiple but at least one contained `Interval` classes specified in square brackets. Additionally, the concrete syntax rules for the metaclass `DiscreteDomain` and its referencing metaclass `DomainValue` are shown in Listing 7.8.

```
DiscreteDomain ::= "domain" "<" id[] ">" "[" values ("," values)* "]";
DomainValue ::= (name[] "=")? int[INTEGER];
```

Listing 7.8 Concrete syntax rule for the `DiscreteDomain` and the `DomainValue` metaclasses.

A `DiscreteDomain` is defined by the keyword *domain* followed by its attribute id in angle brackets and potentially multiple but at least one contained `DomainValue` classes specified in square brackets.

An example for defining domains is given in Listing 7.9. The listing shows four example domains. The first two domains with the id `discrete1` and `discrete2` are discrete domains each specifying three discrete values. Each discrete domain value is represented by a name as string followed by a numerical representation as integer. The latter two domains with id `num1` and `num2` specify intervals of integers. Domain `num1` contains one interval with a lower bound of 1 and an upper bound of 5. Domain `num2` contains a further interval with a lower bound of 10 and an upper bound of 100.

```
domain <discrete1> [red=1, blue=2, green=3];
domain <discrete2> [left=0, middle=1, right=2];
domain <num1> [1..5]
domain <num2> [1..5, 10..100]
```

Listing 7.9 Example application of the concrete syntax rules for the `NumericalDomain` and `Interval` metaclasses, as well as for the `DiscreteDomain` and the `DomainValue` metaclasses.

The metaclass `Attribute` is defined by the concrete syntax rule depicted in Listing 7.10. Each attribute is defined by a name, a referenced domain followed by potentially many deselected domain values referenced by `DomainValue` classes and an optional value attribute assignment. The notation of representing deselected domain values as a comma separated list in curly brackets starts with the leading keyword *without*.

```
Attribute ::= name[] "[" domain[] "]" ("without" "{" deselectedDomainValues['"',
    '"'] ("," deselectedDomainValues['"','"'])* "}")? (":=" (value['"','"']))?;
```

Listing 7.10 Concrete syntax rule for the `Attribute` metaclass.

An example of the definition of `Attribute` classes is shown in Listing 7.11. A discrete domain containing the color values *red, blue, green* and *yellow* is specified referenced by the four defined attributes in this example.

Line 3 in the listing contains the definition of the unconfigured attribute `attribute1` which references domain `d1` but does not specify deselected or assigned domain values. In contrast, attribute `attribute2` is set as it has value *blue* of the referenced domain assigned. Attribute `attribute3` is defined to reference domain `d1` but without the domain values *red* and *yellow*. Finally, the attribute `attribute4` equals the previous attribute but has the value *green* assigned.

```
domain <d1> [red=1, blue=2, green=3, yellow=4]
...
attribute1 [d1]
attribute2 [d1] := "blue"
attribute3 [d1] without {"red", "yellow"}
attribute4 [d1] without {"red", "yellow"} := "green"
```

Listing 7.11 Example of the application of the concrete syntax rule for the `Attribute` metaclass.

Constraints on feature models are separated into constraints on features and constraints on attributes. Feature constraints are either imply or exclude constraints expressed by the metaclasses `Imply` and `Exclude`. The concrete syntax definitions of these metaclasses are depicted in Listing 7.12. Both metaclasses start with the keyword *constraint* followed by the attribute id in angle brackets. Left and right operands of both constraints are `Feature` classes, while the operator differs. `Imply` classes are identified by the `->` keyword representing an imply operator while `Exlude` classes are identified by the `<->` keyword.

```
Imply   ::= "constraint" "<" id[] ">" leftOperand[] "->"  rightOperand[];
Exclude ::= "constraint" "<" id[] ">" leftOperand[] "<->" rightOperand[];
```

Listing 7.12 Concrete syntax rules for the `Imply` and `Exclude` metaclasses.

An example of applying syntax rules for these feature constraint metaclasses is given in Listing 7.13. In this example three features are assumed with the identifiers `f1`, `f2`, and `f3`. Based on these features, two constraints are defined applying the concrete syntax rules for `Imply` and `Exclude`. The first constraint with id `c1` is an instance of the metaclass `Imply` identified by the keyword `->` between the left and the right referenced operands. In addition, the `Imply` class references feature `f1` as a left operand and feature `f2` as a right operand. The second constraint with id `c2` is an instance of the metaclass `Exclude` identified by the keyword `<->` between the left and the right referenced operands. The `Exclude` class references feature `f1` as a left operand and feature `f2` as a right operand.

```
feature "Feature 1" <f1>
feature "Feature 2" <f2>
feature "Feature 3" <f3>
...
constraint <c1> f1 -> f2
constraint <c2> f3 <-> f2
```

Listing 7.13 Examples of applying the concrete syntax rules for the `Imply` and `Exclude` metaclasses.

Furthermore, attribute constraints are defined by the metaclass `AttributeConstraint` and its referencing metaclass `AttributeReference`. The concrete syntax rules for both metaclasses are depicted in Listing 7.14. The syntax rule for the `AttributeConstraint` starts with the keyword *constraint* similar to the syntax rules for `Imply` and `Exclude` metaclasses. However, the keyword is followed by referencing an attribute operand defined by the `AttributeOperand` metaclass, a relational operator and a second referenced `AttributeOperand`. In the metamodel, an operator enumeration is defined comprising literals representing relational operators on numerical values.

While in the metamodel, a string name is applied, the concrete syntax rule defines a Java-like notation for the operators. The metaclasses `AttributeValue` and `AttributeReference` inherit from the abstract metaclass `AttributeOperand`. For each of these metaclasses a syntax rule is specified, as depicted in the listing. The concrete syntax rule for an `AttributeValue` defines it to be either a name of a discrete domain value represented as a string or an integer representation of a domain value. Furthermore, the `AttributeReference` is defined by a syntax rule comprising a reference to a feature id and a reference to an attribute name. The combination of feature id and attribute name prevents ambiguity in parsing the syntax rule as multiple features can have attributes of the same name.

```
AttributeConstraint ::= "constraint" "<" id[] ">" attribute1 operator[equal:"=="
    , unequal:"!=", greaterThan:">", greaterThanOrEqual:">=", lessThan:"<",
    lessThanOrEqual:"<="] attribute2;
AttributeValue ::= (name['"','"'] | int[INTEGER]);
AttributeReference ::= feature[]"."attribute[];
```

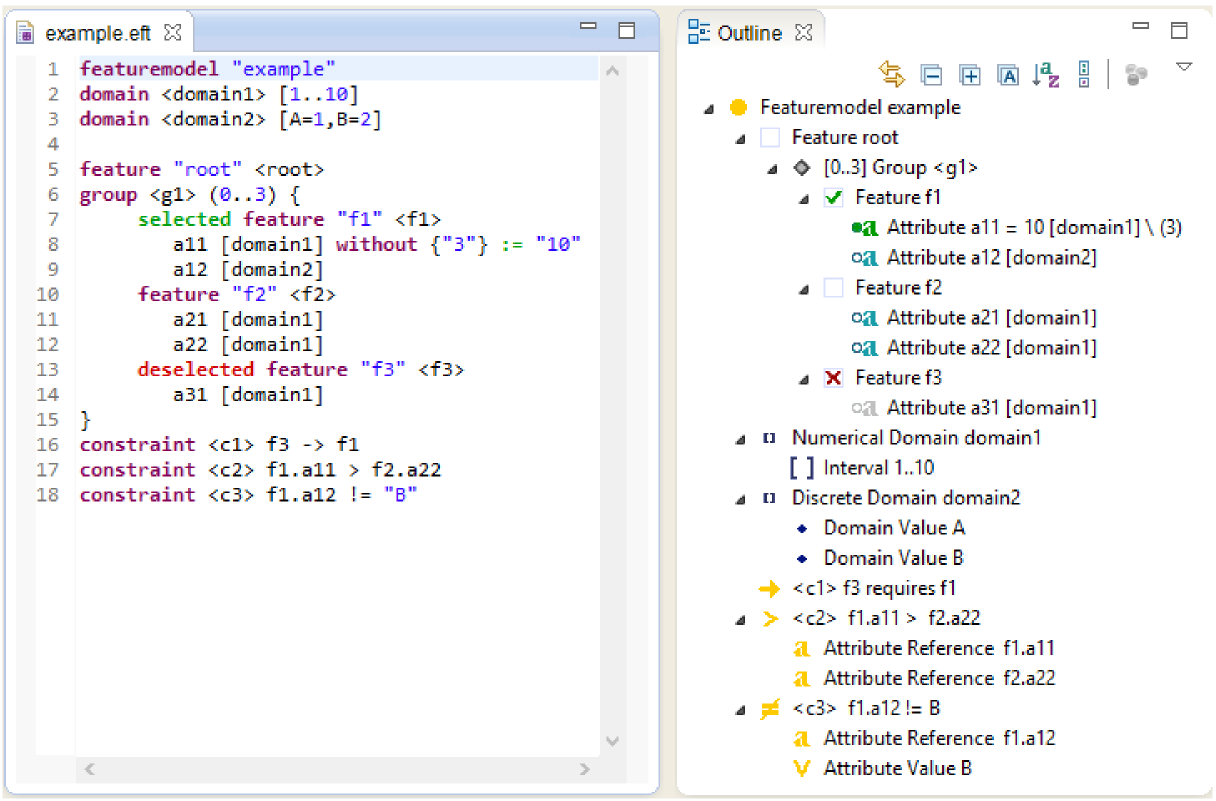
Listing 7.14 Concrete syntax rules for the `AttributeConstraint` and `AttributeReference` metaclasses.

Listing 7.15 shows an example of attribute constraints defined by the concrete syntax rule for `AttributeConstraint` metaclasses. In this example two features with the identifiers `f1` and `f2` are assumed. Both features have two attributes each. The attribute name `a1` is used by both features, however the domains of these attributes differ. Three attribute constraints are defined by applying the concrete syntax rules introduced above depicted in lines 9 to 11. The first instantiated constraint has id `ac1`. It defines a constraint with a relational `greaterThan` operator between two `AttributeReferences`, which are attribute `a1` of feature `f1`, and attribute `a2` of the same feature. The second constraint `ac2` defines a `lessThan` operator between the two `AttributeReferences` of attributes with the same name `a1` of the features `f1` and `f2`. The last constraint `ac3` defines an `unequal` constraint between the `AttributeReference` to attribute `a22` of feature `f2` and the `DomainValue` represented by the string `blue`.

```
 1 ...
   feature "Feature 1" <f1>
 3   a1  [d1]
     a2  [d2]
 5 feature "Feature 2" <f2>
     a1  [d2]
 7   a22 [d2]
   ...
 9 constraint <ac1> f1.a1 > f1.a2
   constraint <ac2> f1.a1 < f2.a1
11 constraint <ac3> f2.a22 != "blue"
```

Listing 7.15 Example of applying the concrete syntax rules for the `AttributeConstraint` and `AttributeReference` metaclasses.

For each textually specified model in the generated EMFText editor a synchronized outline view visualizing the graphical notation is integrated in the Eclipse UI. An example of a feature model specification in textual notation following the concrete syntax rules explained above is depicted on the left side in Figure 7.4 opposed to the rendered graphical notation in the outline view shown on the right side. The visualized feature model in this figure is named `example`. It comprises two attribute domains. The first defined domain with id `domain1` is a numerical



```
featuremodel "example"
domain <domain1> [1..10]
domain <domain2> [A=1,B=2]

feature "root" <root>
group <g1> (0..3) {
      selected feature "f1" <f1>
          a11 [domain1] without {"3"} := "10"
          a12 [domain2]
      feature "f2" <f2>
          a21 [domain1]
          a22 [domain1]
      deselected feature "f3" <f3>
          a31 [domain1]
}
constraint <c1> f3 -> f1
constraint <c2> f1.a11 > f2.a22
constraint <c3> f1.a12 != "B"
```

```
Featuremodel example
  Feature root
    [0..3] Group <g1>
      Feature f1
        Attribute a11 = 10 [domain1] \ (3)
        Attribute a12 [domain2]
      Feature f2
        Attribute a21 [domain1]
        Attribute a22 [domain1]
      Feature f3
        Attribute a31 [domain1]
  Numerical Domain domain1
    [ ] Interval 1..10
  Discrete Domain domain2
    Domain Value A
    Domain Value B
  <c1> f3 requires f1
  <c2> f1.a11 > f2.a22
    Attribute Reference f1.a11
    Attribute Reference f2.a22
  <c3> f1.a12 != B
    Attribute Reference f1.a12
    Attribute Value B
```

Figure 7.4 Graphical and textual notation of extended feature models applied in the PUMA tool suite.

domain specifying a value interval between 1 and 10. The second domain with id `domain2` is a discrete domain defining the discrete values `A` with integer representation 1 and `B` with integer representation 2.

In this example, names and identifiers of features equal. It is crucial that identifiers do not contain white space and are unique, while this restriction does not hold for names. Hence, the root feature has the name `root` defined in apostrophes and the id `root`. This features contains a group with id `g1` of optional features modeled by the lower bound 0 of the group and the upper bound 3 which equals the number of the contained features. The first feature `f1` is selected containing two attributes. The selected state of the feature is highlighted in the textual notation by the green color and in the graphical notation by a green tick mark. The first attribute `a11` of feature `f1` references domain `domain1`, while the second attribute `a12` references domain `domain2`. Furthermore, the first attribute has the domain value 1 assigned represented by a green filled circle in front of the attribute in the graphical notation. In addition, the domain value 3 is deselected for attribute `a11`. Attribute `a12` can still be assigned depicted by an empty blue circle in the graphical notation.

The second feature of the group `g1` has id `f2` and is yet undecided. This feature contains two unassigned attributes `a21` and `a21`, both referencing domain `domain1`. However, a third feature `f3` is contained in the group `g1`, which is deselected. The deselected state is highlighted in red in the textual notation and by a red check mark in the graphical notation. In addition, this feature contains an attribute `a31` with domain `domain1`. As the feature of this attribute is deselected, the attribute is disabled depicted by a gray icon in the graphical notation. Three cross-tree constraints are additionally specified in this example. The first constraint with id `c1` is a feature imply constraint between the feature `f3` and the feature `f1`. The second constraint is a relational less than constraint between attribute `a11` of feature `f1` and the attribute `a22` of feature `f2`. The last constraint `c3` is an attribute constraint on attribute `a12` of feature `f1` defining that this attribute should not equal the domain value `B` of the attribute domain.

As shown in this example, the textual notation depicted on the left side in Figure 7.4 is more compact than the graphical tree-like notation depicted on the right side. A domain expert skilled in the textual notation is faster in defining features, attributes and feature relations textually than in specifying the domain concepts in a graphical way. Writing concepts textually is usually faster than graphical editing. However, the graphical notation visualizes elements and configuration states of the feature model in a reasonable way using icons. For a user of a feature model and to check the defined relations, a graphical expression makes relations easier to understand. In addition, a conditional evaluation of feature and attribute states is implemented for the graphical notation allowing for selecting state-specific icons, as explained in the example above.

The complete concrete syntax is presented in Appendix B together with feature model examples in textual EFeatureText notation and the corresponding graphical notation applied in the outline view. An example for specifying an extended feature model in EFeatureText Notation is shown in the next section.

7.2.2. Business ByDesign Example in EFeatureText Notation

For *Business ByDesign* the feature model containing 78 features, 2 attributes and 23 cross-tree constraints is specified in EFeatureText and described in Appendix B.2. An excerpt of the feature model containing 19 features, 2 attributes and 3 cross-tree constraints is depicted in FODA notation in Figure 2.3 and described in Section 2.3.1. This feature model is specified in EFeatureText notation in Listing 7.16. The listing shows a complete configuration of the *Business ByDesign* feature model were all features are either selected or deselected and attribute values are set. The displayed configuration comprises 10 selected and 9 deselected features and equals the configuration presented in FODA notation in Figure 2.4.

```
featuremodel "Business ByDesign"
domain <Count> [10..10000]

selected feature "Business ByDesing" <bbd>
group <stakeholderConfiguration> (1..1){
  selected feature "Stakeholders" <Stakeholders>
    Employees [Count] := "100"
    Users [Count] := "15"
}
group <modules> (1..5) {
  selected feature "Marketing" <Marketing>
  group <marketingSelection> (1..2) {
    selected feature "Market Development" <Market_Development>
    selected feature "Campaign Management" <Campaign_Management>
  }
  deselected feature "Purchasing" <Purchasing>
  group <purchasingSelection> (1..2) {
    deselected feature "Self-Service Procurement" <SelfService_Procurement>
    deselected feature "Purchase Request and Order Management" <
   Purchase_Request_and_Order_Management>
  }

  selected feature "Supply Chain Setup Management" <
   Supply_Chain_Setup_Management>
  group <supplychainSetupSelection> (1..3) {
    selected feature "Supply Chain Design" <Supply_Chain_Design>
    deselected feature "Execution Design" <Execution_Design>
    deselected feature "Production Models" <Production_Models>
  }

  selected feature "Project Management" <Project_Management>
  group <projectSelection> (1..1) {
    deselected feature "Project Planning and Execution" <
   Project_Planning_and_Execution>
    selected feature "Basic Project Planning" <Basic_Project_Planning>
  }
  group <cashFlowSelection> (0..2) {
    deselected feature "Expense and Reimbursement Management" <
   Expense_and_Reimbursement_Management>
    selected feature "Payment and Liquidity Management" <
   Payment_and_Liquidity_Management>
  }

  deselected feature "Product Development" <Product_Development>
  group <productSelection> (1..1) {
    deselected feature "Product Engineering" <Product_Engineering>
  }
}

constraint <require1> Campaign_Management -> Market_Development
constraint <require2> Project_Planning_and_Execution ->
   Purchase_Request_and_Order_Management
constraint <exclude3> Basic_Project_Planning <-> Product_Engineering
constraint <attr4> Stakeholders.Employees >= Stakeholders.Users
```

Listing 7.16 Complete configuration of the extended feature model specification of the Business ByDesign excerpt in textual EFeatureText notation.

7.2.3. Language Tooling for EFeatureText

The framework EMFText in combination with the EMF framework generates a comprehensive language tooling comprising a parser, an editor with syntax highlighting and an evaluation engine for ensuring well-formedness according to the language definition of EFeatureText.

Beside the syntactic well-formedness criteria defined by the structure of the metamodel, further static semantic rules are to be implemented to guarantee model consistency. For instance, to ensure the uniqueness of identifiers. The EMF validation framework[8] implements static semantics on model elements. Rules implemented in the validation framework are automatically checked when manipulating a model in the generated graphical EMF editor and the textual EMFText editor. For extended feature models, a consistency rule is implemented in Java to check the uniqueness of identifiers. Another rule is implemented to ensure that deselected domain values and assigned domain values of an attribute are included in the referenced attribute domain.

EFeatureText files are persisted as text and are identified by the file extension *eft*. An EFeatureText interpreter is manually implemented to apply the language in the PUMA tool suite. Among others, the interpreter executes configuration operations on extended feature models.

7.2.4. Related Work on Feature Model Languages

Various textual variability models have been proposed in literature [ES13]. Table 7.1 gives an overview of textual DSLs for feature models found in related work and compares them with EFeatureText. The languages are compared by (i) their capability of defining attributes, (ii) support of group-cardinality, (iii) defining relational constraints on attributes, (iv) the definition of configuration states of features and attributes, and (v) their abstract syntax definition as EMOF compliant metamodel. An applicable language for the PUMA tool suite is required to combine these characteristics.

Feature Description Language (FDL) is a textual notation for feature diagrams specified in FODA with additional or feature groups proposed by van Deursen and Klint [VDK02]. The language neither supports attributes nor group-cardinality. Another textual language proposed by Acher is called Feature Model Script Language for Manipulation and Automatic Reasoning (FAMILIAR) [ACLF13]. FAMILIAR offers language and tool support for feature models, manipulation of feature models and analysis. This language focusses on feature models represented in propositional form and does not cover attribute nor group-cardinality based feature models. Clafer is a language for specifying metamodels, feature models and combinations of

[8]http://www.eclipse.org/modeling/emf/?project=validation

Table 7.1 Comparison of textual DSLs for feature models

Language	**Attributes**	**Group-Cardinality**	**Relational Constraints**	**Configuration States**	**EMOF metamodel**
FDL [VDK02]	-	-	-	-	-
FAMILIAR [ACLF13]	-	-	-	-	-
Clafer [BCW11]	+	+	+	+	-
TVL [CBH11]	+	+	+	-	-
VSL [APS+10]	+	+	-	+	+
EFeatureText	+	+	+	+	+

both [BCW11]. The language can be used for multi-objective optimizations of feature model configurations. However the language does not provide an EMOF compliant metamodel. Textual Variability Language (TVL) is a textual DSL for attributed group-cardinality based feature models with attributes and relational constraints proposed by Classen et al. [CBH11]. Syntax and semantics for the language are provided. However, TVL is not based on an EMOF metamodel definition and does not support the expression of feature configuration states. Variability Specification Language (VSL) is a textual specification language for feature models included in the Compositional Variability Management (CVM) Framework [APS$^+$10]. The language is integrated in Eclipse and complies to an EMOF metamodel definition. VSL is capable of expressing feature and attribute configuration states. However, VSL does not support the definition of relational constraints on attributes.

A further textual DSL for feature models applied in the tool FeatureMapper, developed at the software technology group at the Technische Universität Dresden, is available in the EMFText concrete syntax zoo[9]. The textual notation of EFeatureText is similar to this DSL while the expresiveness of the languages differ. However, the metamodels of the languages differ, as well as the concrete syntax rules, especially for the definition of attributes and constraints. In the language applied in the FeatureMapper, attributes, their domains and constraints are specified in a String representation. Furthermore, the name of a feature must be unique. In contrast, the abstract syntax of EFeatureText defines particular finite attribute domains and explicitly specifies allowed constraints while feature names are separated from unique feature identifiers. Furthermore, constraints and feature groups are identifiable by a unique identifier as well.

To summarize, EFeatureText provides a human readable notation for specifying group-cardinality based feature models and attributes together with configuration states. The language is based on a EMOF metamodel definition and implemented in EMFText. Hence, generated language tooling can be easily integrated into the Eclipse-based PUMA tool suite.

7.3. RBACText – Textual Language for Role Based Access Control on Extended Feature Models

The intention of the textual DSL *RBACText* is to provide a comprehensive language to define access control restrictions on feature model configuration operations. More information and the source code of RBACText are provided online in the Github repository and a related wiki[10].

RBACText is a scripting language implementing the concepts of applying RBAC on extended feature models as described in Section 6.4. Configuration operations on features and attributes as described in Section 6.3.1 are defined textually in RBACText. Hence, the language allows to define roles and restrict the access on configuration operations on extended feature models. The abstract syntax of RBACText is defined by the EMOF compliant metamodel depicted in Figure 6.8 in Section 6.4. A concrete syntax for the textual specification of RBAC concepts in a human readable and machine interpretable DSL is defined based on this abstract syntax. The

[9] `http://emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_Feature_Models`
[10] `https://github.com/extFM/extFM-Tooling/wiki/Role-Based-Access-Control-on-Featuremodels`

complete concrete syntax specified in EMFText and an example instantiation is included in the Appendix in Chapter C. In the following the concrete syntax rules of RBACText are explained.

7.3.1. Concrete Syntax Rules of RBACText

For each metaclass in the access control metamodel a concrete syntax rule is specified to textually represent the syntax of this metaclass, its attributes and references to other metaclasses. Keywords are used in these rules as syntactic elements to markup and structure expressions. The following rules are specified for defining role-based access control on extended feature models.

The root container of an access control instance is the `AccessControlModel`. The concrete syntax rule for this metaclass is presented in Listing 7.17. An `AccessControlModel` class is identified by the keyphrase *access control* and references a `FeatureModel` defined in the concrete syntax by the keyword *on* followed by the path to the `FeatureModel` in the Eclipse workspace. Contained elements of the `AccessControlModel` are `Permissions`, `Groups`, and `Subjects` specified in arbitrary order in the following.

```
AccessControlModel ::= "access control" "on" featureModel['<','>'] (roles |
    groups | subjects)* ;
```

Listing 7.17 Concrete syntax rule for the `AccessControlModel` metaclass.

A definition of an `AccessControlModel` is exemplified in Listing 7.18. The defined model references a feature model in textual EFeatureText notation depicted by the file extension *eft*. The path to the referenced feature model is relational to the persisted `AccessControlModel`. Permissions, roles, groups and subjects are specified according to their concrete syntax rules explained in the following.

```
access control on <../path/feature_model.eft>
// roles, groups and subjects are defined subsequently
```

Listing 7.18 Example for applying the concrete syntax rule for the `AccessControlModel` metaclass.

Permissions on feature models are expressed by means of configuration operations. Configuration operations are select and deselect of features, assigning attribute values and deselecting attribute domain values. These operations are expressed as concrete syntax rules as follows. The rule for a `FeatureOperation` starts with specifying its type, as depicted in Listing 7.19. The type is identified by the corresponding keyword, *select* or *deselect*, followed by the feature id.

```
FeatureOperation ::= type[select : "select", deselect : "deselect"] feature[TEXT
    ];
```

Listing 7.19 Concrete syntax rule for the `FeatureOperation` metaclass.

An example for the `FeatureOperation` concrete syntax rule is given in Listing 7.20. Two feature operations are defined for the feature with id *feature1* which is assumed to exist in the referenced feature model. The first operation defines the selection of the feature, while the second operation in the second line defines the deselection of the feature.

```
select feature1;
deselect feature1;
```

Listing 7.20 Example for applying the concrete syntax rule for the `FeatureOperation` metaclass.

The rule for the `AttributeValueOperation` metaclass shown in Listing 7.21 equals the rule for the `FeatureOperation` metaclass. The referenced attribute value of this metaclass is fully qualified by the feature id and attribute name each separated by a dot. The definition of `AttributeValueOperations` allows for a fine grained specification of access permissions on the level of attribute domain values.

```
AttributeValueOperation ::= type[select : "select", deselect : "deselect"]
    feature[] "." attribute[TEXT]  "."  value[TEXT];
```

Listing 7.21 Concrete syntax rule for the `AttributeValueOperation` metaclass.

An example for defining configuration operations on attribute values is visualized in Listing 7.22. In this example a feature *feature1* with an attribute *attribute1* is assumed. The attribute *attribute1* references a discrete domain comprising the domain values *red* and *green*. The listing shows select and deselect `AttributeValueOperations` per domain value.

```
select feature1.attribute1.green
deselect feature1.attribute1.green
select feature1.attribute1.red
deselect feature1.attribute1.red
```

Listing 7.22 Example for applying the concrete syntax rule for the `AttributeValueOperation` metaclass.

Instead of specifying permissions on each attribute value separately, an `AttributeOperation` can be specified to abstract from select and deselect operations on all attribute values of the attribute's domain. The concrete syntax rule for specifying an `AttributeOperation` is visualized in Listing 7.23. The rule starts with the keyword *assign* followed by the id of the referenced feature and the attribute name separated by a dot.

```
AttributeOperation ::= "assign" feature[] "." attribute[TEXT] ;
```

Listing 7.23 Concrete syntax rule for the `AttributeOperation` metaclass.

An example for defining configuration operations on an attribute is visualized in Listing 7.24. The listing defines an `AttributeOperation` on the attribute *attribute1* of feature *feature1*. This rule represents coarse grain access on all domain values of an attribute and can therefore be assumed as syntactic sugar for avoiding to specify an access control rule per domain value.

```
assign feature1.attribute1
```

Listing 7.24 Example for applying the concrete syntax rule for the `AttributeOperation` metaclass.

The concrete syntax rule for the metaclass `Role` is depicted in Listing 7.25. The definition of a `Role` starts with specifying its type. A role can be abstract or concrete, as explained in Section 6.4. The default type of a role is concrete. If the role is abstract, the definition of this role starts with the keyword *abstract*. A role is subsequently identified by the keyword *role* followed by an optional name and mandatory id definition. The subsequent keyword *extends* followed by a comma separated list of referenced `Roles` expresses that each role references potentially multiple parent `Roles` and inherits their permissions. The reference to parent roles is optional denoted by a question mark and parenthesizing the expression. An optional comma separated list of potentially multiple referenced `Permissions`, enclosed in curly brackets, completes the concrete syntax rule for the `Role` metaclass.

```
Role ::= type[abstract : "abstract", concrete : ""] "role" name['"','"']? id['<'
    ,'>'] ("extends" (parentRoles[]) ("," parentRoles[])*)? (("{" permissions[]
    ("," permissions[] )* "}") )?;
```

Listing 7.25 Concrete syntax rule for the `Role` metaclass.

An example for instantiating the metaclass `Role` is visualized in Listing 7.26. In this example two roles are defined with different permissions on configuration operations are defined. An abstract role *r1* and a concrete role *r2* are defined in this listing. Additionally, the role with id *r1* is parent of the other role with id *r2*. Hence, role *r2* has the permissions assigned to the parent role, as well as permissions directly assigned.

```
abstract role "Role 1" <r1> { select feature1, deselect feature1 }
role "Role 2" <r2> extends r1 { assign feature1.attribute1 }
```

Listing 7.26 Example for applying the concrete syntax rule for the `Role` metaclass.

The `Subject` metaclass is defined by the concrete syntax rule depicted in Listing 7.27 starting with the keyword *subject* and followed by an optional name and an id. Each subject plays at least on role expressed by the keyword *plays* followed by a comma separated list of roles.

```
Subject ::= "subject" name['"','"']? id['<','>'] "plays" (roles[] ("," roles[])*
    )?;
```

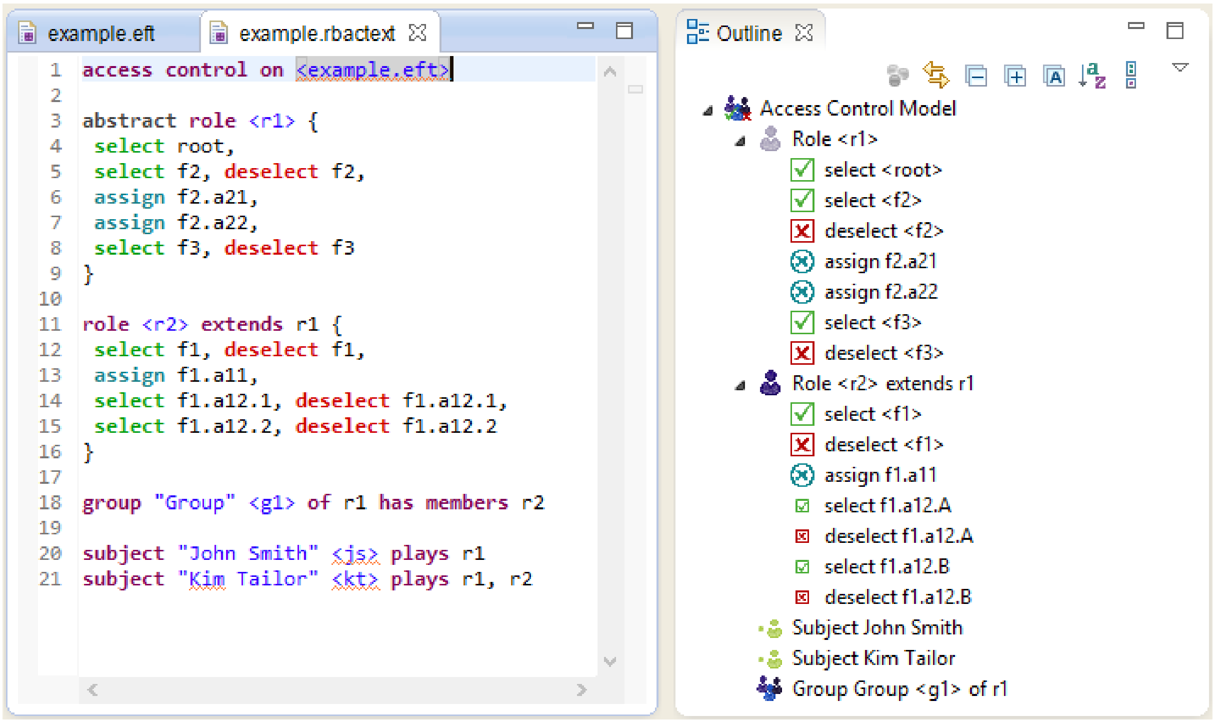
Listing 7.27 Concrete syntax rule for the `Subject` metaclass.

An example for instantiating the metaclass `Subject` is visualized in Listing 7.28. Two subjects are defined to play different roles specified by the role identifiers. While the first subject plays only one role *r1*, the second subject can play two roles *r1* and *r2*.

```
subject "John Smith" <js> plays r2
subject "Jane Doe" <jd> plays r1, r2
```

Listing 7.28 Example for applying the concrete syntax rule for the `Subject` metaclass.

The concrete syntax rule of the `Group` metaclass depicted in Listing 7.29 starts with the keyword *group* followed by an optional name and an id. Subsequently, the optional group owner `Role` is specified starting with the keyword *of* followed by the referenced group owner id. Each group references potentially multiple `Roles` as group members expressed by the keyphrase *has members* followed by a comma separated list of referenced member `Roles`.



```
access control on <example.eft>

abstract role <r1> {
 select root,
 select f2, deselect f2,
 assign f2.a21,
 assign f2.a22,
 select f3, deselect f3
}

role <r2> extends r1 {
 select f1, deselect f1,
 assign f1.a11,
 select f1.a12.1, deselect f1.a12.1,
 select f1.a12.2, deselect f1.a12.2
}

group "Group" <g1> of r1 has members r2

subject "John Smith" <js> plays r1
subject "Kim Tailor" <kt> plays r1, r2
```

```
Access Control Model
  Role <r1>
    select <root>
    select <f2>
    deselect <f2>
    assign f2.a21
    assign f2.a22
    select <f3>
    deselect <f3>
  Role <r2> extends r1
    select <f1>
    deselect <f1>
    assign f1.a11
    select f1.a12.A
    deselect f1.a12.A
    select f1.a12.B
    deselect f1.a12.B
  Subject John Smith
  Subject Kim Tailor
  Group Group <g1> of r1
```

Figure 7.5 Graphical and textual notation of access control on extended feature models applied in the PUMA tool suite.

```
Group ::= "group" name['"','"']? id['<','>'] ("of" owner[])? "has members" (
    members[] ("," members[])*)?;
```

Listing 7.29 Concrete syntax rule for the `Group` metaclass.

An example for instantiating the metaclass `Group` is illustrated in Listing 7.30. A group with name *group 1* and id *g1* is instantiated. The member of this group is role *r1*, while the two members are *r2* and *r3*.

```
group "group 1" <g1> of r1 has members r2, r3
```

Listing 7.30 Example for applying the concrete syntax rules for the `Subject` metaclass.

For each model specified textually in the generated EMFText language editor according to the explained concrete syntax rules, a graphical notation is displayed in the outline view. This graphical notation is generated by EMF, based on the abstract syntax definition in the appropriate Ecore metamodel. An example for a textual notation of RBACText is depicted on the left side of Figure 7.5 and opposed to its graphical notation applied in the outline view visualized on the right side. The graphical notation is customized to present comprehensible icons per metaclass element. The example refers to the extended feature model definition visualized in Figure 7.4.

In this example, the configuration operations on the feature model are distributed to two roles `r1` and `r2`. Role `r1` is allowed to select an deselect the features `root`, `f2`, and `f3` and assign the attributes `a21` and `a22` of feature `f2`. In contrast, role `r2` is allowed to select and deselect feature `f1` and to assign attribute values to the attributes `a11` and `a12` of this feature. Note that for attribute `a12` the selection and deselection operations are specified for each attribute value by instantiating the `AttributeValueOperation`, while for the other attributes the short form instantiating the metaclass `AttributeOperation` is used. Furthermore, a group named `Group` with id `g1` is defined, where role `r1` is the owner and role `r2` is the group member. Finally, two subjects that play roles are specified. One subject has the name `Johm Smith` with id `js` playing the role `r1`. A second subject with the name `Kim Tailor` with id `kt` playing the roles `r1` and `r2`. A larger example for RBACText can be found in the Appendix C.

7.3.2. Tooling for RBACText

Similar to the EFeatureText language, a parser, editor and evaluation engine is generated by the EMFText and EMF frameworks according to the abstract and concrete syntax definition of RBACText. RBACText files are persisted as text and are identified by the file extension *rbactext*. The language is interpreted by a manually implemented RBACText interpreter in the PUMA tool suite. Model instances are programmatically accessed via a controller service. The service provides interface methods to correctly evaluate the multi-inheritance relation of roles.

Beside the syntactic well-formedness criteria defined by the structure of the access control metamodel, further static semantics are implemented applying the EMF validation framework. Those rules are automatically checked when manipulating a model in the generated graphical EMF editor and the textual EMFTextEditor. For access control models, a rule is implemented to check the uniqueness of identifiers. A further rule is implemented to check that domain values assigned to attributes or deselected from an attribute are included in the attribute domain. Another rule ensures that no cycles are defined in the role hierarchy regarding parent and child role references. In addition, a member of a group cannot be owner as well. This constraint is implemented by a corresponding rule in the EMF validation framework. A further rule is implemented to ensure that the same configuration operation on a feature model element is not defined multiple times for the same role. A third implemented rule ensures that subjects are not directly assigned to abstract roles as an RBAC constraint is defined to only assign subjects to concrete roles.

7.3.3. Related Work on Textual Domain-Specific Languages for Role Based Access Control

A textual DSL to specify RBAC concepts in the Service-Oriented Architecture (SOA) context is provided in [HGS$^+$11]. In this work, RBAC containing a context extension is applied to restrict permissions on web service invocations. The language defines a textual notation to express roles, subjects, resources, operations, contexts and relations between these elements. However, the concrete syntax is not provided. In contrast to RBACText, this language is not implemented in the technological space of Eclipse and EMF and hence not applicable in the tool suite PUMA. Furthermore, RBACText explicitly models configuration operations on features and attributes, which are restricted by roles.

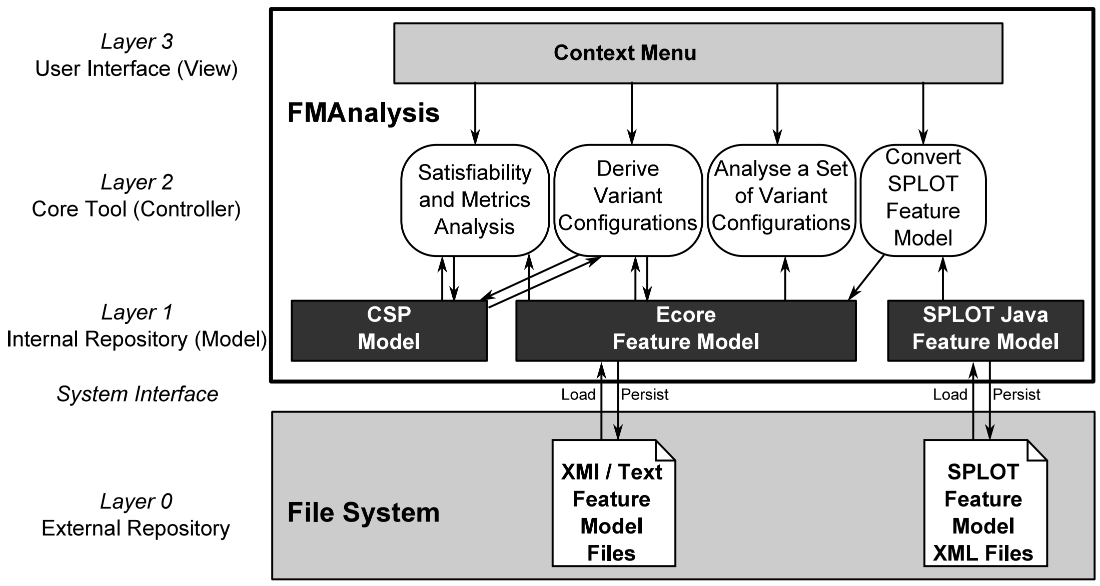

Layer 3
User Interface (View)
Layer 2
Core Tool (Controller)
Layer 1
Internal Repository (Model)
System Interface
Layer 0
External Repository
Context Menu
FMAnalysis
Satisfiability and Metrics Analysis
Derive Variant Configurations
Analyse a Set of Variant Configurations
Convert SPLOT Feature Model
CSP Model
Ecore Feature Model
SPLOT Java Feature Model
Load
Persist
Load
Persist
File System
XMI / Text Feature Model Files
SPLOT Feature Model XML Files

Figure 7.6 Architecture of FMAnalysis utilities.

7.4. FMAnalysis – Utilities for Feature Model Analysis

Some utility methods for extended feature model are implemented regarding satisfiability analysis, comparison of variant configurations, and the import of feature models from an online repository. The architecture of FMAnalysis utilities is depicted in Figure 7.6. The provided methods can be invoked by a user via context menu commands in the Eclipse IDE and programmatically via defined interfaces. More information and the source code of the utilities are provided online in the Github repository and the related wiki[11].

Satisfiability of an extended feature model and related partial configurations is analyzable by translating a feature model into a logical CSP representation and applying a CSP solver, as explained in Section 2.5. FMAnalysis allows for analyzing satisfiability of a feature model in EFeatureText notation by applying the *Choco*[12] CSP solver. This solver is available as Java libraries and was chosen as it is involved in the standard JSR-331[13] for Java-based constraint programming and therefore integrates into the Eclipse and Java technology space. Moreover, other authors reported the applicability of this solver in literature [BRCT05, MSDLM11, KOD10, PBN$^+$11]. If the CSP is solvable, the feature model is satisfiable.

The specification of feature and attribute configuration states in the metamodel for extended feature models depicted in Figure 6.3 unifies the modeling of unconfigured feature models, partial configurations and complete configurations. Hence, feature model configurations and unconfigured feature models are expressed by means of an instance of the feature metamodel. This allows for a checking their satisfiability in a uniform way.

[11]https://github.com/extFM/extFM-Tooling/wiki/Utilities
[12]http://www.emn.fr/z-info/choco-solver/
[13]http://jsr331.org/

Table 7.2 Metrics of the limited *Business ByDesign* feature model.

Metrics		**Value**
Total number of features		19
	Unbound features	19
	Selected features	0
	Deselected features	0
Total number of attributes		2
	Assigned attributes	2
Total number of cross-tree constraints		4
Constraint feature coverage		31%
Constraint attribute coverage		100%
Is feature model satisfiable		*true*
Total number of derivable variants		959

Translating an instance of the proposed metamodel is similar to translations described in literature [BRCT05, MSDLM11, KOD10, PBN$^+$11]. As discussed in Section 2.3.2, group cardinality enables a unified modeling of solitary and grouped features. Hence, translation rules for solitary features and special cases of feature groups are omitted in this approach. Further rules are introduced to translate the configuration states of features and attributes into the CSP. The translation rules applied in this work are summarized in Appendix A.

The translation rules for converting an Ecore feature model into a *Choco* CSP representation are implemented in Java. Hence, a transformative bridge between *Choco* CSP representation and *Ecore* feature model representation is defined in this work.

FMAnalysis is implemented as Eclipe plug-ins and encapsulate the CSP functionality by offering interfaces to check the satisfiability of extended feature models. As partial feature model configurations are modeled in PUMA as feature models too, the analysis methods are applicable on partial configurations as well.

Further analysis methods on extended feature models are available according to the explanation in [BSTRC05]. For instance, the number of derivable products can be calculated, as well as the number of features, attributes, and constraints. Hence, given a feature model as input, the output are feature model metrics as shown in Table 7.2 for the *Business ByDesign* feature model in Listing 7.16.

Furthermore, an algorithm is implemented to derive complete variant configurations from a given unconfigured or partially configured feature model. The configuration variants can be stored either textually in EFeatureText notation or in XMI conforming to the metamodel definition. The utilities can be applied to derive and persist all variant configurations or a particular number. In addition, analysis methods on a set of configurations are implemented. For instance, for a selected feature, the number of configurations containing this feature can be calculated. Another algorithm reveals intersecting features of a set of configurations. These methods are relevant for quality assurance of an SPL [Gol13]. For instance, to determine which features are frequently used in customer configurations and to identify obsolete features not used in configurations.

FMAnalysis provides a method to import feature models from the Software Product Line Online Tools (SPLOT) repository [SPLOT12]. SPLOT feature models are translated into the extended feature model representation applied in the PUMA tool suite. A library fo parsing the XML notation of a SPLOT file and rendering an corresponding Java representation is provided by SPLOT research[14]. This library is applied to import a SPLOT feature model. The transformative bridge for converting this feature model into EFeatureText and persisting in textual and XMI notation is implemented in Java.

Summarizing, FMAnalysis encapsulates common analysis methods for extended feature models, as well as partial and complete configurations. Furthermore, feature models from the SPLOT repository can be imported for evaluating the PUMA tool suite.

7.5. Conper – Consistent Perspectives and Views

The concepts of *multi-perspectives* discussed in Chapter 5 are implemented in the tool Conper and integrated as plug-ins in Eclipse [SLW12a]. Conper is a tool for conservatively extending feature models with perspectives to concisely specify and derive feature model refinements by means of perspectives. With Conper perspectives are derivable as filtered feature models and as pre-configurations where features not contained in a perspective are deselected, as explained in Section 5.2.

Conper is initially developed as an extension to the Featuremapper[15] tool and based on the feature model representation used by the Featuremapper. However, Conper is extended to support feature models with attributes over finite domains defined in EFeatureText to seamlessly integrate into the PUMA tool suite. The source code, further information and examples are available on Github[16]. The architecture of the tool is illustrated in Figure 7.7.

Conper applies three models, (i) a feature model given in EFeatureText notation or as XMI corresponding to the metamodel of EFeatureText, (ii) a view model defining the hierarchical structure of viewgroups and viewpoints given as XMI representation of the corresponding view metamodel, and (iii) a mapping model defined either in the textual notation of MText or as XMI file corresponding to the metamodel of MText.

To derive a perspective, a mapping between features of a feature model and viewgroups of a view model is defined in a textual notation called *MText*. The mapping model contains relative references to the feature model and the view model in the Eclipse workspace. These references are resolved by loading the models into the internal model repository.

[14]`http://gsd.uwaterloo.ca:8088/SPLOT/sxfm.html`
[15]`http://www.featuremapper.org/`
[16]`https://github.com/multi-perspectives/cluster/wiki`

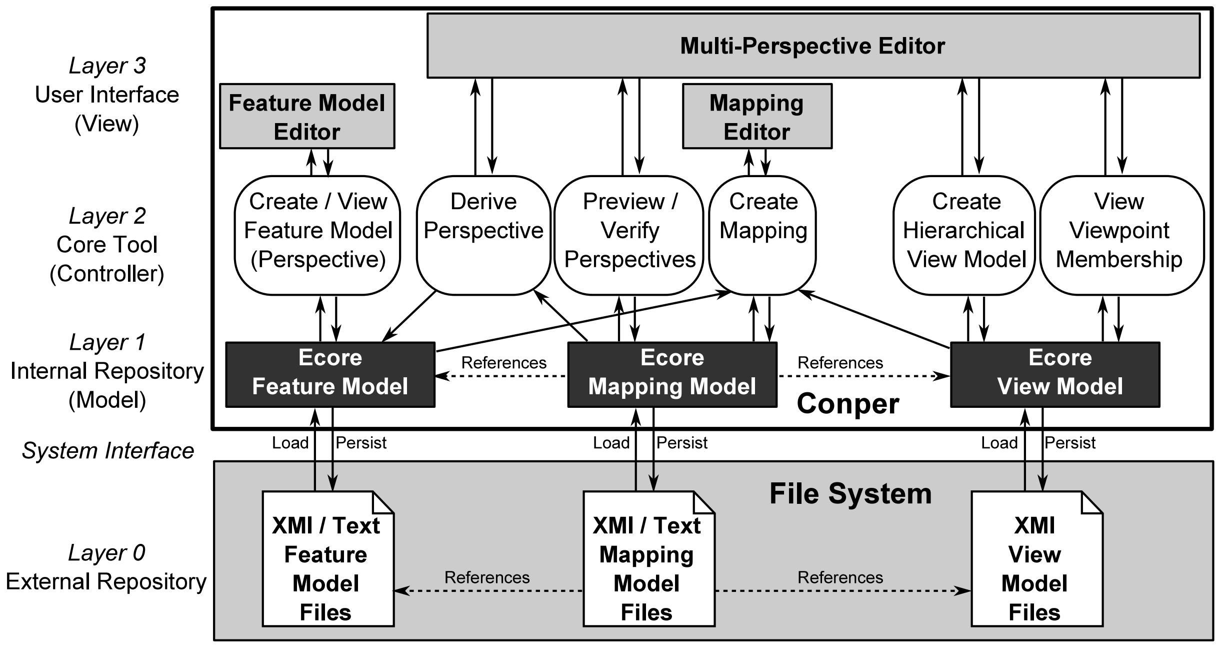


Figure 7.7 Tool architecture of Conper.

Feature models are defined and modified in the feature model editor provided by the EFeatureText language tooling. The multi-perspective editor is intended to create and persist the hierarchical view model comprising viewgroups and viewpoints and to verify and derive perspectives. The mapping is specified in the generated textual editor of this language.

As an example to illustrate the tool, the variable document management system introduced in Section 5.3 is implemented in Conper. The feature model of the document management system specified in textual EFeatureText notation is depicted in Listing 7.31.

```
featuremodel "Document Management System"

feature "Document Management System" <dms>
  group <DocumentTypeGroup> (1..1) {
    feature "Document Type" <DocumentType>
      group <TypesGroup> (1..4) {
        feature "UnicodeText Type" <UnicodeTextType>
        feature "Text Type" <TextType>
        feature "Image Type" <ImageType>
        feature "PDF Type" <PDFType>
      }
  }
  group <OCRGroup> (0..1) {
    feature "OCR" <OCR>
      group <OCRTypes> (1..2) {
        feature "PDF OCR" <PDFOCR>
        feature "Image OCR" <ImageOCR>
      }
  }
  group <IndexGroup> (1..1) {
    feature "Indexing" <Indexing>
      group <MetaIndexing> (0..1) {
        feature "MetaData Index" <MetaDataIndex>
          group <AuthorGroupIndex> (0..1) {
            feature "Author Index" <AuthorIndex>
          }
          group <TitleGroupIndex> (1..1) {
            feature "Title Index" <TitleIndex>
          }
          group <ContentGroupIndex> (1..1) {
            feature "Content Index" <ContentIndex>
          }
        feature "General Index" <GeneralIndex>
      }
      group <FileIndex> (1..1) {
        feature "FileName Index" <FileNameIndex>
      }
  }
  group <SearchGroup> (1..1) {
    feature "Search" <Search>
      group <MetaSearch> (0..1) {
        feature "MetaData Search" <MetaDataSearch>
          group <AuthorGroupSearch> (0..1) {
            feature "Author Search" <AuthorSearch>
          }
          group <TitleGroupSearch> (1..1) {
            feature "Title Search" <TitleSearch>
          }
          group <ContentGroupSearch> (1..1) {
            feature "Content Search" <ContentSearch>
          }
        feature "General Search" <GeneralSearch>
      }
      group <FileSearch> (1..1) {
        feature "FileName Search" <FileNameSearch>
      }
  }
constraint <c1> MetaDataSearch -> MetaDataIndex
constraint <c2> GeneralSearch -> GeneralIndex
constraint <c3> ImageOCR -> ImageType
constraint <c4> PDFOCR -> PDFType
constraint <c5> AuthorSearch -> AuthorIndex
constraint <c6> TextType <-> UnicodeTextType
```

Listing 7.31 Feature model specification of the document management system example written in EFeatureText.

The feature model contains 23 features and 6 cross-tree constraints on features and corresponds to the example described in Section 5.3. 208 valid variant configurations are derivable from this document management feature model. How to define a view model and a mapping between features and viewgroups is explained in the following.

7.5.1. Definition of View Models

A view model defines a hierarchy among potentially overlapping concerns. A metamodel is applied in Conper to specify view model instances according to the definition in Section 5.9.

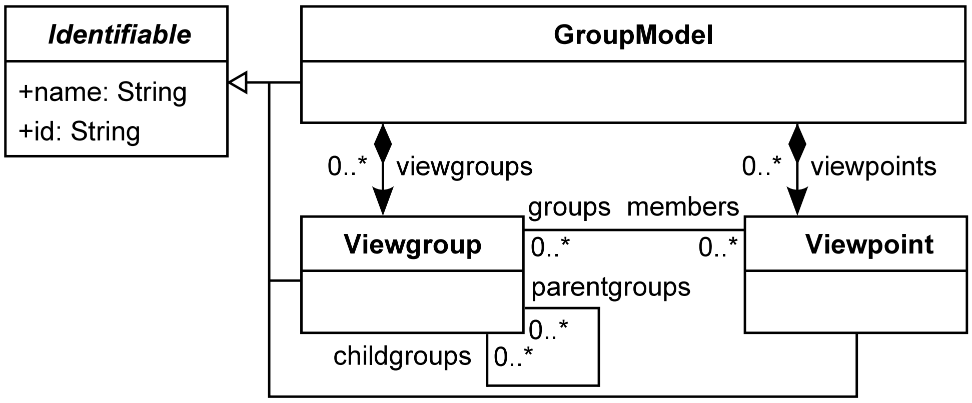


Figure 7.8 Metamodel of the hierarchical view model.

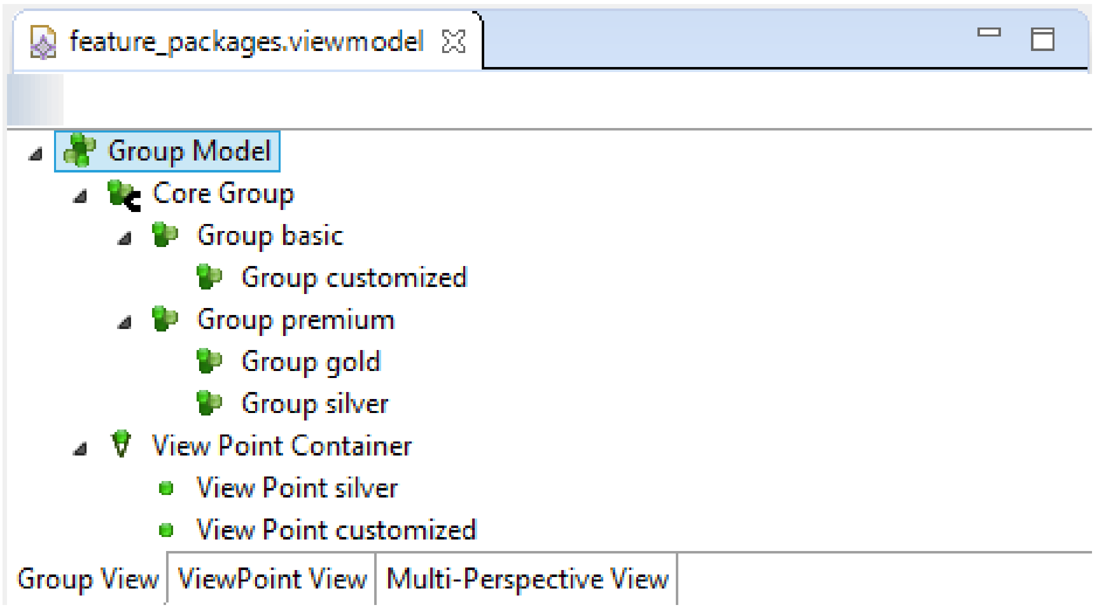


Figure 7.9 View model applied in the document management system example.

The metamodel is implemented in Ecore and depicted in Figure 7.8.

The root metaclass of the model is the `GroupModel` representing a view model. This class has containment references to a set of `Viewgroups` and a set of `Viewpoints`. Each `Viewgroup` represents a concern, while its hierarchical relation is defined by the relations `childgroups` and `parentgroups`. To each `Viewgroup` a set of `Viewpoints` can be assigned via the `members` reference. The opposite of this reference is a `groups` reference indicating that a `Viewpoint` can be assigned to multiple `Viewgroups`. The three metaclasse `GroupModel`, `Viewgroup`, and `Viewpoint` inherit a `name` and `id` attribute from the abstract metaclass `Identifiable` to be identifiable via a name and a unique identifier.

According to the metamodel, the EMF framework generates a graphical editor for instantiating view models. The generated editor is further extended for visualizing the relation between a given viewpoint and a set of viewgroups. The view model applied in the document management example explained in Section 5.3 is implemented in Conper and depicted in Figure 7.9.

The mapping between viewgroups and features is specified in a separate mapping model, as explained in the following section. Hence, mapping and view models are exchangeable and can be reused in various multi-perspective models.

7.5.2. Mapping Between Feature Model and View Model

A mapping model defines the mappings between features of the feature model and viewgroups in a view model. The metamodel for instantiating mapping models is defined in Ecore, as depicted in Figure 7.10. The root metaclass of the model is `MappingModel` which references a view model by means of the `GroupModel` metaclass and an extended feature model by means of the `FeatureModel` metaclass.

A `MappingModel` contains a set of `Mapping`s, each representing a mapping between a single viewgroup of the view model and a set of features of the feature model. Hence, the `Mapping` metaclass references the metaclasses `Viewgroup` contained in the `GroupModel` and `Feature` contained in the `FeatureModel`. Each mapping is subsequently interpreted to define a view on a feature model.

Mappings between features and viewgroups are specified textually in a DSL named *MText*. The abstract syntax of the language is defined by the EMOF compliant metamodel depicted in Figure 7.10, while concrete syntax rules are specified in EMFText. The syntax rules for defining a mapping model, referencing a feature model and a view model, as well as specifying mappings is depicted in Listing 7.32.

```
MappingModel   ::=  "viewmapping" ("featuremodel" featureModel['<','>']) ("
    viewmodel" #1 viewModel['<','>']) (mappings*);
Mapping ::= "view group" viewgroup['"','"'] "contains" features['"','"']  (","
    features['"','"'] )*;
```

Listing 7.32 Concrete syntax rules for the mapping model.

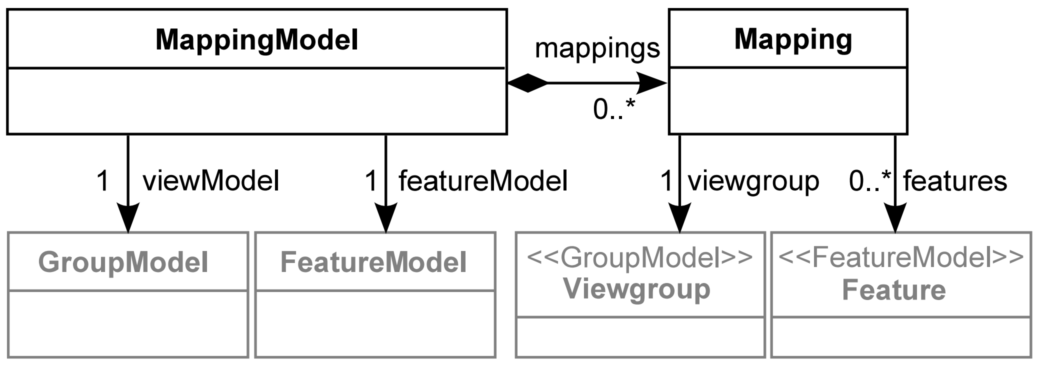


Figure 7.10 Metamodel of the mapping model for mapping features to viewgroups.

A `MappingModel` is identified by the keyword *viewmapping*. In addition, the keyword *featuremodel* followed by a reference to a `FeatureModel` in angle brackets specifies the referenced feature model instance. A `GroupModel` is defined similarly, beginning with the keyword *viewmodel*. Subsequently a set of `Mapping`s is defined.

The concrete syntax rule for the `Mapping` metaclass starts with the keyword *viewgroup* followed by a reference to a `Viewgroup` to define a mapping between a single `Viewgroup` and a set of `Feature`s. The keyword *contains* followed by a comma separated list of `Feature`s defines the referenced `Feature` instances. `Feature` and `Viewgroup` references are specified by their identifier in quotation marks. The complete specification of the concrete syntax comprising layout directives for the language printer is included in Appendix D.

Similar to the EFeatureText and RBACText languages, parser, editor and evaluation engine are generated by the EMFText and EMF frameworks according to the abstract and concrete syntax definitions of MText. MText files are persisted as text and are identified by the file extension *mtext*. The language is interpreted by a manually implemented MText interpreter in the PUMA tool suite. Model instances are programmatically accessed via a provided controller service.

A mapping between features and viewgroups specified in the textual notation is depicted in Listing 7.33. The listing shows an example mapping between the feature model of a document management system and a view model driven by business concerns.

```
viewmapping
  featuremodel <../feature/documentmanagement.feature>
  viewmodel <../viewmodel/documentmanagement.viewmodel>

view group "Basic" contains
  "ImageType" "PDFType" "OCR"

view group "Premium" contains
  "PDFType"
  "MetaDataIndex" "TitelIndex" "ContentIndex"
  "MetaDataSearch" "TitleSearch" "ContentSearch"

view group "Silver" contains
  "AuthorIndex" "AuthorSearch"

view group "Gold" contains
  "ImageType"
   "PDFOCR" "ImageOCR"

view group "Customized" contains
  "UnicodeTextType"

view group "Core" contains
  "DocumentManagement"
  "DocumentType" "TextType"
  "Indexing" "GeneralIndex" "FileNameIndex"
  "Search" "GeneralSearch" "FileNameSearch"
```

Listing 7.33 Textual mapping language to assign document management features to business concern-related viewgroups.

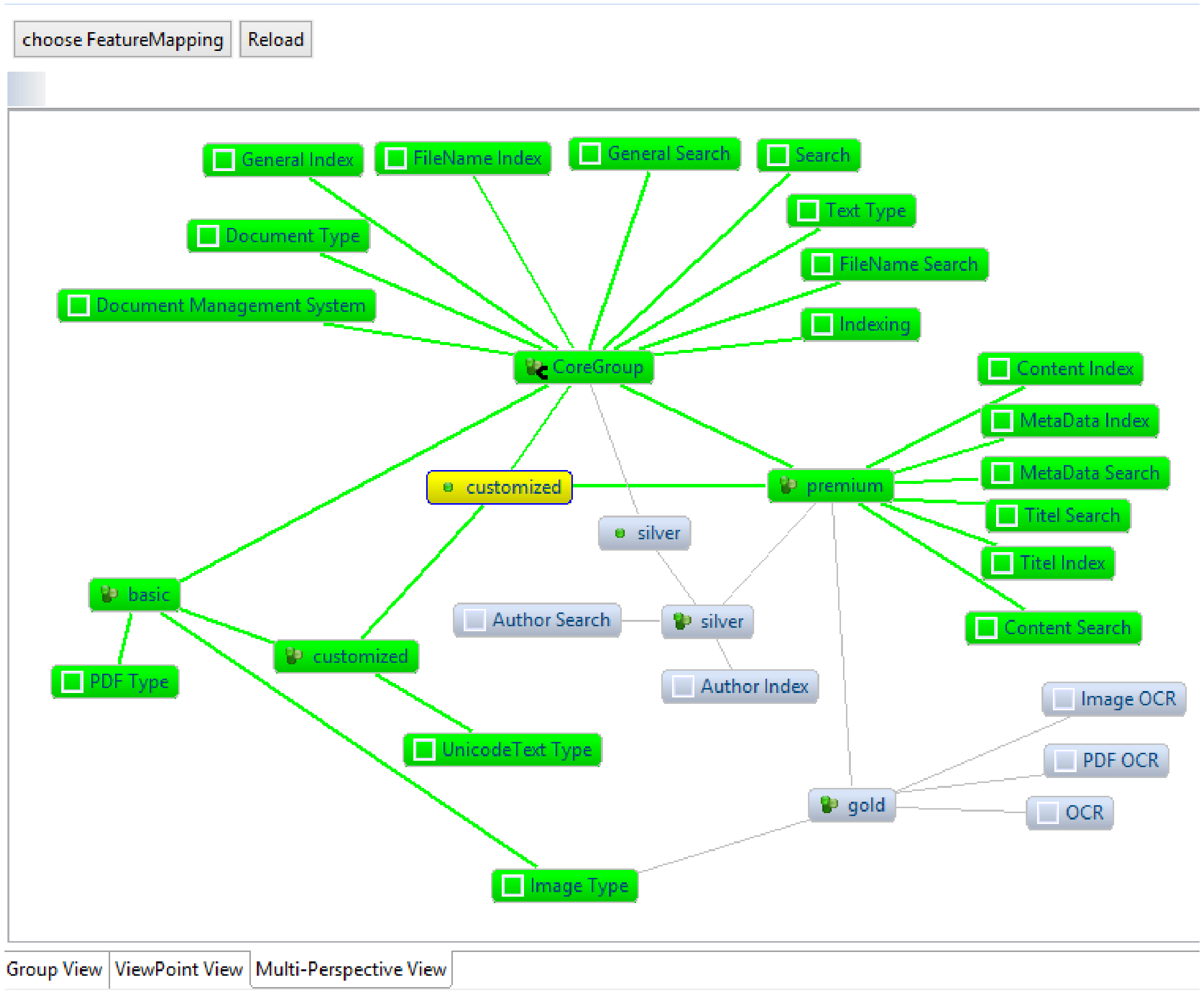


Figure 7.11 Screenshot of the multi-perspective editor.

A feature model, a view model and a mapping model comprise a multi-perspective model. The relations between these models are visualized graphically in the multi-perspective editor in Conper, as explained in the next section.

7.5.3. Multi-Perspective Editor

The multi-perspective editor interprets the mapping model and provides an overview of the relation between viewgroups and assigned features. In addition, a preview of the perspectives defined by viewpoints is available by means of highlighting the set of comprised features. Figure 7.11 depicts a screenshot of the multi-perspective editor. The relations between elements in the multi-perspective model defined for the document management system are shown. Each viewgroup, viewpoint, and feature is visualized by a node and a corresponding icon.

The editor is capable of highlighting references between elements by double clicking on an element. For a viewgroup, features referenced by this viewgroup and hierarchically related viewgroups are highlighted. For a viewpoint, all related viewgroups are highlighted, as well as all features mapped to those viewgroups. Hence, for a viewpoint a preview of the corresponding perspective is available.

In Figure 7.11, the dependencies of viewpoint *customized* are highlighted in green, while the viewpoint is highlighted in yellow. Hence, viewgroups not assigned to a viewpoint can be easily identified, as well as unrelated features. Viewpoint *customized* is not related to the viewgroups *gold* and *silver*. A perspective derived by this viewpoint will therefore not contain the features *Author Search* and *Author Index* assigned to viewgroup *silver*, as well as the features *OCR*, *Image OCR* and *PDF OCR* assigned to viewgroup *gold*.

As shown in the figure, the multi-perspective editor visualizes the relations between viewgroups, viewpoints, and features in a comprehensive and interactive graphical representation. Conper offers a graphical editor for creating, visualizing and checking a multi-perspective model for consistency. Both consistency check algorithms introduced in Section 5.11 are implemented to check the consistency of a multi-perspective model. In addition, single viewpoints can be checked applying the brute-force consistency algorithm. For checking the complete multi-perspective model the efficient incremental algorithm is applied, as explained in Section 5.11.2.

7.5.4. Consistency of Viewpoints

The tool implements the brute-force and the incremental consistency check algorithms as defined in Section 5.11 to check the consistency of viewpoints. While the incremental consistency check is applied on multi-perspective models with a consistent core viewgroup, the brute-force algorithm is applied otherwise. Furthermore, single viewpoints can be checked solely for consistency by applying the brute-force algorithm.

The performance of both algorithms is evaluated on a set of feature models with varying numbers of features and cross-tree constraints. The feature model of the document management system presented before is applied, as well as a feature model of a flood crisis management system from a FeatureMapper case study [HSS+10]. Further feature models are imported from the

Table 7.3 Feature models applied in the performance evaluation of Conper.

Feature Model	Origin	Features	CTCs	CTCR	CSP Check
Document Mgt. System	own	23	6	52%	10,9 ms
Weather Station	SPLOT	23	2	17%	6,3 ms
CD OD Semantic Variability	SPLOT	32	4	18%	13,8 ms
Crisis Management	Featuremapper	84	0	0%	34,4 ms
Generated 1000 Features	SPLOT	1000	100	9%	374,6 ms
Generated 2000 Features	SPLOT	2000	100	7%	1121,8 ms
Generated 5000 Features	SPLOT	5000	150	5%	4239,1 ms
Generated 10000 Features	SPLOT	10000	100	2%	9675,9 ms

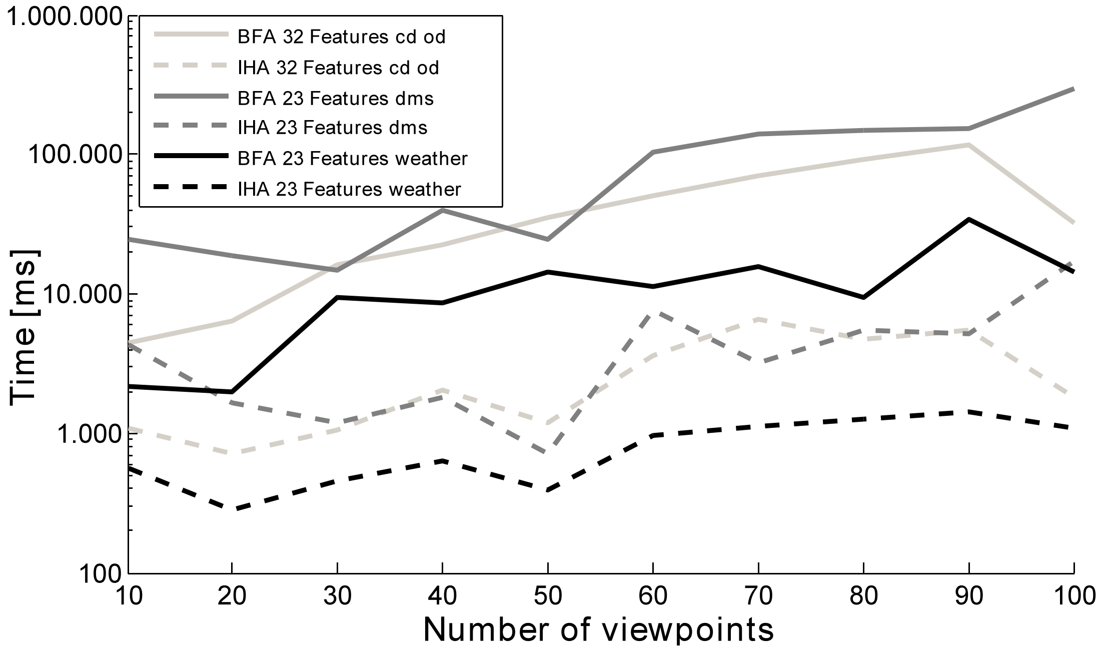


Figure 7.12 Varying the number of viewpoints.

SPLOT repository including 4 generated feature models with up to $10,000$ features. All feature models are satisfiable and listed in Table 7.3. The satisfiability of these feature models is checked by translating each model into a CSP and solving the model utilizing an interface from the FMAnalysis tool. In addition, applied multi-perspective models are randomly generated. View models with a maximum depth of 5, and most 3 child viewgroups are generated randomly. In addition, the mappings between features of feature models and viewgroups of the view model is generated randomly, as well assigning a maximum number of 5 features to a viewgroup. The performance is measured on a laptop with an Intel Core i5-2520M CPU comprising 2.5GHz and 8GB RAM running a Windows 7 Professional SP1 64-bit operating system.

In a first experiment, different numbers of viewpoints are generated for the same viewgroup hierarchy and assigned randomly to the viewgroups. The features of a feature model are mapped randomly to the viewgroups once. This mapping is reused while only the number of viewpoints varies. Figure 7.12 shows the time of the performance measure for three different feature models in relation with a varying number of viewpoints to be checked. The time on the x-axis is displayed logarithmical. The results are shown in this figure for the feature models *CD OD Semantic Variability*, *Document Management System*, and *Weather Station* depicted in Table 7.3.

The results of the experiment show that the incremental algorithm performs better for different numbers of viewpoints compared to the brute-force consistency check . This is due to the fact that the same paths in the view model are checked multiple times by the brute-force algorithm.

In a second experiment, feature models with different numbers of features are checked while the same view model is applied. Figure 7.13 visualizes the influence of the number of features in a feature model on the performance of the consistency algorithms. The time is displayed on the x-axis logarithmical.

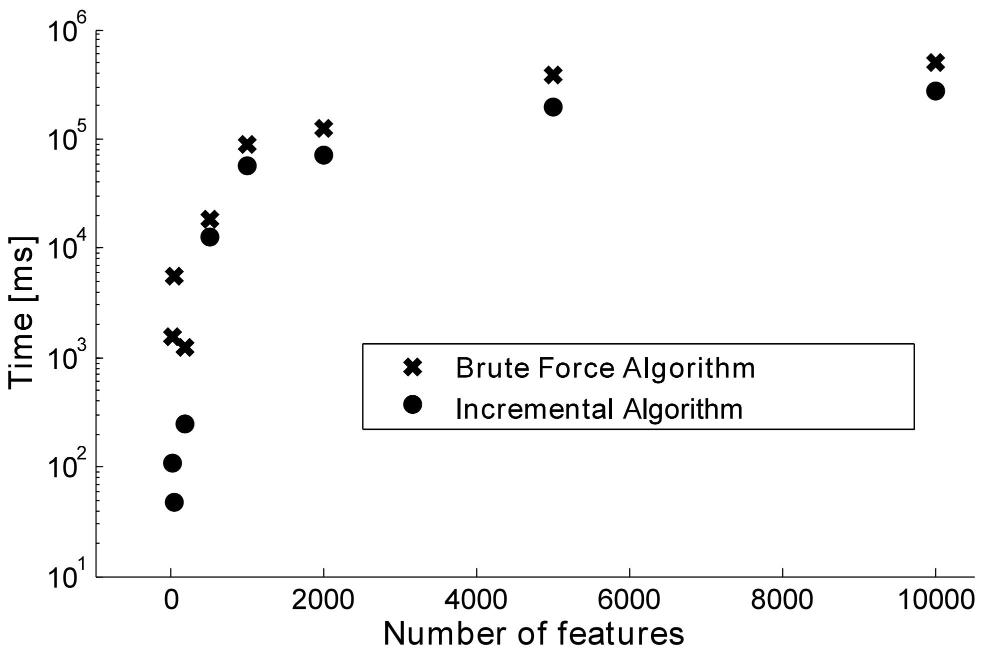


Figure 7.13 Feature models with varying sizes.

The figure shows results for the feature models described in Table 7.3. The experiment reveals that both algorithms scale with respect to the number of features. However, the performance of the incremental algorithm is always better, as expensive satisfiability checks are omitted. The performance converges to the performance of the brute-force algorithm only in the worst case, if all viewpoints are inconsistent in the view model.

The performance evaluation conducted in the explained two experiments confirms that the incremental consistency check is efficiently scalable to feature models and view models of different sizes. Checking the consistency of viewpoints is required to derive valid perspectives, as explained in the following section.

7.5.5. Derivation of Perspectives

A perspective is derived in Conper either as a partial feature model configuration or as a filtered feature model, as explained in Section 5.2 and visualized in Figure 5.3. In the first case, features not contained in a perspective are deselected, as depicted in Figure 5.3(a). In the latter case, features and related constraints not contained in the perspective are filtered out, as depicted in Figure 5.3(b).

In the multi-perspective editor, a single viewpoint is selected to derive a perspective. A command is available in the context menu to trigger the derivation of the corresponding perspective. The user specifies whether to derive a partial configuration or a filtered feature model.

As an example, the perspective of the viewpoint *customized* in the document management example is derived as a partial configuration and as a filtered feature model. The perspective of the viewpoint *customized* is represented as a partial configuration in Listing 7.34, where features not contained in the perspective are deselected. In contrast, Listing 7.35 shows the same perspective by means of a filtered feature model.

```
featuremodel "Document Management System"


 feature "Document Management System" <dms>
  group <DocumentTypeGroup> (1..1) {
     feature "Document Type" <DocumentType>
      group <TypesGroup> (1..4) {
         feature "UnicodeText Type" <UnicodeTextType>
         feature "Text Type" <TextType>
         feature "Image Type" <ImageType>
         feature "PDF Type" <PDFType>
        }
    }

  group <OCRGroup> (0..1) {
    deselected feature "OCR" <OCR>
      group <OCRTypes> (1..2) {
        deselected feature "PDF OCR" <PDFOCR>
        deselected feature "Image OCR" <ImageOCR>
        }
    }

  group <IndexGroup> (1..1) {
     feature "Indexing" <Indexing>
      group <MetaIndexing> (0..1) {
         feature "MetaData Index" <MetaDataIndex>
          group <AuthorGroupIndex> (0..1) {
            deselected feature "Author Index" <AuthorIndex>
            }

          group <TitleGroupIndex> (1..1) {
             feature "Title Index" <TitleIndex>
            }

          group <ContentGroupIndex> (1..1) {
             feature "Content Index" <ContentIndex>
            }
         feature "General Index" <GeneralIndex>
        }

      group <FileIndex> (1..1) {
         feature "FileName Index" <FileNameIndex>
        }
    }

  group <SearchGroup> (1..1) {
     feature "Search" <Search>
      group <MetaSearch> (0..1) {
         feature "MetaData Search" <MetaDataSearch>
          group <AuthorGroupSearch> (0..1) {
            deselected feature "Author Search" <AuthorSearch>
            }

          group <TitleGroupSearch> (1..1) {
             feature "Title Search" <TitleSearch>
            }

          group <ContentGroupSearch> (1..1) {
             feature "Content Search" <ContentSearch>
            }
         feature "General Search" <GeneralSearch>
        }

      group <FileSearch> (1..1) {
         feature "FileName Search" <FileNameSearch>
        }
    }
constraint <c1> MetaDataSearch -> MetaDataIndex
constraint <c2> GeneralSearch -> GeneralIndex
constraint <c3> ImageOCR -> ImageType
constraint <c4> PDFOCR -> PDFType
constraint <c5> AuthorSearch -> AuthorIndex
constraint <c6> TextType <-> UnicodeTextType
```

Listing 7.34 Derived perspective of the *customized* view point as a partial configuration.

Both representations define the same set of derivable variant configurations but differ in the contained features and constraints. For instance, the constraints *c3*, *c4*, and *c5* are removed from the filtered feature model in Listing 7.35 as the features restricted by these constraints are not contained.

In addition the features *Author Search*, *Author Index*, *OCR*, *Image OCR*, and *PDF OCR* are not contained in the filtered feature model. In the perspective represented as partial configuration, these 6 features are deselected, as depicted in Listing 7.34 by the red keyword *deselected* representing the feature configuration state.

```
featuremodel "Document Management System"


 feature "Document Management System" <dms>
  group <DocumentTypeGroup> (1..1) {
     feature "Document Type" <DocumentType>
      group <TypesGroup> (1..4) {
         feature "UnicodeText Type" <UnicodeTextType>
         feature "Text Type" <TextType>
         feature "Image Type" <ImageType>
         feature "PDF Type" <PDFType>
        }
    }

  group <IndexGroup> (1..1) {
     feature "Indexing" <Indexing>
      group <MetaIndexing> (0..1) {
         feature "MetaData Index" <MetaDataIndex>
          group <TitleGroupIndex> (1..1) {
             feature "Title Index" <TitleIndex>
            }

          group <ContentGroupIndex> (1..1) {
             feature "Content Index" <ContentIndex>
            }
          feature "General Index" <GeneralIndex>
        }

      group <FileIndex> (1..1) {
          feature "FileName Index" <FileNameIndex>
        }
    }

  group <SearchGroup> (1..1) {
     feature "Search" <Search>
      group <MetaSearch> (0..1) {
          feature "MetaData Search" <MetaDataSearch>
           group <TitleGroupSearch> (1..1) {
              feature "Title Search" <TitleSearch>
            }

           group <ContentGroupSearch> (1..1) {
              feature "Content Search" <ContentSearch>
            }
          feature "General Search" <GeneralSearch>
        }

      group <FileSearch> (1..1) {
          feature "FileName Search" <FileNameSearch>
        }
    }
constraint <c1> MetaDataSearch -> MetaDataIndex
constraint <c2> GeneralSearch -> GeneralIndex
constraint <c6> TextType <-> UnicodeTextType
```

Listing 7.35 Derived perspective of the *customized* view point as a filtered feature model.

7.6. DyscoGraph – Dynamic Staged Configuration through Graph Rewriting

The tool *DyscoGraph* combines staged configuration with graph rewriting to model and execute a set of configuration and reconfiguration processes by means of a single specialization tree, as explained in Chapter 6.

Hence, DyscoGraph executes a specialization tree to derive multiple variant configurations by configuring and reconfiguring a given feature model. In addition, DyscoGraph allows for adapting the specialization tree during runtime by applying graph rewrite rules. The tool therefore comprises a workflow execution engine and an adaptation engine. The specialization tree is executed by the workflow engine to conduct staged configuration operations on the referenced feature model. In addition, an adaptation engine adapts the workflow during execution by applying rewrite rules to integrate and remove stakeholders. The architecture of DyscoGraph is depicted in Figure 7.14.

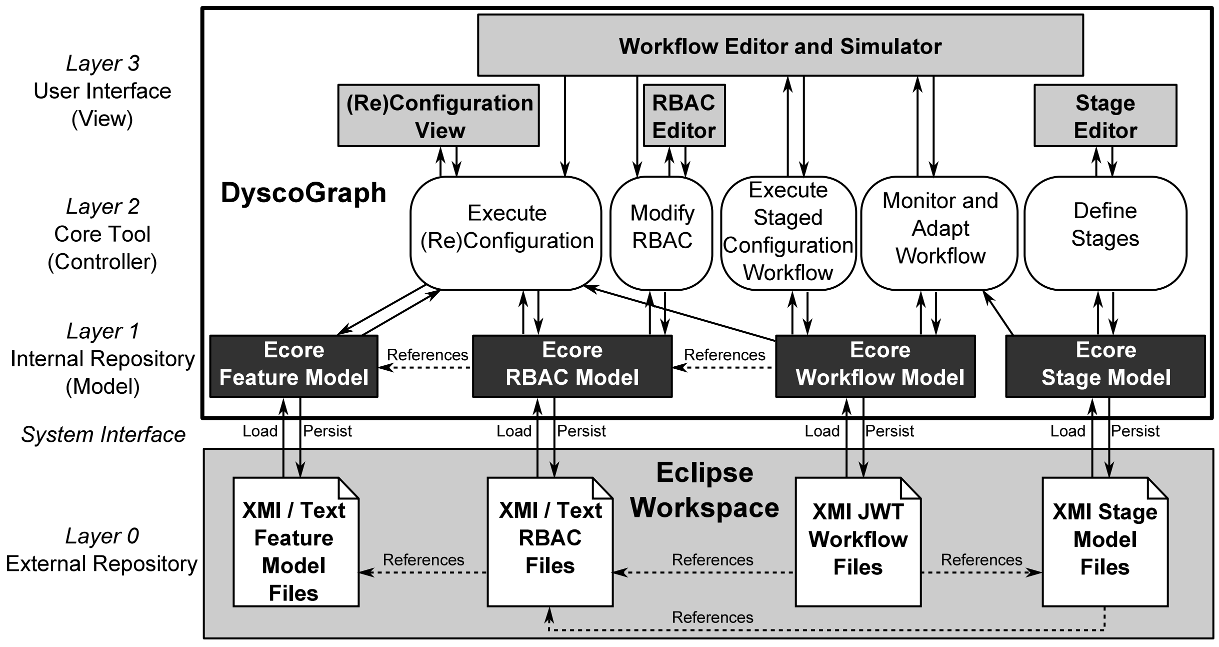


Figure 7.14 Tool architecture of DyscoGraph.

Staged reconfiguration workflows are modeled in EMF utilizing the Java Workflow Tooling (JWT) framework[17]. JWT is chosen as it is an official Eclipse project providing various extension mechanisms to integrate own concepts. The JWT framework is intended for modeling and executing business processes. The framework provides a graphical editor to define workflows in a graphical UML like activity diagram notation. Moreover, interfaces are provided to execute the workflow and adapt the graphical representation.

A JWT workflow in DyscoGraph constitutes a specialization tree to derive multiple variant configurations. Furthermore, a simulator is implemented in the tool to interpret the workflow model as a staged configuration workflow to stepwise configure an extended feature model. Configuration operations, access restrictions, and stakeholders are modeled in an access control model and specified in textual RBACText notation, as explained in Section 7.3.

Furthermore, JWT allows to integrate own models and concepts by applying aspect weaving. In DyscoGraph, references to a stage model, an access control model, as well as logs for specialization actions, are assumed as aspects and woven into the JWT workflow. A stage model defines predecessor and successor relationships among stages. Stages are applied to partition specialization actions and to identify search patterns in the rewrite rules for integrating and removing stakeholders. Logs persist the configuration operations conducted in a specialization action. Logs are applied during reconfiguration as explained in Section 6.6.2. The access control model defines configuration views for configuring stakeholders. A view is instantiated by executing an enabled specialization action via double click.

DyscoGraph implements a semi-automatic reconfiguration strategy to change partial configurations and propagate reconfiguration changes to depending stakeholders in the specialization tree.

[17]`http://www.eclipse.org/jwt/`

Rewrite rules are implemented in Java and modify the specialization tree and the referenced access control model during executing the staged configuration workflow. An adaptation engine is implemented to apply a sequence of rewrite rules to integrate or remove a stakeholder. Further information and the source code of DyscoGraph are provided online in the Github repository and the related wiki[18].

The graph rewrite rules for integrating and removing stakeholders during the execution of the staged configuration workflow, explained in Sections 6.7.2 and 6.7.4, are implemented in Java. The request to add or remove a stakeholder triggers the adaptation engine to apply the corresponding sequence of rewrite rules on the workflow model and the referenced access control model. After applying a sequence of rewrite rules, the specialization tree remains connected.

The applicability of the implemented concepts is evaluated on a yard management application, as explained in the next section.

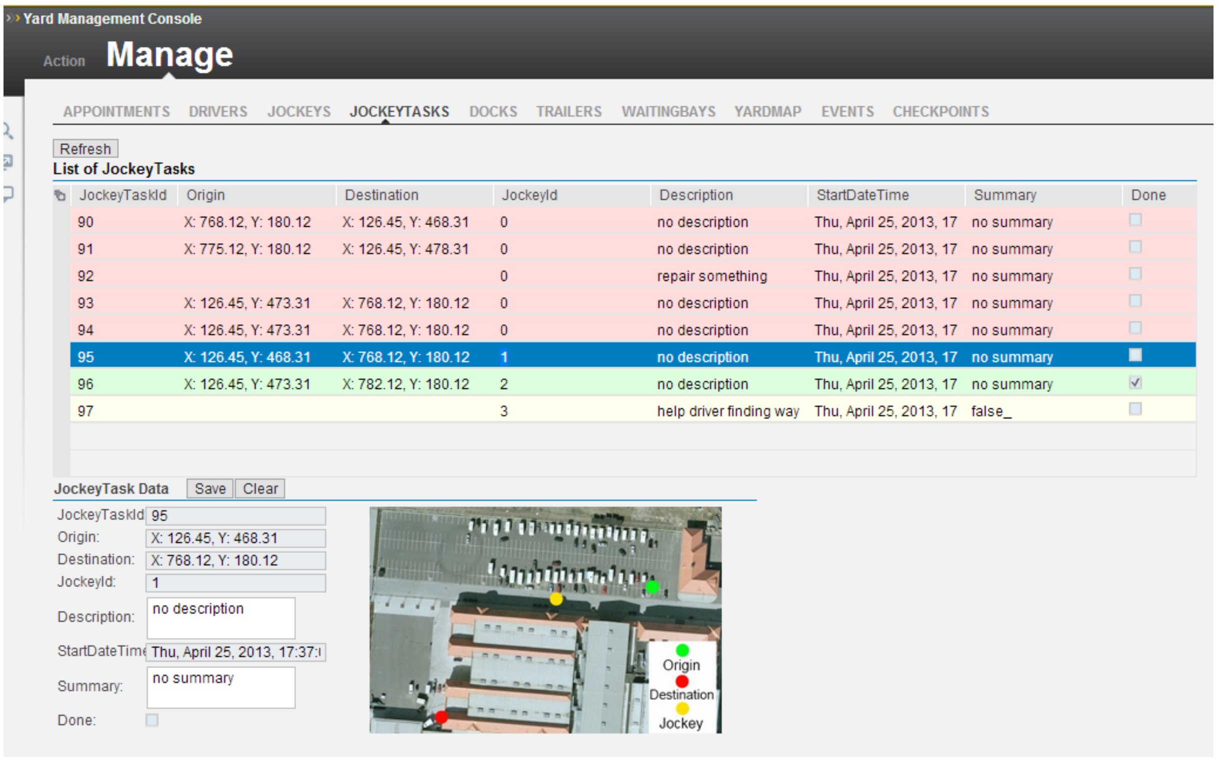
7.6.1. Example Yard Management Application

An example of a variable yard management application is applied to evaluate the applicability of the concepts of adaptive staged configuration workflows in an industrial project context. A variable yard management application, in the context of the *INDENICA*[19] project, serves as a basis. The INDENICA project is a European research project with the focus on variability in service-centric computing. The goal of the project is to abstract from service heterogeneity in service-oriented environments. Variability in functional and qualities of service platforms is observed to develop a virtual service platform abstracting from external services. A yard management application is developed on the virtual service platform as a case study for INDENICA. Figure 7.15 shows a screenshot of the yard management application prototypically implemented on the virtual service platform and integrated into the SAP HANA Cloud platform[20]. The SAP HANA Cloud platform abstracts from cloud infrastructure offering application and database services.

The application manages and controls logistics communication of arriving goods at a warehouse. The main functionality of the yard management prototype covers dock door scheduling and yard jockey support. Therefore, features for monitoring and coordinating the movement of vehicles in a yard, scheduling docks and staging areas for vehicles, and planning the jockey tasks to deliver or receive goods. Variation points are defined in the application to tailor the application according to customer requirements. For instance, yards vary in their capability of handling vehicles. Hence, the support for loading of ships and trains is optionally available. Furthermore, perishable freight requires a special refrigeration dock.

The variability of the application is modeled on an abstract level by means of an extended feature model. The feature model is visualized in EFeatureText notation in Listing B.5 in Appendix B.5. Stakeholders involved in the staged configuration workflow comprise an application provider responsible for configuring platform services, a number of resellers that pre-configure essential

[18]`https://github.com/extFM/extFM-Tooling/wiki/Dynamic-Staged-Configuration`
[19]`http://www.indenica.eu/`
[20]`http://www.sap.com/pc/tech/cloud/software/hana-cloud-platform-as-a-service/index.html`



List of JockeyTasks

JockeyTaskId	Origin	Destination	JockeyId	Description	StartDateTime	Summary	Done
90	X: 768.12, Y: 180.12	X: 126.45, Y: 468.31	0	no description	Thu, April 25, 2013, 17	no summary	☐
91	X: 775.12, Y: 180.12	X: 126.45, Y: 478.31	0	no description	Thu, April 25, 2013, 17	no summary	☐
92			0	repair something	Thu, April 25, 2013, 17	no summary	☐
93	X: 126.45, Y: 473.31	X: 768.12, Y: 180.12	0	no description	Thu, April 25, 2013, 17	no summary	☐
94	X: 126.45, Y: 473.31	X: 768.12, Y: 180.12	0	no description	Thu, April 25, 2013, 17	no summary	☐
95	X: 126.45, Y: 468.31	X: 768.12, Y: 180.12	1	no description	Thu, April 25, 2013, 17	no summary	☐
96	X: 126.45, Y: 473.31	X: 782.12, Y: 180.12	2	no description	Thu, April 25, 2013, 17	no summary	✓
97			3	help driver finding way	Thu, April 25, 2013, 17	false_	☐

JockeyTask Data

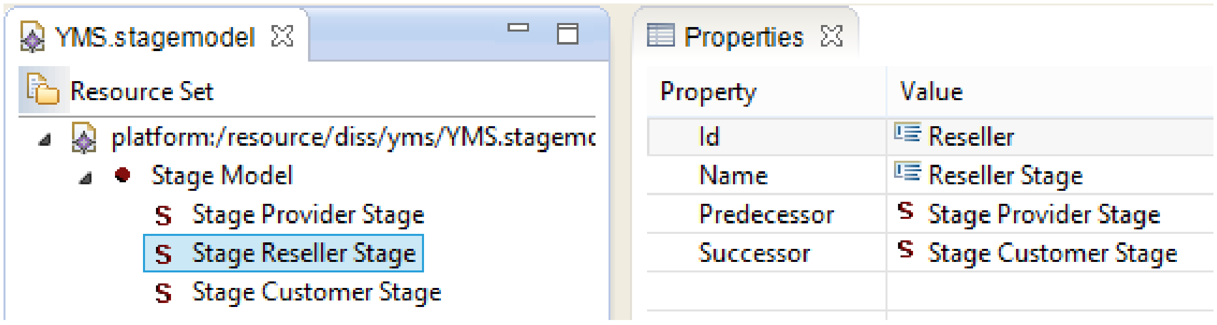
JockeyTaskId	95
Origin:	X: 126.45, Y: 468.31
Destination:	X: 768.12, Y: 180.12
JockeyId:	1
Description:	no description
StartDateTime	Thu, April 25, 2013, 17:37:
Summary:	no summary
Done:	☐

Figure 7.15 Screenshot of a variable yard management application.

yard management services, and customers renting the application from these resellers. Hence, three related stages are defined in a stage model, *Provider*, *Reseller*, and *Customer* as depicted in the screenshot in Figure 7.16. A corresponding stage metamodel is implemented in EMF to weave instances of this metamodel into the JWT workflow. The metamodel is depicted in Figure 7.17. The `StageModel` contains a set of `Stage`s. Each `Stage` is identified by an attribute `id`, which corresponds to the id of an abstract role in the role model. Hence, a stage is related to an abstract role by a naming convention. Furthermore, a relationship between successor and predecessor stages is defined to sequentialize stages.

The access of these stakeholders on configuration operations is restricted by applying RBAC, as explained in Section 6.4. An related access control model for the yard management system is defined in RBACText notation and depicted in Listing 7.36. The listing shows the initial access control model where only abstract stakeholder roles and their permissions are defined. Subsequently, concrete roles representing particular stakeholders are added and removed by applying rewrite rules. Appendix C.2 shows an example access control model after integrating further stakeholders by means of concrete roles and role groups.

In this example, the specialization tree definition starts with a minimal workflow comprising an initial node, a final node, and the control action to terminate the workflow, as depicted in Figure 7.18. All concrete stakeholders are integrated in the workflow by applying rewrite rules.



Resource Set
platform:/resource/diss/yms/YMS.stagemc
Stage Model
Stage Provider Stage
Stage Reseller Stage
Stage Customer Stage

Property	Value
Id	Reseller
Name	Reseller Stage
Predecessor	Stage Provider Stage
Successor	Stage Customer Stage

Figure 7.16 Stage model for the yard management system.

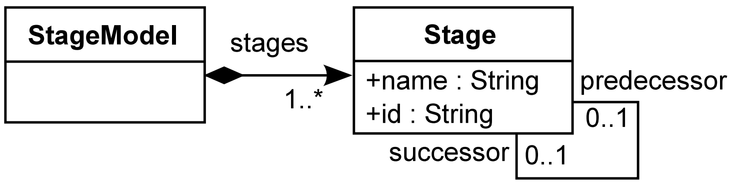


Figure 7.17 Stage Metamodel.

```
access control on <YMS.eft>

abstract role "Provider" <Provider> {
  select YMS,
    select Authentication, deselect Authentication,
    select Persistence, deselect Persistence,
    select Connectivity, deselect Connectivity,
  select JAAS, deselect JAAS,
    select JDBC, deselect JDBC,
    select JPA, deselect JPA,
    select RFC, deselect RFC,
    select SOAP, deselect SOAP,
    select REST, deselect REST
}

abstract role "Reseller" <Reseller> {
  select YM, deselect YM,
  select YJ, deselect YJ,
  select MC, deselect MC,
  select LS, deselect LS,
  assign YM.SchedulingType }

abstract role "Customer" <Customer> {
  select EnableShips, deselect EnableShips,
  select EnableTrains, deselect EnableTrains,
  select SpecialDocks, deselect SpecialDocks,
  select Coordinate, deselect Coordinate,
  select RoadMap, deselect RoadMap,
  select SatelliteMap, deselect SatelliteMap }
```

Listing 7.36 Access control model defining stakeholder roles and their permissions in the yard management system example.

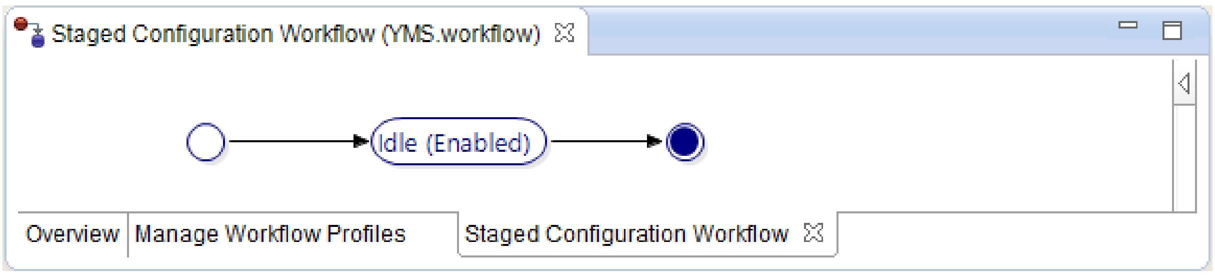
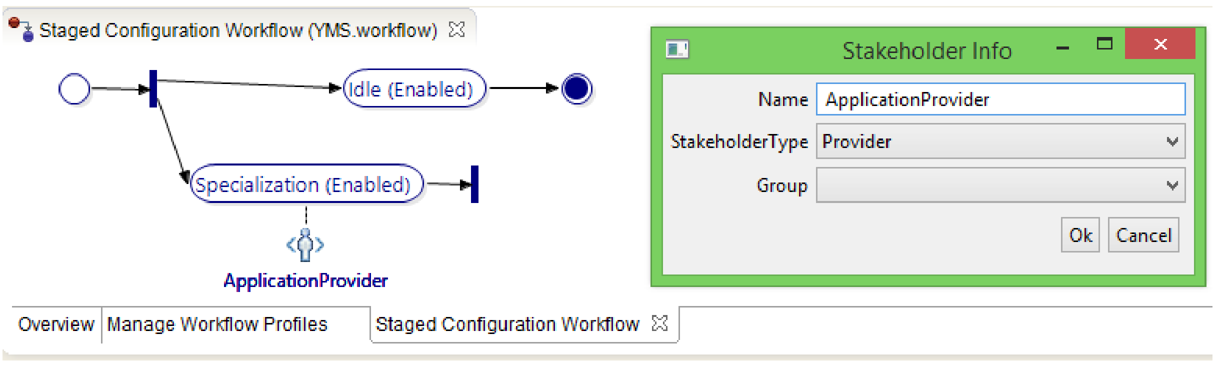

Staged Configuration Workflow (YMS.workflow)
Idle (Enabled)
Overview
Manage Workflow Profiles
Staged Configuration Workflow

Figure 7.18 Initial specialization tree of the yard management example.



Name	ApplicationProvider
StakeholderType	Provider
Group	

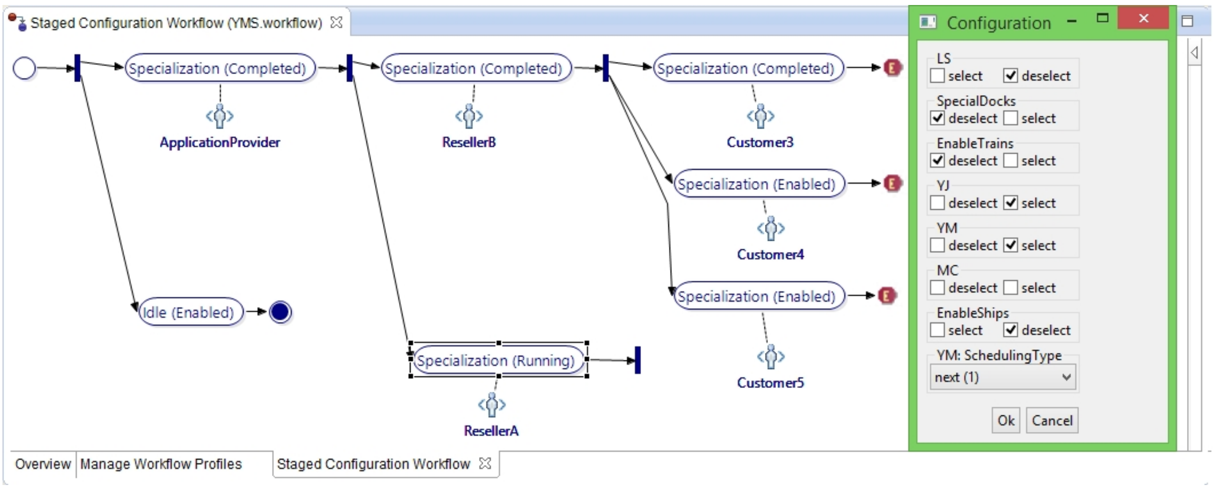
Figure 7.19 Specialization tree after integrating stakeholder *ApplicationProvider*.

After initializing the workflow, an *ApplicationProvider* is added to the workflow by specifying a name for the stakeholder and an abstract role, as depicted in Figure 7.19. The dialog for specifying the input parameters of a new stakeholder are shown on the right side, whereas the result of integrating the stakeholder are shown on the left side. As explained in Section 6.7.2, input parameters for integrating a new stakeholder are a name, an abstract role corresponding to a stage, and a predecessing concrete role. The *ApplicationProvider* is the first stakeholder that is integrated into the staged configuration workflow. Hence, only the abstract role *Provider* related to the first stage is selectable. Furthermore, no predecessing stakeholders in terms of concrete roles exist. Therefore the input parameter of a concrete role is omitted.

The selected abstract role *Provider* represents the stage in the specialization tree for adding the new specialization action related to the stakeholder. In addition, the permissions defined for the abstract role are inherited by the new concrete stakeholder role.

A *specialization tree* is modeled as a JWT workflow in a graphical notation similar to a UML activity diagram. DyscoGraph interprets the staged configuration workflow as a staged configuration workflow on an extended feature model referenced by the integrated access control model. A configuration workflow comprises potentially multiple specialization actions. An annotation depicts the current runtime state of an action according to the action lifecycle explained in Section 6.6.1. As the first stakeholder `ApplicationProvider` in this example did not start the configuration task yet, the status of this action is *enabled*.

A specialization action is depicted as a rectangle with rounded corners in the figure. Roles involved in configuration are modeled in an access control model. A specialization action is



Configuration

LS: select [✓ deselect]
SpecialDocks: [✓ deselect] select
EnableTrains: [✓ deselect] select
YJ: deselect [✓ select]
YM: deselect [✓ select]
MC: deselect select
EnableShips: select [✓ deselect]
YM: SchedulingType: next (1)

Ok Cancel

Figure 7.20 Execution of the specialization action of stakeholder *Reseller A* showing corresponding configuration view.

assigned to a stakeholder role defined in the referenced access control model. A role is visualized by a human-like icon. A role can execute a specialization action to perform a configuration task on the referenced feature model.

DyscoGraph provides a configuration view for a stakeholder according to the permissions defined for a role. These permissions specify a set of configuration operations. In the configuration view, the configuration operations a stakeholder is allowed to conduct, are represented by check boxes. Figure 7.20 depicts the configuration view of stakeholder *Reseller A* in the yard management example. The displayed configuration operations correspond to the permissions defined in the access control model for this stakeholder role.

As configuration operations on features are mutual exclusive, these operations are represented as a radio group, where a stakeholder can only select one of them. Configuration operations on a feature are *select(feature)* and *deselect(feature)*. Configuration on attributes are defined by attribute values which are *select(attributevalue)* and *deselect(attributevalue)*.

The configuration operations are evaluated if the corresponding partial configuration of the feature model is satisfiable. The *Ok* button is disabled, if the partial configuration is not satisfiable. As the partial configuration is evaluated after conducting a configuration operation, feedback about a valid configuration step is provided instantly to a configuration stakeholder.

The final stage in this example is the *Customer*, where concrete roles representing customers finish the configuration processes by deriving concrete variant configurations. A derived variant configuration is subsequently transformed into the variation representation applied in the solution space of the yard management application and persisted as XML file. This file is subsequently uploaded to the application server. The new configuration is instantly available in the yard management application.

The case study of a yard management system reveals that DyscoGraph is applicable to model and simulate adaptive staged reconfiguration processes that are introduced in Chapter 6.

7.7. Summary

The tool suite PUMA, introduced in this chapter, provides reference implementations for the concepts of multi-perspectives and adaptive staged reconfiguration workflows proposed in this thesis. Hence, PUMA implements the configuration management framework concepts related to the problem space.

The tool suite is developed applying an MDSD approach. Abstract syntaxes of structural models are defined by means of EMOF compliant metamodels applying the EMF framework. This framework implicitly specifies a concrete syntax for instantiating models in a tree-like graphical representation, which is further extended to provide a meaningful graphical visualization of the concepts. Textual DSLs are specified and implemented in the EMFText framework for extended feature models, access control models, and the mapping definition to create multi-perspectives.

The textual DSLs enable domain experts to define conceptual models rapidly providing a quick overview of the specified concepts. EFeatureText is a textual DSL for specifying group-cardinality based features models with attributes. RBACText is a textual DSL for defining access control on features and attributes of such feature models. MText is a textual DSL for defining the relation between features and viewgroups to derive filtered perspectives. Tooling is provided for these languages via a graphical user interface for the implemented DSLs. Various interfaces are defined to access and manipulate the models programmatically.

Beside the introduced languages and corresponding language tooling, PUMA comprises three tools to implement the configuration management concepts related to the problem space that are presented in the previous chapters. These tools are (i) a generic feature model analysis tool to analyze the satisfiability of an extended feature model, (ii) the tool *Conper* to specify concise perspectives and views on extended feature models, and (iii) *DyscoGraph* to model and execute adaptive staged configuration processes. The architecture of the tools and comprising functionalities are explained in this chapter. The tools exchange models and are combined in the integrated tool suite PUMA. However, the tools are loosely coupled and can be utilized stand alone.

Different case studies from academic and industrial context are implemented in the tools to evaluate the applicability of the proposed conceptual configuration management framework. Furthermore, the performance and scalability of the consistency algorithms proposed in the multi-perspective approach is measured on feature and view models of different sizes.

Summarizing, the tool suite PUMA is applicable to different scenarios, and provides interfaces for further extensions. The tool suite is implemented in Eclipse and available as open source.

8. Conclusion and Future Work

The best way to predict the future is to invent it.

— Alan Kay, 1971

This thesis provides an overview of the state-of-the-art of configurable cloud applications. The analysis of cloud applications leads to the following requirements. Configurable applications in the cloud demand for an automated configuration process and application provisioning to serve customers just-in-time. Customers have different concerns and therefore require pre-configured applications according to these concerns. As the demands of customers change, reconfiguration support is essential with respect to functional scalability of a cloud application. Furthermore, multiple different stakeholders are involved in the configuration and reconfiguration processes of cloud applications. However, in a cloud environment not all stakeholders participating in the configuration process are known before application runtime. In addition, a reconfigurable multi-tenant aware application architecture is required to provision customizations automatically.

A structured analysis of SPL engineering reveals that SPL methods are applicable to address these requirements. In SPL engineering, concepts related to the problem and solution space are distinguished. Concepts in the problem space abstract from implementation details in the solution space.

The SPL concepts of feature models and staged configuration processes are extended in this thesis to meet the identified requirements in the problem space. Feature models define the configuration space of an application by means of features and relations among them. Staged configuration is a method to derive a variant configuration from a feature model in multiple specialization steps, where each step may be related to a different stakeholder.

8.1. Contribution

The main contribution of this thesis is a feature-based configuration management framework for reconfigurable cloud applications introduced in Chapter 3. The framework comprises the concepts of multi-perspectives and adaptive staged reconfiguration workflows in the problem space, and a development method for reconfigurable multi-tenant aware applications based on a self-adaptive architecture in the solution space.

A formalization of multi-perspectives is provided in Chapter 5 to concisely define concern-specific pre-configurations of a feature model. A perspective is a concern-specific pre-configuration that satisfies feature model constraints. Each concern corresponds to a partial view on the feature model. Combining multiple views yields a perspective. Hence, a perspective comprises concern-specific configuration parameters, while unrelated parameters are concisely filtered.

Furthermore, an adaptive staged reconfiguration workflow is proposed in Chapter 6 to reduce configuration redundancies by automating multiple staged configuration processes in a single workflow by means of a specialization tree. Thus, multiple variant configurations are derivable from a specialization tree in contrast to conventional staged configuration workflows. This workflow design has two further advantages. First, concrete partial configurations serve as reusable pre-configurations. Second, reconfiguration decisions of stakeholders are propagated to depending stakeholders. In addition, combining a specialization tree with an adaptation engine enables a dynamic stakeholder management, as stakeholders can be integrated and removed from the workflow during execution.

Reference implementation of these concepts are provided by means of the tool suite PUMA to evaluate the applicability of these concepts in different case studies. The tool suite explained in Chapter 7 is developed following an MDSD approach and comprises the tools Conper, DyscoGraph, and FMAnalysis.

A self-adaptive architecture is applicable in the solution space to develop reconfigurable cloud applications. Hence, multi-tenant extensions to the self-adaptive architecture MQuAT are proposed in Chapter 4 to express variant configurations by means of functional and quality constraints. However, the proposed extensions require empirical evaluation.

8.2. Future Work

The development and implementation of a multi-tenant aware architecture for reconfigurable cloud application will be investigated in future studies. For instance, further multi-user capabilities of MQuAT are required. In addition, different architectural strategies for implementing reconfigurable multi-tenant aware applications are to be discussed in future work. Additional empirical evaluation of all concepts proposed in this theses on further industrial case studies would support the evidence of the industrial applicability of the configuration management framework.

Further research questions remain. In future work, a formalization of multi-perspectives on attributed feature models will be provided as multi-perspectives are formalized in this thesis for group-cardinality based feature models without attributes.

Furthermore, configuration states of features and attributes, discussed in Section 6.3.1, can be formalized by means of a process algebra. Such a formalization allows for model checking configuration states of features and attribute, as well as configuration operations in a staged configuration workflow.

In addition, the distribution of configuration decisions to stages may cause deadlocks in the configuration workflow. To obey lifeness and correctness of a configuration workflow, a translation of the logical dependencies between configuration decisions into a workflow Petri net is promising [LMSW13]. The translation of configuration decisions of extended feature models, as well as the postponing of configuration decisions require further research.

The concepts of multi-perspectives and adaptive staged configuration workflows are not restricted to cloud applications and therefore applicable to other domains. In future work, benefits are to be evaluated of applying these concepts to other configuration scenarios, such as multi-product lines.

Managing the information of feature and variant configurations is essential for quality assurance in SPL organizations. An interesting research issue is how to conduct quality assurance based on feature models of a cloud application [Gol13]. In an SPL organization not only the relations among features are of interest, but also comprehensive information about the current feature utilization and other feature and product related information. Such information is especially important in further developing the product line to react on advancements of the market. Based on this information statistics on feature utilization are derivable for instance to ensure the quality of the overall SPL by identifying favorite and rejected features. Hence, various SPL information need to be modeled, managed and restricted to specific responsible organizational stakeholders by providing adequate views [MS13].

Assuming cloud applications as agile SPLs is a further area for future research regarding application evolution and runtime updates. Applications in the cloud must quickly react on market demands requiring changes in functionality. Evolutionary changes require an open configuration space. The configuration framework proposed in this work assumes a closed configuration space by means of a domain feature model. The correctness of variant configurations is ensured with respect to dependencies defined in the feature model. However, to ensure the correct derivation of variant configurations in an open configuration space requires further research.

Summarizing, the proposed framework is a foundation for developing self-service portals and automating configuration and reconfiguration of cloud applications. Based on the introduced concepts, different research directions for future investigations can be identified.

Bibliography

[95/46/EC] European Union. Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. `http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:EN:HTML`, 1995. (Cited on page 17)

[ABGK02] Colin Atkinson, Christian Bunse, Hans-Gerhard Groß, and Thomas Kühne. Towards a General Component Model for Web-Based Applications. *Journal Annals of Software Engineering*, 13(1-4):35–69, June 2002. (Cited on page 74)

[ACF$^+$09] Mathieu Acher, Philippe Collet, Franck Fleurey, Philippe Lahire, and Jean-Paul Rigault. *Modeling Context and Dynamic Adaptations with Feature Models*, volume 9 of *MRT '09*, pages 89–98. CEUR, 2009. (Cited on page 105)

[ACLF11] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Slicing Feature Models. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, New York, NY, USA, 2011. ACM Press. (Cited on page 124)

[ACLF12] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Separation of Concerns in Feature Modeling: Support and Applications. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*, AOSD '12, New York, NY, USA, 2012. ACM Press. (Cited on pages 94 and 124)

[ACLF13] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming*, 78(6):657–681, 2013. (Cited on page 193)

[AFG$^+$09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report EECS-2009-28, EECS Department, University of California, Berkeley, 2009. (Cited on pages 15 and 24)

[AHH11] Ebrahim Khalil Abbasi, Arnaud Hubaux, and Patrick Heymans. A Toolset for Feature-Based Configuration Workflows. In *Proceedings of the 15th International*

Software Product Line Conference, SPLC '11, pages 65–69. IEEE, 2011. (Cited on page 45)

[Aik11] Larry Aiken. Why Multi-Tenancy is Key to Successful and Sustainable Software-as-a-Service (SaaS). *Cloudbook Journal*, 2, 2011. (Cited on page 19)

[AK09] Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. *The Journal of Object Technology*, 8(5):49–84, 2009. (Cited on page 32)

[APS+10] Andreas Abele, Yiannis Papadopoulos, David Servat, Martin Törngren, and Matthias Weber. The CVM Framework – A Prototype Tool for Compositional Variability Management. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '10, pages 101–105. Universität Duisburg-Essen, 2010. ICB-Research Report. (Cited on pages 193 and 194)

[Ass00] Uwe Assmann. Graph Rewrite Systems for Program Optimization. *ACM Transactions on Programming Languages and Systems*, 22(4):583–637, 2000. (Cited on page 169)

[Bab86] Wayne A. Babich. *Software Configuration Management: Coordination for Team Productivity.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. (Cited on page 51)

[Bat04] Don S. Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 702–703. IEEE Computer Society, 2004. (Cited on page 35)

[Bat05] Don S. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Software Product Line Conference*, SPLC '05, pages 7–20. Springer, 2005. (Cited on pages 34, 46, 86 and 115)

[BCW11] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled. In *Proceedings of the Third International Conference on Software Language Engineering*, SLE '10, pages 102–122, Berlin, Heidelberg, 2011. Springer-Verlag. (Cited on pages 193 and 194)

[Ben06] Messaoud Benantar. *Access Control Systems: Security, Identity Management and Trust Models.* Springer, 2006. (Cited on page 25)

[BFG+02] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J. Henk Obbink, and Klaus Pohl. Variability Issues in Software Product Lines. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE '01, pages 13–21. Springer-Verlag, 2002. (Cited on page 32)

[BFK$^+$99] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. PuLSE: A Methodology to Develop Software Product Lines. In *Proceedings of the 1999 Symposium on Software Reusability*, SSR '99, pages 122–131, New York, NY, USA, 1999. ACM. (Cited on pages 31 and 48)

[BGP12] Luciano Baresi, Sam Guinea, and Liliana Pasquale. Service-Oriented Dynamic Software Product Lines. *IEEE Computer*, 45(10):42–48, 2012. (Cited on page 48)

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999. (Cited on page 74)

[BKNT11] Christian Baun, Marcel Kunze, Jens Nimis, and Stefan Tai. *Cloud Computing: Web-basierte dynamische IT-Services*. Springer, second edition, 2011. (Cited on pages 10 and 11)

[BLB$^+$00] Keith H. Bennett, Paul J. Layzell, David Budgen, Pearl Brereton, Linda A. Macaulay, and Malcolm Munro. Service-Based Software: The Future for Flexible Software. In *Proceedings of the 7th Asia-Pacific Software Engineering Conference*, APSEC '00, pages 214–221. IEEE Computer Society, 2000. (Cited on page 11)

[Bos00] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. (Cited on page 34)

[BRCT05] David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. Using Constraint Programming to Reason on Feature Models. In *Proceedings of the Seventeenth International Conference on Software Engineering and Knowledge Engineering*, SEKE '05, 2005. (Cited on pages 45, 46, 200 and 201)

[BSBG08] 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 *Proceedings of the 12th International Conference on Software Product Lines*, SPLC '08, pages 23–32, 2008. (Cited on page 48)

[BSL$^+$10] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 73–82, New York, NY, USA, 2010. ACM. (Cited on page 37)

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years later: A Literature Review. *Journal of Information Systems*, 35:615–636, 2010. (Cited on pages 35, 37, 119 and 134)

[BSTRC05] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. Using Java CSP Solvers in the Automated Analyses of Feature Models. In *Proceedings of the Generative and Transformational Techniques in Software Engineering, International Summer School*, GTTSE '05, pages 399–408. Springer, 2005. (Cited on page 201)

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated Reasoning on Feature Models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, CAiSE '05, pages 491–503. Springer, 2005. (Cited on pages 38, 39, 46, 135 and 251)

[BZ10] Cor-Paul Bezemer and Andy Zaidman. Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 88–92, 2010. (Cited on pages 19 and 23)

[CA82] Timothy C. K. Chou and Jacob A. Abraham. Load Balancing in Distributed Systems. *Software Engineering, IEEE Transactions on*, SE-8(4):401–412, 1982. (Cited on page 20)

[CBH11] Andreas Classen, Quentin Boucher, and Patrick Heymans. A Text-Based Approach to Feature Modelling: Syntax and Semantics of TVL. *Science of Computer Programming*, 76(12):1130–1143, 2011. (Cited on pages 193 and 194)

[CBUE02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. Generative Programming for Embedded Software: An Industrial Experience Report. In *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 156–172. Springer Berlin Heidelberg, 2002. (Cited on page 38)

[CC06] Frederick Chong and Gianpaolo Carraro. Architecture Strategies for Catching the Long Tail. Online MSDN Article, 2006. `http://msdn.microsoft.com/en-us/library/aa479069.aspx`. (Cited on pages 22 and 74)

[CCW06] Frederick Chong, Gianpaolo Carraro, and Roger Wolter. Multi-Tenant Data Architecture. Online MSDN Article, 2006. `http://msdn.microsoft.com/en-us/library/aa479069.aspx`. (Cited on pages 19 and 22)

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000. (Cited on pages V, 32, 33, 34, 35, 37, 41, 78 and 99)

[Cha13] Mabel Joselin Brun Chaperon. Analyse von Konfigurationsmöglichkeiten mandantenfähiger SaaS-Anwendungen mittels erweiterter Feature-Modelle. Diploma thesis, Technische Universität Dresden, June 2013. (Cited on pages 23, 28 and 91)

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Using Feature Models. In *Proceedings of the Third International Software Product Line Conference*, SPLC '04, pages 266–283, 2004. (Cited on pages 5, 41, 42 and 121)

[CHE05a] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing Cardinality-Based Feature Models and Their Specialization. *Journal of Software Process: Improvement and Practice*, 10(1):7–29, 2005. (Cited on pages 94 and 124)

[CHE05b] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration through Specialization and Multi-Level Configuration of Feature Models. *Journal of Software Process: Improvement and Practice*, 10(2):143–169, 2005. (Cited on pages 42 and 45)

[CHH09] Andreas Classen, Arnaud Hubaux, and Patrick Heymans. A Formal Semantics for Multi-Level Staged Configuration. In *Proceedings of VaMoS '09*, 2009. (Cited on pages 42, 94 and 121)

[CHS08] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a Feature: A Requirements Engineering Perspective. In *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering*, FASE '08/ETAPS '08, pages 16–30, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on page 32)

[CMRD13] Kyle Christensen, Michael Moaz, Ed Romson, and Alexandre Dayon. 10 Key Strategies Customer Service Executives Need To Consider Featuring Gartner. Webinar, 2013. `https://www.salesforce.com/form/event/10-key-css-strategies.jsp`. (Cited on page 17)

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. (Cited on pages 31 and 32)

[Coh90] Jacques Cohen. Constraint Logic Programming Languages. *Communications of the ACM*, 33(7):52–68, July 1990. (Cited on page 39)

[Coh03] Sholom Cohen. Predicting When Product Line Investment Pays. Technical Report CMU/SEI-2003-TN-017, Software Engineering Institute, 2003. (Cited on page 29)

[CP10] Dave Clarke and José Proença. Towards a Theory of Views for Feature Models. In *Proceedings of FMSPLE '10*, 2010. (Cited on page 124)

[CS10] Ronni J. Colville and George Spafford. Top Seven Considerations for Configuration Management for Virtual and Cloud Infrastructures. Technical Report Gartner RAS Core Research Note G00208328, Gartner, 2010. (Cited on page 2)

[CW98] Reidar Conradi and Bernhard Westfechtel. Version Models for Software Configuration Management. *Journal of ACM Computing Surveys*, 30(2):232–282, June 1998. (Cited on page 51)

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature Diagrams and Logics: There and Back Again. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 23–34. IEEE Computer Society, 2007. (Cited on page 36)

[Dav87] Stanley M. Davis. *Future Perfect: A Startling Vision of the Future We Should Be Managing Now.* Addison-Wesley Longman, Incorporated, 1987. (Cited on page 32)

[DDF$^+$06] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A Survey of Autonomic Communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2):223–259, December 2006. (Cited on pages 78 and 81)

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Inc., 1976. (Cited on page 94)

[DPAG11] Jessica Díaz, Jennifer Pérez, Pedro P. Alarcón, and Juan Garbajosa. Agile Product Line Engineering – A Systematic Literature Review. *Journal for Software-Practice & Experience*, 41(8):921–941, 2011. (Cited on page 34)

[DtH01] Marlon Dumas and Arthur ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. *≪UML≫ 2001 The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 76–90, 2001. (Cited on page 45)

[EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jörg. Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation Volume 2: Applications, Languages and Tools.* World Scientific Publishing Co., Inc., 1999. (Cited on page 160)

[EIA-649-B] American National Standards Institute. ANSI/EIA-649-B, Configuration Management Standard. `http://webstore.ansi.org/RecordDetail.aspx?sku=EIA-649-B`, 2011. (Cited on page 50)

[Els12] Christoph Elsner. *Automating Staged Product Derivation for Heterogeneous Multi-Product-Lines.* PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2012. (Cited on page 177)

[ES13] Holger Eichelberger and Klaus Schmid. A Systematic Analysis of Textual Variability Modeling Languages. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 12–21. ACM, 2013. (Cited on page 193)

[ESSPL10] Christoph Elsner, Christa Schwanninger, Wolgang Schröder-Preikschat, and Daniel Lohmann. Multi-Level Product Line Customization. In *Proceedings of the 2010 conference on New Trends in Software Methodologies, Tools and Techniques*, SoMeT '10, pages 37–58, Amsterdam, The Netherlands, The Netherlands, 2010. IOS Press. (Cited on page 119)

[Fah08] Dirk Fahland. Translating UML2 Activity Diagrams to Petri Nets for Analyzing IBM WebSphere Business Modeler Process Models. Informatik-Berichte 226, Humboldt-Universität zu Berlin, 2008. (Cited on page 45)

[FE10] Borko Furht and Armando Escalante, editors. *Handbook of Cloud Computing.* Springer, 2010. (Cited on page 14)

[FK92] David F. Ferraiolo and D. Richard Kuhn. Role Based Access Control. In *Proceedings of the 15th National Computer Security Conference*, NCSC '92, pages 554–563, 1992. (Cited on page 26)

[FKC07] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control.* Artech House, Inc., Norwood, MA, USA, second edition, 2007. (Cited on page 25)

[FKN$^+$92] Anthony Finkelstein, Jeff Kramer, Bashar Nuseibeh, Larry Finkelstein, and Michael Goedicke. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, 1992. (Cited on page 94)

[FS09] Franck Fleurey and Arnor Solberg. A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 606–621. Springer, 2009. (Cited on page 89)

[GAB$^+$13] Frank Gens, Margaret Adam, David Bradshaw, Christian A. Christiansen, Laura DuBois, Alejandro Florean, Phil Hochmuth, Vladimír Kroa, Robert P. Mahowald, Satoshi Matsumoto, Chris Morris, Tony Olvet, Kelly Quinn, Mary Johnston Turner, Richard L. Villars, and Melanie Posey. Worldwide and Regional Public IT Cloud Services 2013–2017 Forecast. Technical Report 242464, International Data Corporation (IDC), 2013. (Cited on page 1)

[GFd98] Martin Lo Griss, John Favaro, and Massimo d'Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse*, ICSR '98, pages 76–85, 1998. (Cited on page 33)

[GH04] Hassan Gomaa and Mohamed Hussein. Dynamic Software Reconfiguration in Software Product Families. In *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 435–444. Springer Berlin Heidelberg, 2004. (Cited on page 47)

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. (Cited on page 179)

[GKR$^+$06] Kurt Geihs, Mohammad Ullah Khan, Roland Reichle, Arnor Solberg, Svein Hallsteinsen, and Simon Merral. Modeling of Component-Based Adaptive Distributed Applications. In *Proceedings of the ACM Symposium on Applied Computing*, SAC '06, pages 718–722. ACM, 2006. (Cited on pages 75 and 90)

[GLR$^+$02] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The Missing Link of MDA. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105. Springer Berlin Heidelberg, 2002. (Cited on page 160)

[Gol13] David Gollasch. Qualitätssicherung mittels Feature-Modellen. Bachelor thesis, Technische Universität Dresden, September 2013. (Cited on pages 202 and 225)

[Göt13] Sebastian Götz. *Multi-Quality Auto-Tuning by Contract Negotiation.* Dissertation, Technische Universität Dresden, Dresden, Germany, 2013. (Cited on pages 78, 79 and 82)

[GP92] Thomas R. G. Green and Marian Petre. When Visual Programs are Harder to Read than Textual Programs. In *Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics)*, Human-Computer Interaction: Tasks and Organisation, pages 167–180, 1992. (Cited on page 184)

[Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Addison-Wesley Professional, first edition, 2009. (Cited on page 183)

[GS03] Jack Greenfield and Keith Short. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 16–27, New York, NY, USA, 2003. ACM. (Cited on page 35)

[GTL00] Murali Sitaraman Gary T. Leavens. *Foundations of Component-Based Systems.* Cambridge University Press, 2000. (Cited on page 74)

[GWCA11] Sebastian Götz, Claas Wilke, Sebastian Cech, and Uwe Aßmann. *Sustainable Green Computing: Practices, Methodologies and Technologies*, chapter Architecture and Mechanisms for Energy Auto Tuning. IGI Global, 2011. (Cited on page 78)

[GWS$^+$10] Sebastian Götz, Claas Wilke, Matthias Schmidt, Sebastian Cech, and Uwe Aßmann. Towards Energy Auto Tuning. In *Proceedings of the First Annual*

International Conference on Green Information Technology, GREEN IT '10, 2010. (Cited on pages V, 77 and 78)

[HCH09] Arnaud Hubaux, Andreas Classen, and Patrick Heymans. Formal Modelling of Feature Configuration Workflows. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 221–230. ACM, 2009. (Cited on page 45)

[HGS+11] Waldemar Hummer, Patrick Gaubatz, Mark Strembeck, Uwe Zdun, and Schahram Dustdar. An Integrated Approach for Identity and Access Management in a SOA Context. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, SACMAT '11, pages 21–30, New York, NY, USA, 2011. ACM. (Cited on page 199)

[HHPS08] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, 2008. (Cited on page 47)

[HHS+11] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Dirk Deridder, and Ebrahim Abbasi. Supporting Multiple Perspectives in Feature-Based Configuration. *Software and Systems Modeling*, 10:1–23, 2011. 10.1007/s10270-011-0220-1. (Cited on pages 94 and 124)

[HJK+09] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin Heidelberg, 2009. (Cited on page 183)

[HJK+13] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Model-Based Language Engineering with EMFText. In *Generative and Transformational Techniques in Software Engineering IV*, volume 7680 of *Lecture Notes in Computer Science*, pages 322–345. Springer Berlin Heidelberg, 2013. (Cited on page 183)

[HL93] Hans-Joachim Habermann and Frank Leymann. *Repository: Eine Einführung.* Oldenbourg Verlag München, Wien, first edition, 1993. (Cited on page 180)

[HM05] John K. Halvey and Barbara M. Melby. *Information Technology Outsourcing Transactions: Process, Strategies, and Contracts.* John Wiley & Sons, 2005. (Cited on page 14)

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, October 2004. (Cited on page 180)

[HSS+10] Florian Heidenreich, Pablo Sanchez, Joao Santos, Steffen Zschaler, Mauricio Alferez, Joao Araujo, Lidia Fuentes, Uira Kulesza, Ana Moreira, and Awais Rashid. Relating Feature Models to Other Models of a Software Product

Line: A Comparative Study of FeatureMapper and VML*. *Transactions on Aspect-Oriented Software Development VII*, 2010. (Cited on page 209)

[HST$^+$08] Patrick Heymans, Pierre-Yves Schobbens, Jean-Christophe Trigaux, Yves Bontemps, Raimundas Matulevicius, and Andreas Classen. Evaluating Formal Properties of Feature Diagram Languages. *IET Software*, 2008. (Cited on pages 101 and 102)

[Hub12] Arnaud Hubaux. *Feature-based Configuration: Collaborative, Dependable, and Controlled.* PhD thesis, University of Namur, Belgium, 2012. (Cited on pages 44, 121 and 124)

[Hur10] Hurwitz & Associates. The Sources of Web Application Downtime. Technical report, Phurnace Software, 2010. (Cited on page 2)

[HW07] Florian Heidenreich and Christian Wende. Bridging the Gap Between Features and Models. In *Proceedings of the Second Workshop on Aspect-Oriented Product Line Engineering*, AOPLE '07, 2007. (Cited on page 61)

[HWC12] Øystein Haugen, Andrzej Wasowski, and Krzysztof Czarnecki. CVL: Common Variability Language. In *Proceedings of the 16th International Software Product Line Conference*, SPLC '12, pages 266–267. ACM, 2012. (Cited on page 34)

[IDC09] International Data Corporation. Defining "Cloud Services" – an IDC update. `http://blogs.idc.com/ie/?p=422`, 2009. retrieved on 2013/05/22. (Cited on page 16)

[IEEE 828-2012] IEEE Computer Society. IEEE 828-2012 - IEEE Standard for Configuration Management in Systems and Software Engineering. `http://standards.ieee.org/findstds/standard/828-2012.html`, 2012. (Cited on page 51)

[INCITS 359] American National Standards Institute. ANSI INCITS 359-2004, Role Based Access Control. `http://www.profsandhu.com/journals/tissec/ANSI+INCITS+359-2004.pdf`, 2004. (Cited on page 26)

[ISO 10006] International Organization for Standardization. ISO 10006:2003, Quality Management Systems – Guidelines for Quality Management in Projects. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=36643`, 2003. (Cited on pages 23 and 24)

[ISO 42010] International Organization for Standardization. ISO/IEC/IEEE 42010:2011, Systems and Software Engineering Standard. `http://www.iso-architecture.org/ieee-1471/`, 2011. (Cited on pages 72, 73 and 94)

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006. (Cited on page 129)

[Jam12] Kris Jamsa. *Cloud Computing: SaaS, PaaS, IaaS, Virtualization, Business Models, Mobile, Security, and More.* Jones & Bartlett Pub, 2012. (Cited on pages IX and 11)

[JOP11] Venkata Josyula, Malcolm Orr, and Greg Page. *Cloud Computing: Automating the Virtualized Data Center*, chapter Service Life Cycle Management, page 256. Cisco Press, 2011. (Cited on pages 14 and 15)

[KCH$^+$90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University Pittsburgh, Software Engineering Institute, 1990. (Cited on pages 31, 34, 35 and 37)

[KK02] Dimitris Karagiannis and Harald Kühn. Metamodelling Platforms. In *E-Commerce and Web Technologies*, volume 2455 of *Lecture Notes in Computer Science*, pages 182–182. Springer Berlin Heidelberg, 2002. (Cited on page 181)

[KKL$^+$98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5(1):143–168, 1998. (Cited on page 61)

[KL03] Alexander Keller and Heiko Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11:57–81, 2003. (Cited on page 18)

[KOD10] Ahmet Serkan Karataş, Halit Oğuztüzün, and Ali H. Doğru. Mapping Extended Feature Models to Constraint Logic Programming over Finite Domains. In *Proceedings of the 14th International Software Product Line Conference*, SPLC '10, pages 286–299, 2010. (Cited on pages 45, 46, 132, 200, 201 and 251)

[Koz11] Heiko Koziolek. The SPOSAD Architectural Style for Multi-tenant Software Applications. In *Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture*, WICSA '11, pages 320–327, 2011. (Cited on page 89)

[KPMG13] KPMG AG and Bitkom. Cloud-Monitor 2013 - Cloud-Computing in Deutschland – Status quo und Perspektiven. `http://www.bitkom.org/files/documents/Studie_Cloud_Monitor.pdf`, 2013. (Cited on page 9)

[Kru13] Charles W. Krueger. Multistage Configuration Trees for Managing Product Family Trees. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 188–197. ACM Press, 2013. (Cited on pages 86, 121 and 177)

[Lad99] Robert Laddaga. Guest Editor's Introduction: Creating Robust Software through Self-Adaptation. *IEEE Intelligent Systems*, 14(3):0026–29, 1999. (Cited on page 74)

[LK06] Jaejoon Lee and Kyo C. Kang. A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering. In *Proceedings of the 10th International Software Product Line Conference*, SPLC '06, pages 140–150, 2006. (Cited on pages 48 and 177)

[LKL02] Kwanwoo Lee, Kyo Chul Kang, and Jaejoon Lee. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*, ICSR-7, pages 62–77, London, UK, UK, 2002. Springer-Verlag. (Cited on page 35)

[LMSW13] Malte Lochau, Stephan Mennicke, Julia Schroeter, and Tim Winkelmann. Extended Version of Automated Verification of Feature Model Configuration Processes based on Workflow Petri Nets. Technical report, TU Braunschweig, 2013. (Cited on pages 4, 127, 135 and 225)

[LP07] Felix Loesch and Erhard Ploedereder. Optimization of Variability in Software Product Lines. In *Proceedings of the 11th International Software Product Line Conference*, pages 151–162, 2007. (Cited on page 37)

[LSR07] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. (Cited on page 31)

[LSZ12] Ioanna Lytra, Stefan Sobernig, and Uwe Zdun. Architectural Decision Making for Service-Based Platform Integration: A Qualitative Multi-Method Study. In *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, WICSA/ECSA '12, pages 111–120. IEEE, 2012. (Cited on page 176)

[MA02] Dirk Muthig and Colin Atkinson. Model-Driven Product Line Architectures. In *Proceedings of the Second International Software Product Line Conference*, SPLC '02, 2002. (Cited on page 29)

[Ma07] Dan Ma. The Business Model of "Software-as-a-Service". In *Proceedings of the IEEE International Conference on Services Computing*, SCC '07, pages 701–702, Los Alamitos, CA, USA, 2007. IEEE Computer Society. (Cited on page 11)

[Man02] Mike Mannion. Using First-Order Logic for Product Line Model Validation. In *Software Product Lines*, volume 2379 of *Lecture Notes in Computer Science*, pages 176–187. Springer Berlin Heidelberg, 2002. (Cited on page 46)

[MBJ$^+$09] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Models@Run.time to Support Dynamic Adaptation. *Computer*, 42:44–51, 2009. (Cited on page 75)

[MBNJ09] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jézéquel. Taming Dynamically Adaptive Systems Using Models and Aspects. In *Proceedings of the International Conference on Software Engineering*, ICSE '09, pages 122–132. IEEE Computer Society, 2009. (Cited on page 89)

[MCdO07] Marcílio Mendonça, Donald D. Cowan, and Toacy Cavalcante de Oliveira. A Process-Centric Approach for Coordinating Product Configuration Decisions. In *Proceedings of the 40th Hawaii International International Conference on Systems Science*, HICSS '07, page 283. IEEE Computer Society, 2007. (Cited on pages 44 and 45)

[McI68] Malcolm Douglas McIlroy. Mass-Produced Software Components. In *Proceedings of Software Engineering Concepts and Techniques, 1968 NATO Conference on Software Engineering*, pages 88–98, 1968. (Cited on pages 32 and 77)

[MdlILG08] Jose Luis Marín de la Iglesia and Jose Emilio Labra Gayo. Doing Business by Selling Free Services. In *Web 2.0*, pages 1–14. Springer US, 2008. (Cited on page 91)

[Mee11] Stephanie Meerkamm. Configuration of Multi-Perspectives Variants. In *Business Process Management Workshops*, volume 66 of *Lecture Notes in Business Information Processing*, pages 277–288. Springer Berlin Heidelberg, 2011. (Cited on page 124)

[MG11] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. NIST Special Publication 800-145, National Institute of Standards and Technology, Information Technology Laboratory, 2011. (Cited on pages 10, 12, 14 and 15)

[MH10] Mario Meir-Huber. *Cloud Computing: Praxisratgeber und Einstiegsstrategien.* Entwickler.Press, 2010. (Cited on page 13)

[Mie10] Ralph Mietzner. *A Method and Implementation to Define and Provision Variable Composite Applications, and its Usage in Cloud Computing.* PhD thesis, Universität Stuttgart, 2010. (Cited on pages 29, 67, 68 and 89)

[MS98] Kimbal Marriott and Peter J. Stuckey. *Programming With Constraints: An Introduction.* MIT Press, 1998. (Cited on page 38)

[MS13] Dirk Muthig and Julia Schroeter. A Framework for Role-Based Feature Management in Software Product Line Organizations. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, New York, NY, USA, 2013. ACM Press. (Cited on pages 4, 127, 137 and 225)

[MSD$^+$12] Raúl Mazo, Camille Salinesi, Daniel Diaz, Olfa Djebbi, and Alberto Lora-Michiels. Constraints: The Heart of Domain and Application Engineering in the Product Lines Engineering Strategy. *International Journal of Information System Modeling and Design*, 3(2):33–68, April 2012. (Cited on pages 46 and 157)

[MSDLM11] Raúl Mazo, Camille Salinesi, Daniel Diaz, and Alberto Lora-Michiels. Transforming Attribute and Clone-Enabled Feature Models into Constraint Programs over Finite Domains. In *Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE '11)*, pages 188–199. SciTePress, 2011. (Cited on pages 45, 46, 200, 201 and 251)

[MWC09] Marcílio Mendonça, Andrzej Wasowski, 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, 2009. (Cited on page 46)

[NCB$^+$13] Linda M. Northrop, Paul C. Clements, Felix Bachmann, John Bergey, Gary Chastek, Sholom Cohen, Patrick Donohoe, Lawrence Jones, Robert Krut, Reed Little, John McGregor, and Liam O'Brien. A Framework for Software Product Line Practice, Version 5.0. Online Article, 2013. `http://www.sei.cmu.edu/productlines/framework.html`. (Cited on page 51)

[Nei80] James M. Neighbors. *Software Construction Using Components.* PhD thesis, Department of Information and Computer Science, University of California, Irvine, 1980. (Cited on page 33)

[Nei84] James M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *Software Engineering, IEEE Transactions on*, SE-10(5):564–574, sept. 1984. (Cited on page 180)

[NH02] Thomas T. Nagle and Reed K. Holden. *The Strategy and Tactics of Pricing.* Prentice Hall, 2002. (Cited on page 91)

[Nit09] Nitu. Configurability in SaaS (Software as a Service) Applications. In *Proceedings of the 2nd India software engineering conference*, ISEC '09, pages 19–26, New York, NY, USA, 2009. ACM. (Cited on page 18)

[NKF03] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. Viewpoints: Meaningful Relationships are Difficult. In *Proceedings of the International Conference on Software Engineering*, ICSE '03, 2003. (Cited on page 94)

[Obj03] Object Management Group. *MDA Guide Version 1.0.1*, 2003. (Cited on page 181)

[OMG2004] UML Human-Usable Textual Notation (HUTN), version 1.0, 2004. (Cited on page 183)

[OMG2011a] MOF 2 XMI Mapping, version 2.4.1, 2011. (Cited on page 183)

[OMG2011b] OMG Unified Modeling Language (OMG UML) Superstructure, version 2.4.1, 2011. (Cited on pages 146 and 181)

[OMG2012] OMG Object Constraint Language (OCL), version 2.3.1, 2012. (Cited on page 86)

[Par66] Douglas Parkhill. *The Challenge of the Computer Utility.* Addison-Wesley Publishing Company, 1966. (Cited on page 10)

[Par72] David L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972. (Cited on page 94)

[Par76] David L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, 2:1–9, 1976. (Cited on page 32)

[Par13] Beth Pariseau. Self-Service Portals Give IT Control in the Cloud. Web Journal SearchCloudComputing.com, May 2013. (Cited on page 27)

[PBN$^+$11] Leonardo Teixeira Passos, Thorsten Berger, Marko Novakovic, Krzysztof Czarnecki, Yingfei Xiong, and Andrzej Wasowski. A Study of Non-Boolean Constraints in Variability Models of an Embedded Operating System. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, pages 2:1–2:8, 2011. (Cited on pages 132, 200 and 201)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques.* Springer, 2005. (Cited on pages 4, 31, 32, 33, 34, 73 and 124)

[Pil06] Frank T. Piller. *Mass Customization.* Gabler Edition Wissenschaft. Dt. Univ.-Verl., Wiesbaden, fourth edition, 2006. (Cited on page 31)

[RA11] 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*, pages 16:1–16:4. ACM, 2011. (Cited on pages 67 and 68)

[RBP$^+$91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object Oriented Modeling and Design.* Prentice-Hall, 1991. (Cited on page 27)

[RBSP02] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending Feature Diagrams With UML Multiplicities. In *Proceedings of 6th Conference on Integrated Design & Process Technology*, IDPT '02, Pasadena, California, USA, 2002. (Cited on pages IX and 38)

[RN09] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Upper Saddle River, NJ, USA, third edition, 2009. (Cited on pages 39 and 45)

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation Volume 1: Foundations.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. (Cited on page 160)

[RSPA11a] Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. Combining Runtime Adaptation and Static Binding in Dynamic Software Product Lines. Technical Report 02, School of Computer Science, University of Magdeburg, February 2011. (Cited on page 42)

[RSPA11b] Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. Tailoring Dynamic Software Product Lines. In *Proceedings of the 10th International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 3–12, New York, NY, USA, 2011. ACM Press. (Cited on page 124)

[RW08] Ulrike Ranger and Erhard Weinell. The Graph Rewriting Language and Environment PROGRES. In *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science*, pages 575–576. Springer Berlin Heidelberg, 2008. (Cited on page 163)

[RW12] Manfred Reichert and Barbara Weber. *Enabling Flexibility in Process-Aware Information Systems.* Springer, 2012. (Cited on pages VI and 149)

[SAP09] *SAP BusinessObjects User Management System Administrator's Guide*, 2009. (Cited on page 26)

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. *Software Product Lines: Going Beyond*, pages 77–91, 2010. (Cited on page 35)

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, second edition, 2009. (Cited on page 181)

[SCFY96] Ravi Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. In *IEEE Computer*, volume 29, pages 38–47. IEEE Press, 1996. (Cited on page 26)

[SCG+12] Julia Schroeter, Sebastian Cech, Sebastian Götz, Claas Wilke, and Uwe Aßmann. Towards Modeling a Variable Architecture for Multi-Tenant SaaS-Applications. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 111–120, New York, NY, USA, January 2012. ACM Press. (Cited on pages 4, 71, 72 and 83)

[Sch11] Julia Schroeter. Towards Generating Multi-Tenant Applications. In *Proceedings of the GTTSE and SLE 2011 Students' Workshop*, July 2011. (Cited on pages 4 and 127)

[SDK+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, 1995. (Cited on page 73)

[SDM+11] Camille Salinesi, Olfa Djebbi, Raúl Mazo, Daniel Diaz, and Alberto Lora-Michiels. Constraints: The Core of Product Line Engineering. In *Proceedings of the Fifth IEEE International Conference on Research Challenges in Information Science)*, RCIS '11, pages 1–10. IEEE, 2011. (Cited on page 46)

[Seg08] Sergio Segura. Automated Analysis of Feature Models Using Atomic Sets. In *Proceedings of the 12th International Conference of Software Product Lines*, SPLC '08, pages 201–207. Lero Int. Science Centre, University of Limerick, Ireland, 2008. (Cited on page 46)

[SFK00] Ravi Sandhu, David F. Ferraiolo, and D. Richard Kuhn. The NIST Model for Role Based Access Control: Toward a Unified Standard. In *Proceedings of the 5th ACM Workshop on Role Based Access Control*. ACM Press, 2000. (Cited on pages 26 and 27)

[SJ04] Klaus Schmid and Isabel John. A Customizable Approach to Full Lifecycle Variability Management. *Science of Computer Programming - Special issue: Software variability management*, 53:259–284, 2004. (Cited on page 34)

[SK95] Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, first edition, 1995. (Cited on page 180)

[SLW11] Julia Schroeter, Malte Lochau, and Tim Winkelmann. Extended Version of Multi-Perspectives on Feature Models. In *Technical Report TUD-FI11-07-Dezember 2011*. Technische Universität Dresden, December 2011. (Cited on pages 4, 91 and 119)

[SLW12a] Julia Schroeter, Malte Lochau, and Tim Winkelmann. Conper: Consistent Perspectives on Feature Models. In *Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications*, ECMFA '12, pages 55–58. Technical University of Denmark DTU, July 2012. (Cited on pages 4, 91 and 202)

[SLW12b] Julia Schroeter, Malte Lochau, and Tim Winkelmann. Multi-Perspectives on Feature Models. In *Model Driven Engineering Languages and Systems - Proceedings of the 15th International Conference on Model Driven Engineering*

Languages & Systems, volume 7590 of *MODELS '12*, pages 252–268. Springer Berlin Heidelberg, October 2012. (Cited on pages 4, 91, 92 and 103)

[SMM+12] 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, September 2012. ACM. (Cited on pages 4, 127, 128, 130 and 144)

[Smo02] Kari Smolander. Four Metaphors of Architecture in Software Organizations: Finding out the Meaning of Architecture in Practice. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 211–221, 2002. (Cited on page 72)

[SOS+12] Karsten Saller, Sebastian Oster, Andy Schürr, Julia Schroeter, and Malte Lochau. Reducing Feature Models to Improve Runtime Adaptivity on Resource Limited Devices. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 135–142, New York, NY, USA, September 2012. ACM Press. (Cited on pages 93 and 125)

[SPLOT12] Software Product Line Online Tools (SPLOT). Project Website `http://www.splot-research.org`, April 2012. (Cited on page 202)

[SRP03] Detlef Streitferdt, Matthias Riebisch, and Ilka Philippow. Details of Formalized Relations in Feature Models Using OCL. In *Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the*, pages 297–304, 2003. (Cited on pages 38 and 39)

[SSSPS07] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is The Linux Kernel a Software Product Line? In *Proceedings of the International Workshop on Open Source Software and Product Lines*, SPLC-OSSPL '07, Kyoto, Japan, 2007. (Cited on page 29)

[STB+04] Mirjam Steger, Christian Tischer, Birgit Boss, Andreas Müller, Oliver Pertler, Wolfgang Stolz, and Stefan Ferber. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In *Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 34–50. Springer Berlin Heidelberg, 2004. (Cited on page 37)

[SV01] Pierangela Samarati and Sabrina De Capitani di Vimercati. Access Control: Policies, Models, and Mechanisms. In *Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design: Tutorial Lectures*, FOSAD '00, pages 137–196, London, UK, UK, 2001. Springer-Verlag. (Cited on page 25)

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006. (Cited on page 129)

[SZG+08] Wei Sun, Xin Zhang, Chang Jie Guo, Pei Sun, and Hui Su. Software as a Service: Configuration and Customization Perspectives. In *Congress on Services Part II, 2008. SERVICES-2. IEEE*, pages 18–25, 2008. (Cited on page 18)

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman, Amsterdam, 1998. (Cited on page 76)

[TBD+08] Pablo Trinidad, David Benavides, Amador Durán, Antonio Ruiz-Cortés, and Miguel Toro. Automated Error Analysis for the Agilization of Feature Modeling. *Journal of Systems and Software*, 81(6):883 – 896, 2008. (Cited on pages 46 and 120)

[TCPB07] Pablo Trinidad, Antonio Ruiz Cortés, Joaquín Peña, and David Benavides. Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 51–56. IEEE Computer Society, 2007. (Cited on page 61)

[TH02] Steffen Thiel and Andreas Hein. Modelling and Using Product Line Variability in Automotive Systems. *IEEE Software*, 19(4):66–72, 2002. (Cited on page 29)

[Tha12] Cheng Thao. *A Configuration Management System for Software Product Lines.* PhD thesis, University of Wisconsin-Milwaukee, 2012. (Cited on page 51)

[TKES11] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In *Proceedings of the 15th International Conference of Software Product Lines*, SPLC '11, pages 191–200. IEEE, 2011. (Cited on page 120)

[Tsa93] Edward Tsang. *Foundations of Constraint Satisfaction (Computation in Cognitive Science).* Academic Press Inc., 1993. (Cited on page 46)

[VDK02] Arie Van Deursen and Paul Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002. (Cited on page 193)

[VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *Sigplan Notices*, 35(6):26–36, 2000. (Cited on page 180)

[Ver99] Dinesh C. Verma. *Supporting Service Level Agreements on IP Networks.* Macmillan Technical Publishing, 1999. (Cited on pages 17 and 18)

[Wat05] Bret Waters. Software as a Service: A Look at the Customer Benefits. *Journal of Digital Asset Management*, 1:32–39, 2005. (Cited on page 11)

[WDS09] Jules White, Brian Dougherty, and Douglas C. Schmidt. Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening. *Journal of Systems and Software (JSS)*, 82:1268–1284, 2009. (Cited on pages 46 and 94)

[WfMC99] Workflow Management Coalition. The Workflow Management Coalition Specification: Terminology & Glossary. `http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf`, 1999. Document WFMC-TC-1011, Issue 3.0. (Cited on page 44)

[WKM12] Erik Wittern, Jörn Kuhlenkamp, and Michael Menzel. Cloud Service Selection Based on Variability Modeling. In *Proceedings of the 10th International Conference on Service-Oriented Computing*, ICSOC '12, pages 127–141, Berlin, Heidelberg, 2012. Springer-Verlag. (Cited on page 177)

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley Professional, 1999. (Cited on pages 31, 32, 34, 73 and 74)

[WSB+08] Jules White, Douglas C. Schmidt, David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated Diagnosis of Product-line Configuration Errors in Feature Models. In *Proceedings of the 12th International Software Product Line Conference*, SPLC '08, pages 225–234, Washington, DC, USA, 2008. IEEE Computer Society. (Cited on pages 46 and 251)

[WZ11] Erik Wittern and Christian Zirpins. On the Use of Feature Models for Service Design: The Case of Value Representation. In *Proceedings of the 2010 international conference on Towards a service-based internet*, ServiceWave '10, pages 110–118, Berlin, Heidelberg, 2011. Springer-Verlag. (Cited on page 34)

[WZL+11] Rui Wang, Yong Zhang, Shijun Liu, Lei Wu, and Xiangxu Meng. A Dependency-Aware Hierarchical Service Model for SaaS and Cloud Services. In *Proceedings of the IEEE International Conference on Services Computing*, SCC '11, pages 480–487, 2011. (Cited on page 89)

[XHSC12] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. Generating Range Fixes for Software Configuration. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE '12, pages 58–68, Piscataway, NJ, USA, 2012. IEEE Press. (Cited on page 46)

[YR06] Liguo Yu and Srini Ramaswamy. A Configuration Management Model for Software Product Line, 2006. (Cited on page 51)

[Zac87] John A. Zachman. A Framework for Information Systems Architecture. *IBM Systems Journal*, 26(3):276–292, 1987. (Cited on page 72)

[ZKT10] Lamia Abo Zaid, Frederic Kleinermann, and Olga De Troyer. Feature Assembly: A New Feature Modeling Technique. In *Proceedings of the 29th International Conference on Conceptual Modeling*, ER '10, pages 233–246. Springer, 2010. (Cited on page 124)

[ZYZJ08] Wei Zhang, Hua Yan, Haiyan Zhao, and Zhi Jin. A BDD-Based Approach to Verifying Clone-Enabled Feature Models' Constraints and Customization. In *High Confidence Software Reuse in Large Systems*, volume 5030 of *Lecture Notes in Computer Science*, pages 186–199. Springer Berlin Heidelberg, 2008. (Cited on page 46)

Part III.

Appendix

A. Translation of Extended Feature Model in Constraint Satisfaction Problem

An extended feature model containing attributes and group-cardinality, as introduced in Section 6.3, is translated into a CSP by applying the rules *a)* to *l)* depicted in Table A.1. Most of the translation rules have been proposed previously in literature. As such, rules *a), d), f), g)* are taken from [BTRC05], while rule *e)* was proposed in [KOD10], and rules *h), i)* originate in [MSDLM11]. Translation rules *b)* and *c)* express the configuration state of features in a partial configuration, as proposed in [WSB$^+$08]. The translation rules *j)* and *k)* are added to specify the configuration state of attributes in a partial feature model configuration.

Each feature is represented as a CSP variable with a Boolean domain $[0, 1]$, as stated by translation rule *a)*, where 0 represents the deselection of a feature and 1 the selection accordingly. A selected or deselected feature adds an additional constraint to the CSP limiting the feature variable assignment, as shown in the translation rules *b)* and *c)*. The relationship between parent and child feature is translated into an implication constraint, as depicted in translation rule *d)*.

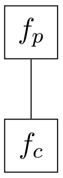
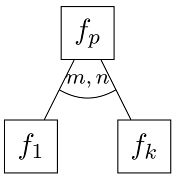
The decomposition relation of grouped features is expressed as sum of all feature variables, as expressed in translation rule *e)*. This avoids the combinatoric explosion by representing the constraint as a propositional logic term instead. In contrast to proposed translations in literature, solitary features are not handled differently from grouped features in the approach proposed in this work, as both are modeled in a feature group, as discussed in Section 2.3.2. Cross-tree constraints on features are translated into implication constraints in the CSP, as stated in the translation rules *f)* and *g)*.

Each attribute is represented as a CSP variable with the domain either represented as an enumeration such as $[1, 2, 5]$ or intervals with an upper and lower bound such as $[0 - 5]$ depending on the domain in the feature model as expressed by translation rules *h)* and *i)*. Thus, each attribute domain variable contains a mandatory value 0 to express that an attribute is not assigned or disabled [MSDLM11]. Any other value represents an attribute value assignment. Thus, the selection of the attribute's feature leads to the evaluation of the attribute domain values. In contrast, if the feature is deselected, the attribute has the value 0 assigned. Attributes are evaluated with respect to an attribute's feature. Hence, explicitly assigning the attribute value 0 to an attribute is evaluated correctly if the feature is selected. For deselected features, the attribute value is irrelevant.

The assignment of an attribute value to an attribute is only evaluated if the attribute's feature is selected. This constraint is expressed in the CSP as depicted by translation rule *j)*. Furthermore, translation rule *k)* expresses that deselected attribute values are only evaluated if the attribute's feature is selected.

Cross-tree constraints on two attributes are translated, as expressed by translation rule *l)*. If two attributes of two different features are constrained in the feature model, a biimplication constraint on the features is introduced in the CSP to ensure that both features are selected to evaluate the attribute constraint. Additionally, a constraint on an attribute value is translated into the CSP according to translation rule *m)*.

Table A.1 Rules for translating an extended feature model into a Constraint Satisfaction Problem.

	Feature model concept	**Graphical notation**	**Constraint satisfaction problem representation**
a)	Undecided feature f	$\boxed{f}$	$0 \leq f \leq 1$
b)	Selected feature f	$\boxed{+f}$	$f = 1$
c)	Deselected feature f	$\boxed{-f}$	$f = 0$
d)	Relation between parent feature f_p and child feature f_c		$(f_c = 1) \rightarrow (f_p = 1)$
e)	Decomposition relation with group cardinality [m,n] between grouped features $f_1, \ldots, f_k$ and their parent feature f_p		$(f_p = 1) \rightarrow$ $(m \leq sum(f_1, \ldots, f_k) \leq n)$
f)	Requires relation between feature f_j and f_k	$\boxed{f_j} \dashrightarrow \boxed{f_k}$	$(f_j = 1) \rightarrow (f_k = 1)$
g)	Excludes relation between two features f_j and f_k	$\boxed{f_j} \leftarrow\!\dashrightarrow \boxed{f_k}$	$(f_j = 1) \rightarrow (f_k = 0)$, $(f_k = 1) \rightarrow (f_j = 0)$
h)	Attribute values $v_1, \ldots, v_i$ are specified as enumeration in the discrete domain D of attribute a which belongs to feature f	$\begin{array}{\|c\|}\hline f \\ \hline \text{a : D} = \{v_1, \ldots, v_i\} \\ \hline\end{array}$	$(f = 1) \rightarrow (a \in \{v_1, \ldots, v_i\})$, $(f = 0) \rightarrow (a = 0)$

Continued on next page

Table A.1 – continued from previous page

	Feature model concept	Graphical notation	Constraint satisfaction problem representation
i)	Attribute values expressed as integer intervals with upper and lower bound $[v_{min}, v_{max}]$ in the discrete domain D of attribute a that belongs to feature f	f a : D = $[v_{min}, v_{max}]$	$(f = 1) \rightarrow (a \in [v_{min}, v_{max}])$, $(f = 0) \rightarrow (a = 0)$
j)	Assigned attribute value v_1 of attribute a with discrete domain D and related to feature f	f a : D := v_1	$(f = 1) \rightarrow (a = v_1)$
k)	Deselected attribute values v_1, v_2 of attribute a with discrete domain D and related to feature f	f a : D$\backslash\{v_1, v_2\}$	$(f = 1) \rightarrow (a \notin \{v_1, v_2\})$
l)	Relational constraint on two attributes a_1 and a_2, where a_1 with domain D_1 belongs to feature f_1 and a_2 with domain D_2 belong to feature f_2; domains D_1 and D_2 and features f_1 and f_2 are not compellingly equal	f_1 $a_1 : D_1$ - - f_2 $a_2 : D_2$	$a_1 \circ a_2, with$ $\circ \in \{=, \neq, \leq, \geq, >, <\}$, $(f_1 = 1) \leftrightarrow (f_2 = 1)$
m)	Relational constraint on an integer value v_i and an attribute a_1 of feature f_1 with domain D_1	f_1 $a_1 : D_1$ — $a_1 \circ v_i$	$a_1 \circ v_i, with$ $\circ \in \{=, \neq, \leq, \geq, >, <\}$

B. EFeatureText: Extended Feature Model Language

The textual modeling language *EFeatureText* for defining attributed and group-cardinality based feature models is implemented in Ecore and EMFText. This language is applied in the tool suite PUMA to specify feature models textually as explained in Chapter 7.

B.1. Concrete Syntax of EFeatureText

The concrete syntax of this language is specified in EMFText as shown in the following Listing B.1.

```
@SuppressWarnings(tokenOverlapping)
SYNTAXDEF eft //Extended Featuremodel Text
FOR <http://www.tudresden.de/extfeature>
START FeatureModel

TOKENS {
  DEFINE INTEGER $('0'..'9')+ $;
  DEFINE COMMENT $'//'(~('\n'|'\r'|'\uffff'))* $;
}


TOKENSTYLES {
  "INTEGER" COLOR #2A00FF;
  "COMMENT" COLOR #AAAAAA;
  ":=" COLOR #009E0F, BOLD;
  "selected" COLOR #009E0F, BOLD;
  "deselected" COLOR #CE0000, BOLD;
}

RULES {
    // syntax definition for container class 'FeatureModel'
  FeatureModel ::= "featuremodel" #1 name['"','"'] !0!0
            domains* !0 root !0 constraints* ;

    // syntax definition for features and their configuration state
  Feature ::= configurationState[selected : "selected", deselected : "deselected
   ", unbound : ""] #1 "feature" #1 name['"','"'] #1 "<" id[] ">"
        (!1 (attributes | groups) )*;

    // syntax definition for groups
  Group ::= "group" #1 "<"id[]">" #1 "(" minCardinality[INTEGER] ".."
   maxCardinality[INTEGER] ")"
```

```
            #1 "{" (!1 childFeatures)+ "}" !0;

    // syntax definition for attributes and their configuration state
  Attribute ::= name[] #1 "[" domain[] "]" ("\\" "{" deselectedDomainValues[’"’
    ,’"’] ("," #1 deselectedDomainValues[’"’,’"’])* "}")?
  (#1 ":=" #1 (value[’"’,’"’]))? ;

    // syntax definition for attribute domains
  NumericalDomain ::= "domain" #1 "<" id[] ">" #1 "[" intervals ("," #1
    intervals)* "]" !0;
  Interval ::= lowerBound[INTEGER] ".." upperBound[INTEGER];

  DiscreteDomain ::= "domain" #1 "<" id[] ">" #1 "[" values ("," #1 values)* "]
    " !0;
  DomainValue ::= (name[] "=")? #0 int[INTEGER];

    // syntax definition for cross-tree constraints
  Imply ::= "constraint" #1 "<"id[]">" #1 leftOperand[] #1 "->" #1 rightOperand
    [] !0;
  Exclude ::= "constraint" #1 "<"id[]">" #1 leftOperand[] #1 "<->" #1
    rightOperand[] !0;

  AttributeConstraint ::= "constraint" #1 "<"id[]">" #1
      attribute1 #1
      operator[equal : "==", unequal : "!=", greaterThan : ">",
    greaterThanOrEqual : ">=", lessThan : "<", lessThanOrEqual : "<="] #1
      attribute2 !0;

  AttributeReference ::= feature[] #0 "." #0 attribute[];

  AttributeValue ::= (name[’"’,’"’] | int[INTEGER]);
}
```

Listing B.1 Concrete syntax of textual language *EFeatureText*.

B.2. SAP Business ByDesign Example

The SaaS application *Business ByDesign* is designed as a configurable CRM application for small and medium size companies. To calculate the price of a tailored application, a configuration self-service portal is offered to customers. The configuration view of this portal presents available features and the dependencies among them to the customers. A feature model is extracted from this configuration view. Table B.1 shows the metrics of this feature model. The *Business ByDesign* feature model written in EFeatureText is depicted in Listing B.2. The listing shows the domain feature model that specifies all derivable variant configurations.

Table B.1 Metrics of *Business ByDesign* feature model.

Metrics	Value
Number of features	78
Number of attributes	2
Number of cross-tree constraints	23
Constraint feature coverage	37%
Constraint attribute coverage	100%
Is feature model satisfiable	*true*

```
featuremodel "Business ByDesign"

domain <employeeDomain> [10..10000]
domain <userDomain> [10..10000]

feature "Customer Relationship Management" <crm> group <stakeholderConfiguration
    > (1..1) {
  feature "Stakeholders" <stakeholders>
    employees [employeeDomain]
    users [userDomain] }
group <modules> (1..17) {
  feature "Marketing" <Marketing> group <marketingSelection> (1..2) {
    feature "Market Development" <Market_Development>
    feature "Campaign Management" <Campaign_Management>
  }

  feature "Sales" <Sales> group <salesSelection> (1..6) {
    feature "Account and Activity Management" <Account_and_Activity_Management>
    feature "Product and Service Portfolio for Sales" <
   Product_and_Service_Portfolio_for_Sales>
    feature "New Business" <New_Business>
    feature "Selling Products and Services" <Selling_Products_and_Services>
    feature "Customer Invoicing" <Customer_Invoicing>
    feature "Sales Planning" <Sales_Planning>
  }

  feature "Service" <Service> group <serviceSelection> (1..4) {
    feature "Entitlement Management" <Entitlement_Management>
    feature "Product and Service Portfolio for Field Service and Repair" <
   Product_and_Service_Portfolio_for_Field_Service_and_Repair>
```

```
    feature "Customer Care" <Customer_Care>
    feature "Field Service and Repair" <Field_Service_and_Repair>
  }

  feature "Sourcing" <Sourcing> group <sourcingSelection> (1..2) {
    feature "Supplier Base Management" <Supplier_Base_Management>
    feature "Sourcing and Contracting" <Sourcing_and_Contracting>
  }

  feature "Purchasing" <Purchasing> group <purchasingSelection> (1..3) {
    feature "Self-Service Procurement" <SelfService_Procurement>
    feature "Purchase Request and Order Management" <
    Purchase_Request_and_Order_Management>
    feature "Supplier Invoicing" <Supplier_Invoicing>
  }

  feature "Product Development" <Product_Development> group <productSelection>
    (1..2) {
    feature "Product Definition" <Product_Definition>
    feature "Product Engineering" <Product_Engineering>
  }

  feature "Supply Chain Setup Management" <Supply_Chain_Setup_Management> group
    <supplychainSetupSelection> (1..3) {
    feature "Supply Chain Design" <Supply_Chain_Design>
    feature "Execution Design" <Execution_Design>
    feature "Production Models" <Production_Models>
  }

  feature "Supply Chain Planning and Control" <Supply_Chain_Planning_and_Control
    > group <supplychainPlanningSelection> (1..2) {
    feature "Demand Planning" <Demand_Planning>
    feature "Demand Management and Order Confirmation" <
    Demand_Management_and_Order_Confirmation>
    feature "Exception Monitoring and Control" <Exception_Monitoring_and_Control
    >
    feature "Supply Planning" <Supply_Planning>
    feature "Supply Control" <Supply_Control>
    feature "Logistics Control" <Logistics_Control>
  }

  feature "Manufacturing Warehousing and Logistics" <
    Manufacturing_Warehousing_and_Logistics> group <manufacturingSelection>
    (1..8) {
    feature "Inbound Logistics" <Inbound_Logistics>
    feature "Outbound Logistics" <Outbound_Logistics>
    feature "Internal Logistics" <Internal_Logistics>
    feature "Inventory Management" <Inventory_Management>
    feature "Production" <Production>
    feature "Quality Assurance" <Quality_Assurance>
    feature "Tracking Tracing and Identification" <
    Tracking_Tracing_and_Identification>
    feature "Task Management and Automation" <Task_Management_and_Automation>
  }

  feature "Project Management" <Project_Management> group <projectSelection>
    (1..1) {
    feature "Project Planning and Execution" <Project_Planning_and_Execution>
```

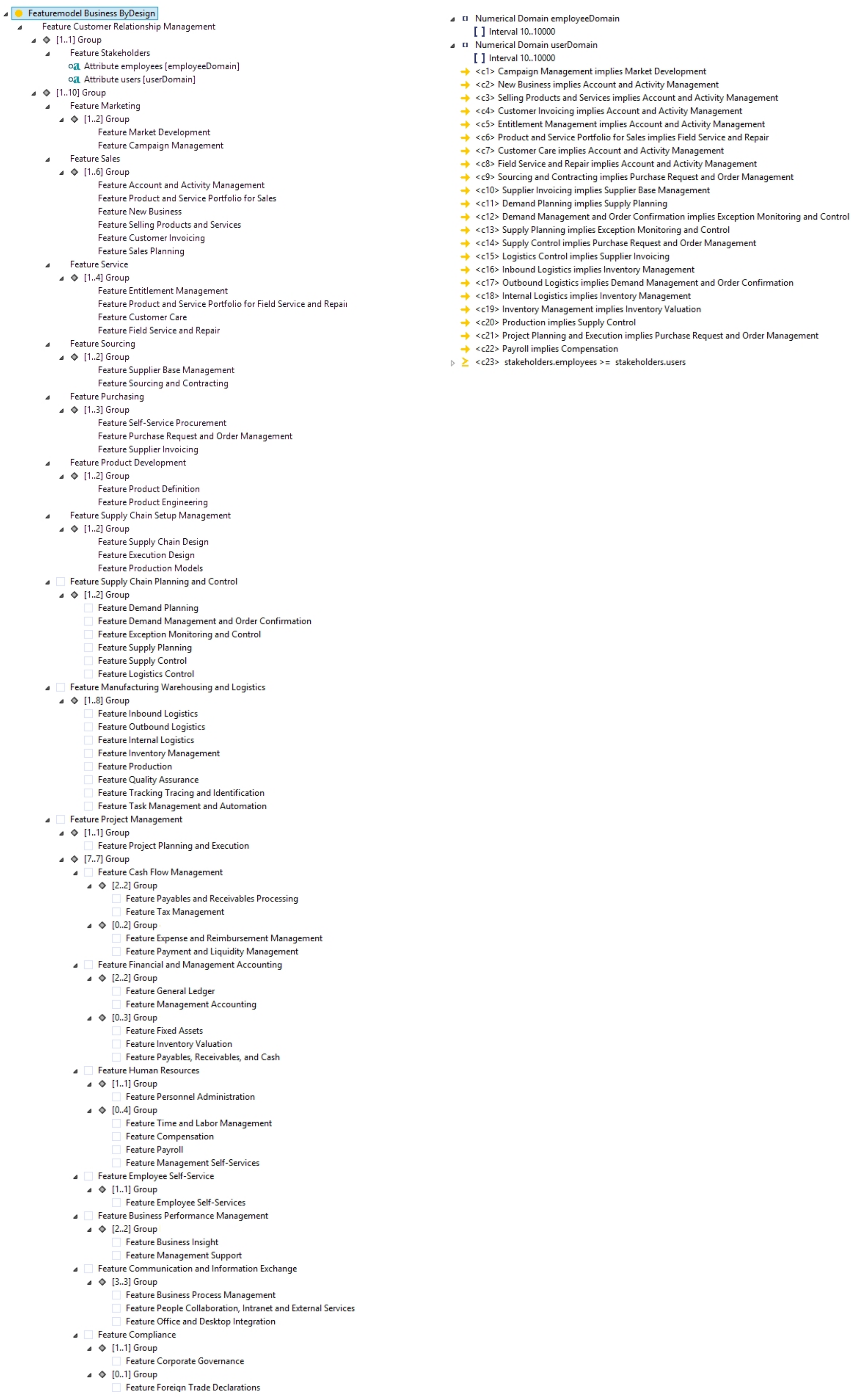
```
}

feature "Cash Flow Management" <Cash_Flow_Management>
group <cashFlowObligatorySelection> (2..2) {
  feature "Payables and Receivables Processing" <
 Payables_and_Receivables_Processing>
  feature "Tax Management" <Tax_Management>
}

group <cashFlowSelection> (0..2) {
  feature "Expense and Reimbursement Management" <
 Expense_and_Reimbursement_Management>
  feature "Payment and Liquidity Management" <Payment_and_Liquidity_Management
 >
}

feature "Financial and Management Accounting" <
 Financial_and_Management_Accounting>
group <financialObligatorySelection> (2..2) {
  feature "General Ledger" <General_Ledger>
  feature "Management Accounting" <Management_Accounting>
}

group <financialSelection> (0..3) {
  feature "Fixed Assets" <Fixed_Assets>
  feature "Inventory Valuation" <Inventory_Valuation>
  feature "Payables, Receivables, and Cash" <Payables_Receivables_and_Cash>
}

feature "Human Resources" <Human_Resources>
group <hrObligatorySelection> (1..1) {
  feature "Personnel Administration" <Personnel_Administration>
}

group <hrSelection> (0..3) {
  feature "Time and Labor Management" <Time_and_Labor_Management>
  feature "Compensation" <Compensation>
  feature "Payroll" <Payroll>
}

feature "Employee Self-Service" <Employee_SelfService> group <
 employeeSelection> (1..2) {
  feature "Employee Self-Services" <Employee_SelfServices>
  feature "Management Self-Services" <Management_SelfServices>
}

feature "Business Performance Management" <Business_Performance_Management>
 group <businessObligatorySelection> (2..2) {
  feature "Business Insight" <Business_Insight>
  feature "Management Support" <Management_Support>
}

feature "Communication and Information Exchange" <
 Communication_and_Information_Exchange> group <
 communicationObligatorySelection> (3..3) {
  feature "Business Process Management" <Business_Process_Management>
  feature "People Collaboration, Intranet and External Services" <
 People_Collaboration_Intranet_and_External_Services>
```

```
    feature "Office and Desktop Integration" <Office_and_Desktop_Integration>
  }
  feature "Compliance" <Compliance>
  group <complianceObligatorySelection> (1..1) {
    feature "Corporate Governance" <Corporate_Governance>
  }

  group <complianceSelection> (0..1) {
    feature "Foreign Trade Declarations" <Foreign_Trade_Declarations>
  }
}
constraint <c1> Campaign_Management -> Market_Development
constraint <c2> New_Business -> Account_and_Activity_Management
constraint <c3> Selling_Products_and_Services -> Account_and_Activity_Management
constraint <c4> Customer_Invoicing -> Account_and_Activity_Management
constraint <c5> Entitlement_Management -> Account_and_Activity_Management
constraint <c6> Product_and_Service_Portfolio_for_Sales ->
   Field_Service_and_Repair
constraint <c7> Customer_Care -> Account_and_Activity_Management
constraint <c8> Field_Service_and_Repair -> Account_and_Activity_Management
constraint <c9> Sourcing_and_Contracting ->
   Purchase_Request_and_Order_Management
constraint <c10> Supplier_Invoicing -> Supplier_Base_Management
constraint <c11> Demand_Planning -> Supply_Planning
constraint <c12> Demand_Management_and_Order_Confirmation ->
   Exception_Monitoring_and_Control
constraint <c13> Supply_Planning -> Exception_Monitoring_and_Control
constraint <c14> Supply_Control -> Purchase_Request_and_Order_Management
constraint <c15> Logistics_Control -> Supplier_Invoicing
constraint <c16> Inbound_Logistics -> Inventory_Management
constraint <c17> Outbound_Logistics -> Demand_Management_and_Order_Confirmation
constraint <c18> Internal_Logistics -> Inventory_Management
constraint <c19> Inventory_Management -> Inventory_Valuation
constraint <c20> Production -> Supply_Control
constraint <c21> Project_Planning_and_Execution ->
   Purchase_Request_and_Order_Management
constraint <c22> Payroll -> Compensation
constraint <c23> stakeholders.employees >= stakeholders.users
```

Listing B.2 Extended feature model specification of the Business ByDesign example written in EFeatureText.

B.2.1. Outline View

A graphical notation of the *Business ByDesign* feature model is generated according to the textual specification in Listing B.2 as depicted in Figure B.1.



```
Featuremodel Business ByDesign
    Feature Customer Relationship Management
        [1..1] Group
            Feature Stakeholders
                Attribute employees [employeeDomain]
                Attribute users [userDomain]
        [1..10] Group
            Feature Marketing
                [1..2] Group
                    Feature Market Development
                    Feature Campaign Management
            Feature Sales
                [1..6] Group
                    Feature Account and Activity Management
                    Feature Product and Service Portfolio for Sales
                    Feature New Business
                    Feature Selling Products and Services
                    Feature Customer Invoicing
                    Feature Sales Planning
            Feature Service
                [1..4] Group
                    Feature Entitlement Management
                    Feature Product and Service Portfolio for Field Service and Repai
                    Feature Customer Care
                    Feature Field Service and Repair
            Feature Sourcing
                [1..2] Group
                    Feature Supplier Base Management
                    Feature Sourcing and Contracting
            Feature Purchasing
                [1..3] Group
                    Feature Self-Service Procurement
                    Feature Purchase Request and Order Management
                    Feature Supplier Invoicing
            Feature Product Development
                [1..2] Group
                    Feature Product Definition
                    Feature Product Engineering
            Feature Supply Chain Setup Management
                [1..2] Group
                    Feature Supply Chain Design
                    Feature Execution Design
                    Feature Production Models
            Feature Supply Chain Planning and Control
                [1..2] Group
                    Feature Demand Planning
                    Feature Demand Management and Order Confirmation
                    Feature Exception Monitoring and Control
                    Feature Supply Planning
                    Feature Supply Control
                    Feature Logistics Control
            Feature Manufacturing Warehousing and Logistics
                [1..8] Group
                    Feature Inbound Logistics
                    Feature Outbound Logistics
                    Feature Internal Logistics
                    Feature Inventory Management
                    Feature Production
                    Feature Quality Assurance
                    Feature Tracking Tracing and Identification
                    Feature Task Management and Automation
            Feature Project Management
                [1..1] Group
                    Feature Project Planning and Execution
                [7..7] Group
                    Feature Cash Flow Management
                        [2..2] Group
                            Feature Payables and Receivables Processing
                            Feature Tax Management
                        [0..2] Group
                            Feature Expense and Reimbursement Management
                            Feature Payment and Liquidity Management
                    Feature Financial and Management Accounting
                        [2..2] Group
                            Feature General Ledger
                            Feature Management Accounting
                        [0..3] Group
                            Feature Fixed Assets
                            Feature Inventory Valuation
                            Feature Payables, Receivables, and Cash
                    Feature Human Resources
                        [1..1] Group
                            Feature Personnel Administration
                        [0..4] Group
                            Feature Time and Labor Management
                            Feature Compensation
                            Feature Payroll
                            Feature Management Self-Services
                    Feature Employee Self-Service
                        [1..1] Group
                            Feature Employee Self-Services
                    Feature Business Performance Management
                        [2..2] Group
                            Feature Business Insight
                            Feature Management Support
                    Feature Communication and Information Exchange
                        [3..3] Group
                            Feature Business Process Management
                            Feature People Collaboration, Intranet and External Services
                            Feature Office and Desktop Integration
                    Feature Compliance
                        [1..1] Group
                            Feature Corporate Governance
                        [0..1] Group
                            Feature Foreign Trade Declarations
    Numerical Domain employeeDomain
        [ ] Interval 10..10000
    Numerical Domain userDomain
        [ ] Interval 10..10000
    <c1> Campaign Management implies Market Development
    <c2> New Business implies Account and Activity Management
    <c3> Selling Products and Services implies Account and Activity Management
    <c4> Customer Invoicing implies Account and Activity Management
    <c5> Entitlement Management implies Account and Activity Management
    <c6> Product and Service Portfolio for Sales implies Field Service and Repair
    <c7> Customer Care implies Account and Activity Management
    <c8> Field Service and Repair implies Account and Activity Management
    <c9> Sourcing and Contracting implies Purchase Request and Order Management
    <c10> Supplier Invoicing implies Supplier Base Management
    <c11> Demand Planning implies Supply Planning
    <c12> Demand Management and Order Confirmation implies Exception Monitoring and Control
    <c13> Supply Planning implies Exception Monitoring and Control
    <c14> Supply Control implies Purchase Request and Order Management
    <c15> Logistics Control implies Supplier Invoicing
    <c16> Inbound Logistics implies Inventory Management
    <c17> Outbound Logistics implies Demand Management and Order Confirmation
    <c18> Internal Logistics implies Inventory Management
    <c19> Inventory Management implies Inventory Valuation
    <c20> Production implies Supply Control
    <c21> Project Planning and Execution implies Purchase Request and Order Management
    <c22> Payroll implies Compensation
    <c23>  stakeholders.employees >=  stakeholders.users
```

Figure B.1 Outline of the BusinessByDesign domain feature model.

B.3. Video Information System Example

The video information system example applied in Chapter 4 is modeled as a feature model with attributes. Table B.2 shows the metrics of this feature model. The unconfigured feature model written in EFeatureText is represented in Listing B.5.

Table B.2 Metrics of the *video information system* feature model.

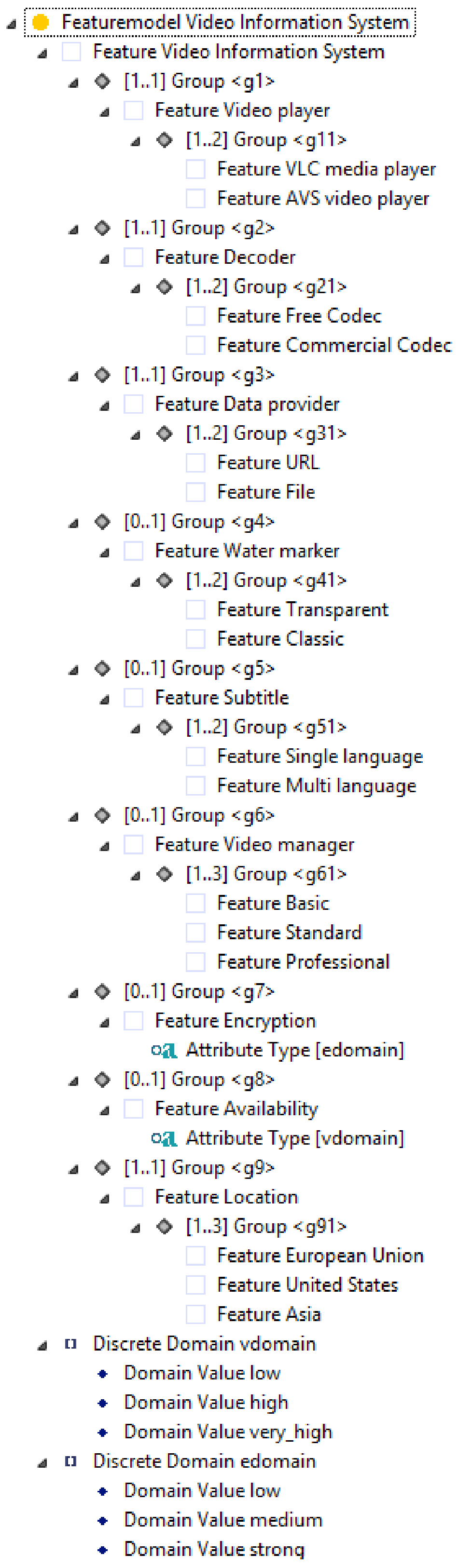
Metrics	Value
Number of features	26
Number of attributes	2
Number of cross-tree constraints	0
Constraint feature coverage	0%
Constraint attribute coverage	0%
Is feature model satisfiable	*true*
Number of derivable variants	387072

```
featuremodel "Video Information System"
domain <vdomain>  [low=1, high=2, very_high=3]
domain <edomain>  [low=1, medium=2, strong=3]

feature "Video Information System" <vis>
  group <g1> (1..1) {
    feature "Video player" <vp>
      group <g11> (1..2) {
        feature "VLC media player" <vlc>
        feature "AVS video player" <avs>
      }
  }
  group <g2> (1..1) {
    feature "Decoder" <dec>
      group <g21> (1..2) {
        feature "Free Codec" <fc>
        feature "Commercial Codec" <cc>
      }
  }
  group <g3> (1..1) {
    feature "Data provider" <dp>
    group <g31> (1..2) {
      feature "URL" <url>
      feature "File" <file>
    }
  }
  group <g4> (0..1) {
    feature "Water marker" <wm>
    group <g41> (1..2) {
      feature "Transparent" <trans>
      feature "Classic" <classic>
    }
  }
  group <g5> (0..1) {
    feature "Subtitle" <st>
      group <g51> (1..2) {
        feature "Single language" <sl>
        feature "Multi language" <ml>
      }
  }
  group <g6> (0..1) {
    feature "Video manager" <vm>
      group <g61> (1..3) {
        feature "Basic" <basic>
        feature "Standard" <standard>
        feature "Professional" <prof>
      }
  }
  group <g7> (0..1) {
    feature "Encryption" <en>
      Type[edomain]
    }
  group <g8> (0..1) {
    feature "Availability" <av>
      Type[vdomain]
  }
  group <g9> (1..1) {
    feature "Location" <loc>
      group <g91> (1..3) {
        feature "European Union" <EU>
        feature "United States" <US>
        feature "Asia" <AS>
      }
  }
```

Listing B.3 Extended feature model specification of the yard management system example written in EFeatureText.

B.3.1. Outline View

According to the textual specification of the model shown in Listing B.3, a tree-like outline is generated accordingly as depicted in Figure B.2.



```
Featuremodel Video Information System
  Feature Video Information System
    [1..1] Group <g1>
      Feature Video player
        [1..2] Group <g11>
          Feature VLC media player
          Feature AVS video player
    [1..1] Group <g2>
      Feature Decoder
        [1..2] Group <g21>
          Feature Free Codec
          Feature Commercial Codec
    [1..1] Group <g3>
      Feature Data provider
        [1..2] Group <g31>
          Feature URL
          Feature File
    [0..1] Group <g4>
      Feature Water marker
        [1..2] Group <g41>
          Feature Transparent
          Feature Classic
    [0..1] Group <g5>
      Feature Subtitle
        [1..2] Group <g51>
          Feature Single language
          Feature Multi language
    [0..1] Group <g6>
      Feature Video manager
        [1..3] Group <g61>
          Feature Basic
          Feature Standard
          Feature Professional
    [0..1] Group <g7>
      Feature Encryption
        Attribute Type [edomain]
    [0..1] Group <g8>
      Feature Availability
        Attribute Type [vdomain]
    [1..1] Group <g9>
      Feature Location
        [1..3] Group <g91>
          Feature European Union
          Feature United States
          Feature Asia
  Discrete Domain vdomain
    Domain Value low
    Domain Value high
    Domain Value very_high
  Discrete Domain edomain
    Domain Value low
    Domain Value medium
    Domain Value strong
```

Figure B.2 Outline of the feature model of a video information system example.

B.4. Extended Document Management System Example

A typical document management system comprises several features regarding document types, indexing and searching capabilities. An example SPL for document management systems is modeled as a feature model. Table B.3 shows the metrics of this feature model.

Table B.3 Metrics of *document management system* feature model.

Metrics	Value
Number of features	46
Number of attributes	4
Number of cross-tree constraints	8
Constraint feature coverage	30%
Constraint attribute coverage	50%
Is feature model satisfiable	*true*

```
featuremodel "Extended Document Management System"

domain <avdomain>  [low=1, high=2, very_high=3]
domain <encdomain>  [low=1, medium=2, strong=3]
domain <amount> [1..10000]

feature "Document Management System" <dms>
  group <DocumentTypeGroup> (1..1) {
    feature "Document Type" <DocumentType>
      group <TypesGroup> (1..4) {
        feature "UnicodeText Type" <UnicodeTextType>
        feature "Text Type" <TextType>
        feature "Image Type" <ImageType>
        feature "PDF Type" <PDFType>
    }
  }
  group <OCRGroup> (0..1) {
    feature "OCR" <OCR>
      group <OCRTypes> (1..2) {
        feature "PDF OCR" <PDFOCR>
        feature "Image OCR" <ImageOCR>
      }
  }
  group <IndexGroup> (1..1) {
    feature "Indexing" <Indexing>
      group <MetaIndexing> (0..1) {
        feature "MetaData Index" <MetaDataIndex>
          group <AuthorGroupIndex> (0..1) {
            feature "Author Index" <AuthorIndex>
          }
          group <TitleGroupIndex> (1..1) {
            feature "Title Index" <TitleIndex>
          }
          group <ContentGroupIndex> (1..1) {
```

```
          feature "Content Index" <ContentIndex>
        }
        feature "General Index" <GeneralIndex>
      }
      group <FileIndex> (1..1) {
        feature "FileName Index" <FileNameIndex>
      }
  }
  group <SearchGroup> (1..1) {
    feature "Search" <Search>
      group <MetaSearch> (0..1) {
        feature "MetaData Search" <MetaDataSearch>
          group <AuthorGroupSearch> (0..1) {
            feature "Author Search" <AuthorSearch>
          }
          group <TitleGroupSearch> (1..1) {
            feature "Title Search" <TitleSearch>
          }
          group <ContentGroupSearch> (1..1) {
            feature "Content Search" <ContentSearch>
          }
        feature "General Search" <GeneralSearch>
      }
      group <FileSearch> (1..1) {
        feature "FileName Search" <FileNameSearch>
      }
  }
  group <QoSGroup> (0..1) {
    feature "Quality of Service" <QoS>
      availability[avdomain]
      encryption[encdomain]
      concurrentusers[amount]
  }
  group <platformGroup> (2..2){
    feature "Application Server" <AppServer>
      group <apsgroup> (1..1) {
        feature "SAPA HANA Cloud Platform" <HANACloud>
        feature "Eclipse Virgo" <Virgo>
      }
    feature "Database" <DB>
      users[amount]
      group <dbgroup> (1..1) {
        feature "SAP HANA" <HANA>
        feature "Oracle 12c" <Oracle>
        feature "MongoDB" <Mongo>
      }
  }
  group <LocationGroup> (1..1) {
    feature "Server Location" <Location>
      group <g91> (1..3) {
        feature "European Union" <EU>
          group <EUDatacenter> (1..4) {
            feature "Spain" <ESP>
            feature "Germany" <GER>
            feature "Norway" <NOR>
            feature "Ireland" <IRL>
          }
        feature "United States" <US>
```

```
          group <USDatacenter> (1..5) {
            feature "California" <CA>
            feature "Washington" <WA>
            feature "Alaska" <AK>
            feature "Texas" <TX>
            feature "Nebraska" <NE>
          }
        feature "Asia" <AS>
          group <ASDatacenter> (1..2) {
            feature "Russia" <RUS>
            feature "India" <IND>
          }
      }
  }

constraint <c1> MetaDataSearch -> MetaDataIndex
constraint <c2> GeneralSearch -> GeneralIndex
constraint <c3> ImageOCR -> ImageType
constraint <c4> PDFOCR -> PDFType
constraint <c5> AuthorSearch -> AuthorIndex
constraint <c6> TextType <-> UnicodeTextType
constraint <c7> HANA -> HANACloud
constraint <c8> QoS.concurrentusers <= DB.users
```

Listing B.4 Extended feature model specification of the document management system example written in EFeatureText.

The document management system feature model written in EFeatureText is depicted in Listing B.4. The listing shows the domain feature model that specifies all derivable variant configurations.

B.5. Yard Management Example

The *yard management* SPL is developed as a prototype in the industrial context of the *INDENICA* project[1]. Table B.4 shows the metrics of this feature model. The feature model of this SPL written in EFeatureText is represented in Listing B.5. The listing shows a partial configuration of the yard management SPL.

Table B.4 Metrics of *yard management* feature model

Metrics	Value
Number of features	20
Number of attributes	1
Number of cross-tree constraints	1
Constraint feature coverage	10%
Constraint attribute coverage	0%
Is feature model satisfiable	*true*
Number of derivable variants	1536

```
featuremodel "Yard Management System"
domain <scheduleType> [next=1, fitting=2]
selected feature "Yard Management System" <YMS>
  group <Authentication_opt> (0..1) {
       deselected feature "Authentication" <Authentication>
      group <JAAS_man> (1..1){
        deselected feature "Java Authentication and Authorization Service" <JAAS
    >
      }
  }
  group <Persistence_man>(1..1){
    selected feature "Persistence" <Persistence>
      group <PersistenceValue_alt>(1..1){
        deselected feature "Java Database Connectivity" <JDBC>
        selected feature "Java Persistence API" <JPA>
        }
  }
  group <Connectivity_man> (1..1){
    selected feature "Connectivity" <Connectivity>
      group <ConnectivityValue_alt>(1..1){
        deselected feature "RFC" <RFC>
        selected feature "SOAP" <SOAP>
        deselected feature "REST" <REST>
      }
  }
  group <YM_man> (1..1){
    feature "Yard Management Service" <YM>
        SchedulingType [scheduleType]
      group <EnableShips_opt>(0..1) {
        feature "Enable Ships" <EnableShips>
```

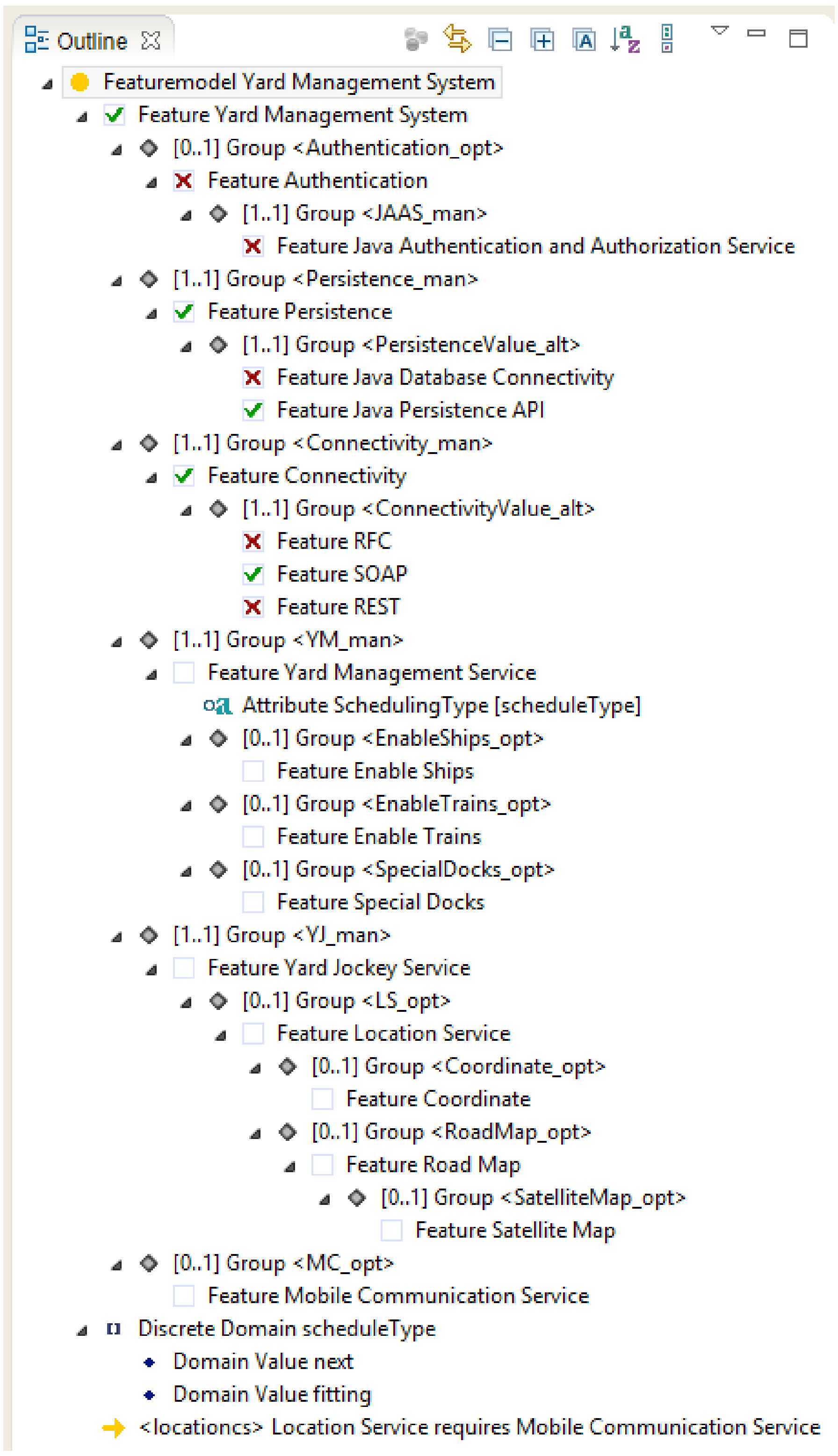
[1] http://www.indenica.eu/

```
      }
      group <EnableTrains_opt>(0..1) {
        feature "Enable Trains" <EnableTrains>
      }
      group <SpecialDocks_opt>(0..1) {
        feature "Special Docks" <SpecialDocks>
      }
  }
  group <YJ_man> (1..1){
    feature "Yard Jockey Service" <YJ>
      group <LS_opt> (0..1){
        feature "Location Service" <LS>
          group <Coordinate_opt>(0..1){
            feature "Coordinate" <Coordinate>
          }
          group <RoadMap_opt>(0..1){
            feature "Road Map" <RoadMap>
              group <SatelliteMap_opt> (0..1){
                feature "Satellite Map" <SatelliteMap>
              }
          }
      }
  }
  group <MC_opt> (0..1){
    feature "Mobile Communication Service" <MC>
  }
constraint <locationcs> LS -> MC
```

Listing B.5 Extended feature model specification of the yard management system example written in EFeatureText.

B.5.1. Outline View

According to the textual specification of the model shown in Listing B.5, a tree-like outline of the partial configuration is generated accordingly as depicted in Figure B.3.



```
Featuremodel Yard Management System
  ✓ Feature Yard Management System
    [0..1] Group <Authentication_opt>
      ✗ Feature Authentication
        [1..1] Group <JAAS_man>
          ✗ Feature Java Authentication and Authorization Service
    [1..1] Group <Persistence_man>
      ✓ Feature Persistence
        [1..1] Group <PersistenceValue_alt>
          ✗ Feature Java Database Connectivity
          ✓ Feature Java Persistence API
    [1..1] Group <Connectivity_man>
      ✓ Feature Connectivity
        [1..1] Group <ConnectivityValue_alt>
          ✗ Feature RFC
          ✓ Feature SOAP
          ✗ Feature REST
    [1..1] Group <YM_man>
      ☐ Feature Yard Management Service
        Attribute SchedulingType [scheduleType]
        [0..1] Group <EnableShips_opt>
          ☐ Feature Enable Ships
        [0..1] Group <EnableTrains_opt>
          ☐ Feature Enable Trains
        [0..1] Group <SpecialDocks_opt>
          ☐ Feature Special Docks
    [1..1] Group <YJ_man>
      ☐ Feature Yard Jockey Service
        [0..1] Group <LS_opt>
          ☐ Feature Location Service
            [0..1] Group <Coordinate_opt>
              ☐ Feature Coordinate
            [0..1] Group <RoadMap_opt>
              ☐ Feature Road Map
                [0..1] Group <SatelliteMap_opt>
                  ☐ Feature Satellite Map
    [0..1] Group <MC_opt>
      ☐ Feature Mobile Communication Service
  Discrete Domain scheduleType
    Domain Value next
    Domain Value fitting
  <locationcs> Location Service requires Mobile Communication Service
```

Figure B.3 Outline of a partial feature model configuration of the yard management example.

C. RBACText: Role Based Access Control Language

The textual modeling language *RBACText* is developed for restricting feature model configuration decisions in terms of Role Based Access Control (RBAC) and implemented in Ecore and EMFText. The language is applied in the tool *DyscoGraph* as explained in Section 7.6.

C.1. Concrete Syntax of RBACText

The concrete syntax of the language is specified in EMFText as shown in the following Listing C.1.

```
@SuppressWarnings(tokenOverlapping)
SYNTAXDEF rbactext // role based access control for feature models
FOR <http://www.tudresden.de/rbac>
START AccessControlModel

TOKENS {
  DEFINE COMMENT $'//'(~('\n'|'\r'|'\uffff'))* $ ;
  DEFINE TEXT $('A'..'Z'|'a'..'z'|'0'..'9'|'_')+ $;
  DEFINE LINEBREAK $ ('\r\n'|'\r'|'\n')+ $;
  DEFINE WHITESPACE $ (' '|'\t'|'\f')+ $ ;
}

TOKENSTYLES {
  "COMMENT" COLOR #AAAAAA;
  "assign" COLOR #147F87, BOLD;
  "select" COLOR #009E0F, BOLD;
  "deselect" COLOR #CE0000, BOLD;
  "abstract" COLOR #404040, BOLD;
}

  RULES {
  // syntax definition for container class 'AccessControlModel'
  AccessControlModel   ::= "access control" #1
                  ("on" #1 featureModel['<','>']) !0
                  (roles | groups | subjects)* ;

  // syntax definition for roles
  Role ::= type[abstract: "abstract", concrete: ""] "role" #1 name['"','"']? #1
    id['<','>'] #1 ("extends" #1 (parentRoles[]) (#1 "," #1 parentRoles[])*)? #1
      (("{" !1 ( permissions | tasks) #1 ("," #1 ( permissions | tasks) )* #1
      !0 "}") )? !0 ;
  // syntax definition for feature configuration operations
  FeatureOperation ::= #4 type[select : "select", deselect : "deselect"] #1
    feature[TEXT] !0;

  // syntax definition for attribute configuration operations
  AttributeOperation ::= #4 "assign" feature[] #0 "." #0 attribute[TEXT]  ;

  AttributeValueOperation ::= type[select : "select", deselect : "deselect"] #1
    feature[] #0 "." #0 attribute[TEXT] #0 "." #0 value[TEXT] !0;


  // syntax definition for subjects
  Subject ::= "subject" #1 name['"','"']? #1 id['<','>'] #1 "plays" #1 (roles[]
    ( #1 "," #1 roles[])* )? !0;

  // syntax definition for groups
  Group ::= "group" #1 name['"','"']? #1 id['<','>'] (#1 "of" #1 owner[])? !0
      "has members" #1 (members[] (#1 "," #1 members[])*)?;
}
```

Listing C.1 Concrete syntax of textual language *RBACText*.

C.2. Example of an Access Control Model for a Yard Management Application

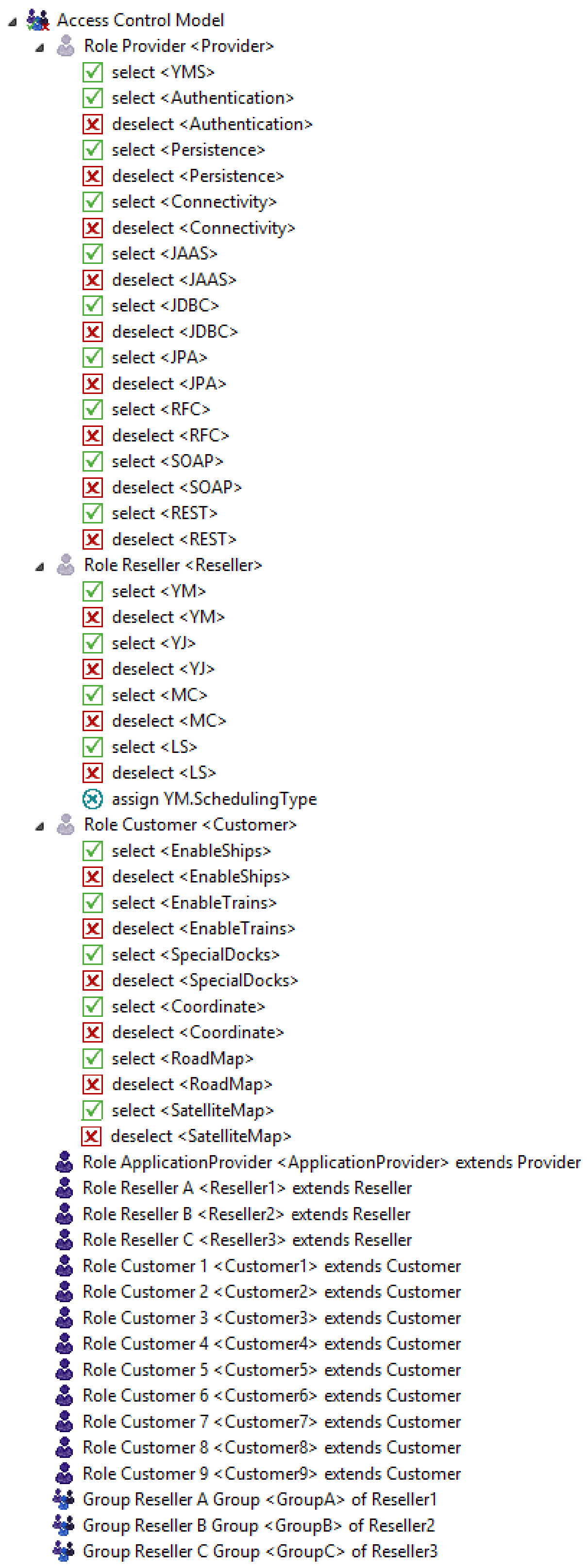
An example representation of how to apply the language is shown in the following Listing C.2. The exemplified access control model specifies restrictions on the referenced feature model of a variable yard management application specified in Listing B.5 and described in Appendix B.5.

```
access control on <YMS.eft>

abstract role "Provider" <Provider> {
  select YMS,
    select Authentication, deselect Authentication,
    select Persistence, deselect Persistence,
    select Connectivity, deselect Connectivity,
    select JAAS, deselect JAAS,
    select JDBC, deselect JDBC,
    select JPA, deselect JPA,
    select RFC, deselect RFC,
    select SOAP, deselect SOAP,
    select REST, deselect REST
}

abstract role "Reseller" <Reseller> {
  select YM, deselect YM,
  select YJ, deselect YJ,
  select MC, deselect MC,
  select LS, deselect LS,
  assign YM.SchedulingType }

abstract role "Customer" <Customer> {
  select EnableShips, deselect EnableShips,
  select EnableTrains, deselect EnableTrains,
  select SpecialDocks, deselect SpecialDocks,
  select Coordinate, deselect Coordinate,
  select RoadMap, deselect RoadMap,
  select SatelliteMap, deselect SatelliteMap }

role "ApplicationProvider" <ApplicationProvider> extends Provider

role "Reseller A" <Reseller1> extends Reseller
role "Reseller B" <Reseller2> extends Reseller
role "Reseller C" <Reseller3> extends Reseller

role "Customer 1" <Customer1> extends Customer
role "Customer 2" <Customer2> extends Customer
role "Customer 3" <Customer3> extends Customer
role "Customer 4" <Customer4> extends Customer
role "Customer 5" <Customer5> extends Customer
role "Customer 6" <Customer6> extends Customer
role "Customer 7" <Customer7> extends Customer
role "Customer 8" <Customer8> extends Customer
role "Customer 9" <Customer9> extends Customer

group "Reseller A Group" <GroupA> of Reseller1 has members Customer1, Customer2,
group "Reseller B Group" <GroupB> of Reseller2 has members Customer3, Customer4,
    Customer5, Customer6
group "Reseller C Group" <GroupC> of Reseller3 has members Customer7, Customer8,
    Customer9
```

Listing C.2 Specification of Role Based Access Control on the yard management system example written in RBACText.

C.3. Outline View

According to the textual specification of a model instance a tree-like outline is generated for the model as depicted in Figure C.1.



```
Access Control Model
  Role Provider <Provider>
    ✓ select <YMS>
    ✓ select <Authentication>
    ✗ deselect <Authentication>
    ✓ select <Persistence>
    ✗ deselect <Persistence>
    ✓ select <Connectivity>
    ✗ deselect <Connectivity>
    ✓ select <JAAS>
    ✗ deselect <JAAS>
    ✓ select <JDBC>
    ✗ deselect <JDBC>
    ✓ select <JPA>
    ✗ deselect <JPA>
    ✓ select <RFC>
    ✗ deselect <RFC>
    ✓ select <SOAP>
    ✗ deselect <SOAP>
    ✓ select <REST>
    ✗ deselect <REST>
  Role Reseller <Reseller>
    ✓ select <YM>
    ✗ deselect <YM>
    ✓ select <YJ>
    ✗ deselect <YJ>
    ✓ select <MC>
    ✗ deselect <MC>
    ✓ select <LS>
    ✗ deselect <LS>
      assign YM.SchedulingType
  Role Customer <Customer>
    ✓ select <EnableShips>
    ✗ deselect <EnableShips>
    ✓ select <EnableTrains>
    ✗ deselect <EnableTrains>
    ✓ select <SpecialDocks>
    ✗ deselect <SpecialDocks>
    ✓ select <Coordinate>
    ✗ deselect <Coordinate>
    ✓ select <RoadMap>
    ✗ deselect <RoadMap>
    ✓ select <SatelliteMap>
    ✗ deselect <SatelliteMap>
  Role ApplicationProvider <ApplicationProvider> extends Provider
  Role Reseller A <Reseller1> extends Reseller
  Role Reseller B <Reseller2> extends Reseller
  Role Reseller C <Reseller3> extends Reseller
  Role Customer 1 <Customer1> extends Customer
  Role Customer 2 <Customer2> extends Customer
  Role Customer 3 <Customer3> extends Customer
  Role Customer 4 <Customer4> extends Customer
  Role Customer 5 <Customer5> extends Customer
  Role Customer 6 <Customer6> extends Customer
  Role Customer 7 <Customer7> extends Customer
  Role Customer 8 <Customer8> extends Customer
  Role Customer 9 <Customer9> extends Customer
  Group Reseller A Group <GroupA> of Reseller1
  Group Reseller B Group <GroupB> of Reseller2
  Group Reseller C Group <GroupC> of Reseller3
```

Figure C.1 Graphical notation for RBACText illustrated on the yard management example.

D. MText: Textual Language for Mapping Viewgroups to Features

The textual modeling language *MText* is developed for defining a mapping between viewgroups and features in a multi-perspective model textually. The language is implemented in Ecore and EMFText and applied in the tool Conper as explained in Section 7.5.

D.1. Concrete Syntax of MText

The concrete syntax of the language MText is specified textually in EMFText as shown in the following Listing D.1.

```
SYNTAXDEF mtext
FOR <http://www.tudresden.de/viewmapping> <optional/path/to/myLanguage.genmodel>
START MappingModel

RULES {
  // syntax definition for class 'MappingModel'
  MappingModel    ::=  "viewmapping" !0
              #3 ("featuremodel" #1 featureModel['<','>']) !0
              #3 ("viewmodel" #1 viewModel['<','>']) !0!0
              (mappings*);

  // syntax definition for class 'Mapping'
  Mapping ::= "view group" #1 viewgroup['"','"'] #1 "contains" !0
        #3 features['"','"']  (#1 "," #1 features['"','"'] )* !0!0;
}
```

Listing D.1 Concrete syntax of textual language *MText*.