



Universität Stuttgart

A Method and Implementation to Define and Provision Variable Composite Applications, and its Usage in Cloud Computing

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der
Universität Stuttgart zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von
Ralph Mietzner
aus Stuttgart

Hauptberichter: Prof. Dr. Frank Leymann

Mitberichter: Univ. Prof. Dr. Schahram Dustdar

Tag der mündlichen Prüfung: 13. Juli 2010

Institut für Architektur von Anwendungssystemen
der Universität Stuttgart

2010

CONTENTS

1	Introduction	15
1.1	Introduction, Problem Domain and Motivation	15
1.2	Research Issues and Contributions	19
1.2.1	Application Metamodel and Component Dependencies	19
1.2.2	Separation of Application Vendors and Providers	20
1.2.3	Defining Variability in Composite Applications	21
1.2.4	Customization of Applications	21
1.2.5	An Architecture to Automatically Provision and Manage Applications	22
1.2.6	QoS-Aware Provisioning	22
1.3	Organization of the Thesis	22
2	Background and Related Work	25
2.1	Decomposition of Applications into Components	26
2.1.1	Component-Based Software Development	27
2.1.2	Service-Oriented Architecture	28
2.1.3	The Role of Middleware in Composite Applications	30
2.1.4	Software Product Line Engineering	31
2.2	Towards Utility and Cloud Computing	34
2.2.1	As a service models	35

2.2.2	Example SaaS Application	38
2.2.3	PaaS Examples	39
2.2.4	IaaS	43
2.2.5	Application Portals	45
2.2.6	Cloud Management Portals	45
2.2.7	Provisioning Engines	46
2.2.8	Topology Modeling and Provisioning	49
2.2.9	Composite Services and Applications	51
2.2.10	Cloud Interoperability	52
2.2.11	Package Standardization Efforts	53
2.3	Summary and Conclusion	58
3	Development of Cafe Application Templates	59
3.1	Roles in Cafe	61
3.1.1	Roles in Cafe	62
3.2	High-Level Cafe Development Process and Terminology	63
3.2.1	Template engineering phase	64
3.2.2	Template customization phase	65
3.2.3	Solution engineering phase	65
3.2.4	Provisioning phase	65
3.3	Cafe Application Metamodel	66
3.3.1	Short Overview	67
3.3.2	Example Application Model	67
3.3.3	Formal Definition	69
3.3.4	Components	70
3.3.5	Component Implementations	71
3.3.6	Multi-Tenancy Patterns	75
3.3.7	Deployment Graph	84
3.3.8	Atomic and Composite Components	86
3.3.9	Serialization of the Cafe Application Metamodel	88
3.4	Cafe Variability Metamodel	90
3.4.1	Requirements for a Variability Mechanism	90
3.4.2	Short Overview	91

3.4.3	Example Variability Model	92
3.4.4	Formal Definition	95
3.4.5	Enabling conditions and new alternatives for refinement variability points	120
3.4.6	Variability Models and Component Dependencies	123
3.4.7	Modeling Requirements on Provider Supplied Compo- nents with Variability	128
3.5	Providers and their Capabilities	132
3.5.1	A Metamodel for Providers	133
3.5.2	Component Binding	136
3.6	Car: Cafe Application Template Archive	139
3.7	Detailed Template Engineering Process	140
3.7.1	Selection of Available Components	142
3.7.2	Development of Missing Components	142
3.7.3	Variability Definition for Components	143
3.7.4	Component and Variability Integration	143
3.7.5	Customization Tool Development	144
3.7.6	Template Packaging	145
3.8	Summary and Conclusion	145
4	Customization of Cafe Application Templates	147
4.1	Requirements for a Customization Tool	149
4.1.1	Deal with Abstraction from Implementation Details	149
4.1.2	Guarantee Complete and Correct Customizations	150
4.1.3	Customer Guidance	151
4.2	Executing Variability Models	151
4.2.1	Different Purposes for Executable Variability Models	152
4.2.2	Operational Semantics for the Cafe Variability Metamodel	153
4.2.3	Complete and Correct Customizations	165
4.3	Generating Customization Flows from Variability Models	169
4.3.1	Introduction to PM-Graphs	170
4.3.2	Mapping Variability Models to PM-Graphs	171
4.3.3	Guarantee Complete and Correct Customizations	184

4.3.4	The Use of Customization Flows in Different Phases of the Cafe Development Process	188
4.4	Template Customization Phase	195
4.5	Solution Engineering Phase	196
4.5.1	Customization Phase	198
4.5.2	Solution Creation Phase	198
4.6	Summary and Conclusion	199
5	Provisioning and Management of Cafe Application Solutions	201
5.1	Service-Oriented Provisioning and Management of Cafe Applications	203
5.1.1	Provisioning Services Offered by Different Provisioning Engines	204
5.1.2	Provisioning and Management Flows	207
5.2	Cafe Provisioning and Management Interface (CPMI)	210
5.2.1	Mapping Provisioning and Management Services to Components via Component Flows	210
5.2.2	Generic Component States and Operations in the CPMI	214
5.2.3	Special Operations for Provider Components	219
5.2.4	Special Operations for Single (Configurable) Instance Components	221
5.3	Application Provisioning	225
5.3.1	Find Realizations	226
5.3.2	Find Correct and Complete Component Bindings	236
5.3.3	Select the Realization Component Binding	246
5.3.4	Component Provisioning	248
5.3.5	Executing Provisioning Activities	256
5.4	Optimization of Component Binding	263
5.4.1	Generic Plugin Mechanism for Optimization Algorithms in Cafe	264
5.4.2	Annotating Already Provisioned Components and Provisioning Services with Cost and Performance Levels	265
5.5	Summary and Conclusion	266

6	Implementation of the Cafe Platform	269
6.1	Cafe Application Modeler	270
6.1.1	Cafe Application Model Editor	271
6.1.2	Cafe Variability Model Editor	272
6.1.3	Component Binding Editor	273
6.1.4	Customization Flow Generator	273
6.2	Cafe System	275
6.2.1	Application Portal	276
6.2.2	Customization Flows	278
6.2.3	Template Repository	279
6.2.4	Solution Repository	280
6.2.5	Provisioning Services and Provisioned Components Repository	280
6.2.6	Selection Facility	281
6.2.7	Optimization Component	282
6.2.8	Solution Creation Service	282
6.2.9	Provisioning Support Services	283
6.2.10	Provisioning Flows	283
6.2.11	Component Flows	288
6.2.12	Abstract BPEL Processes as Blueprints for Component Flows	293
6.3	Summary and Conclusion	294
7	Applicability and Case Studies	295
7.1	General Applicability	296
7.1.1	Application Modeling and Deployment	296
7.1.2	Variability Modeling	302
7.2	Ecco	305
7.2.1	Multi-Tenancy Patterns in eCCo	306
7.3	Decidr	307
7.3.1	Components and Provisioning	307
7.3.2	Headless Cafe	309

7.4	Mocca Sample Application	310
7.4.1	Components	310
7.4.2	Variability	311
7.4.3	Provisioning	312
7.5	Bootstrapping Cafe: Cafe System as a Cafe Application	314
7.6	Further Case Studies	315
7.6.1	EaaS	315
7.6.2	Sametime 3D in the Cloud	316
8	Conclusion and Outlook	317
8.1	Conclusions	317
8.2	Future Work	319

ZUSAMMENFASSUNG

Heutzutage versuchen Firmen Anwendungen, die nicht zu ihrem Kerngeschäft gehören, möglichst kostengünstig an Betreiber auszulagern. Dieser Trend hin zum Outsourcing von Anwendungen begünstigt die Entstehung eines neuen Typs von IT Service Anbietern. Diese Anbieter betreiben und warten Anwendungen für Firmen, die diese Anwendungen ausgelagert haben. Das Geschäftsmodell dieser Anbieter basiert auf Skaleneffekten. Da der Anbieter in der Lage ist, die Anwendungen mehrerer Kunden auf der gleichen Hardware-, Middleware-, und Softwareplattform zu betreiben, kann er sie meist kostengünstiger betreiben, als die Kunden selbst. Da verschiedene Kunden verschiedene Anforderungen hinsichtlich funktionaler und nicht-funktionaler Anforderungen der Anwendungen haben, müssen sowohl die Anwendungen, als auch die darunterliegende Middleware- und Hardwareinfrastruktur flexibel und anpassbar sein. Um ihren Kundenstamm zu erhöhen und Skaleneffekte noch besser auszunutzen, werden nicht nur eigene Anwendungen von den Anbietern angeboten, sondern auch Anwendungen, die von Drittanbietern zugekauft werden.

In dieser Dissertation wird ein Metamodell sowie zugehörige Algorithmen und Werkzeuge eingeführt, das es Anwendungsentwicklern erlaubt, Anwendungen so zu entwickeln und auszuliefern, dass sie automatisch bei einem Anbieter installiert (provisioniert) werden können. Dies sollte möglichst wenig

bis gar kein Eingreifen von Administratoren erfordern. Darüberhinaus wurde ein Metamodell entwickelt mit dessen Hilfe die Variabilität in Anwendungen beschrieben werden kann. Die Definition der Variabilität einer Anwendung mit Hilfe dieses Metamodells erlaubt die automatische Generierung von Werkzeugen, die Kunden durch die Anpassung von Anwendungen leiten. Daher unterstützen die in dieser Arbeit eingeführten Technologien ein sogenanntes Selbstbedienungsmodell, in dem Kunden sich in einem Anwendungsportal für Anwendungen anmelden und von ihnen abmelden können, je nachdem, ob sie eine bestimmte Anwendung benötigen oder nicht. Nachdem sie eine bestimmte Anwendung ausgewählt haben, werden die Kunden durch die Anpassung der Anwendung geleitet. Diese wird dann automatisch mit den, vom Kunden gewählten, funktionalen und nicht-funktionalen Anpassungen aufgesetzt.

Die Konzepte, die in dieser Arbeit eingeführt werden, erlauben es Anwendungsentwicklern, anpassbare Anwendungen zu beschreiben und zu entwickeln, ohne die konkrete Infrastruktur, auf der diese Anwendungen nachher betrieben werden, zu kennen. Anbieter können ihren Kundenstamm erhöhen, indem sie anpassbare Anwendungen von Drittanbietern auf ihrer Infrastruktur betreiben. Kunden können mit dem vorgestellten Ansatz eine beliebige Kombination aus Anbieter und Anwendung wählen und sind nicht auf einen Anbieter fixiert, da nur dieser eine ganz bestimmte Anwendung anbietet. Prototypen, die alle drei Rollen, Anwendungsentwickler, Anbieter und Kunde unterstützen und die die beschriebenen Konzepte implementieren, werden in dieser Arbeit vorgestellt. Darüberhinaus wird der Ansatz in verschiedenen Fallstudien evaluiert.

ABSTRACT

The trend to outsource applications not critical to an enterprise's core business has driven the emergence of a new type of IT service providers. These service providers run and maintain applications for enterprises having outsourced them. The business model of the providers is thus based on the exploitation of economies of scale by offering the same infrastructure, platforms and applications to multiple customers. Since different customers have different requirements regarding functional and nonfunctional aspects of an application, the infrastructure, platforms and applications must be customizable to different customer's needs. This also enables the providers to increase the customer base for one offering. To further increase the customer base, providers do not only host and provide software they have developed in house, but also want to offer applications offered by third-party application vendors. In this thesis, a metamodel, algorithms and tools are introduced allowing application vendors to describe and package applications in a way that they can automatically be set up (provisioned) at a provider either with minimal or no human intervention. Furthermore, a metamodel is introduced allowing application vendors to describe the variability of an application. This variability metamodel enables the generation of customization workflows used by customers to adapt an application to their needs. The combination of the application metamodel and variability metamodel enables a self-service model in which customers can

subscribe to and unsubscribe from applications as they wish. Customers thus select the application they want to use in an application portal and a guided through the customization. Having customized the application, the application is set up automatically with the required quality of services and functionality.

The concepts introduced in this thesis enable application vendors to describe customizable applications without knowing the exact infrastructure they are later provisioned on. Providers can extend their customer base by offering customizable applications developed by application vendors, and customers can follow a best-of-breed strategy in choosing from arbitrary combinations of providers and applications.

Corresponding prototypes for all three roles, application vendors, providers and customers to build, provide, provision and customize applications have been built and are introduced in this thesis. These serve as a proof-of-concept implementation for the proposed concepts. Different case studies are given to show the general applicability of the approach.

ACKNOWLEDGEMENTS

Without the help and support from many people this thesis would not have been possible. First and foremost I want to thank my supervisor Prof. Dr. Frank Leymann at the Institute of Architecture of Application Systems of University of Stuttgart for his continuous support and enthusiasm regarding all aspects of this thesis. I also want to thank Univ-Prof. Dr. Schahram Dustdar of the distributed systems group at TU-Vienna who acted as the academic co-referee of this thesis. Thank you very much for the discussions we had at various conferences and the support by you and your team at TU-Vienna.

During my time at the IAAS I met a bunch of wonderful, bright and supportive people of which some have become good friends of mine. I especially want to thank my long-time office-mate Thorsten Scheibler for the pleasant time and support in all the tasks we had to fulfill. I enjoyed the time with you.

I also want to thank Dimka Karastoyanova for her advice and support regarding different aspects of my thesis and for her friendship. I also want to speak out a big thank you to Tobias Unger with whom I wrote several research papers for the fruitful discussions and the willingness to dive deep into the internals of my thesis. Thank you! I also want to thank Klaus Pohl and his team at the University of Duisburg-Essen for giving me insights into software product line engineering. A special thank you goes to Andreas Metzger with whom I worked on the combination of software product lines and Cafe. Thank

you very much for your support regarding this matter.

Thanks also go to various other colleagues at IAAS, most notably Tammo van Lessen, Jörg Nitzsche, Daniel Martin, Daniel Wutke, Daniel Schleicher, Tobias Anstett, David Schumm, Oliver Kopp, Anja Monakova and Matthias Wieland with whom I had interesting discussions on all things service-oriented. Thank you not only for your insight into technical details but also for the great times we spent together outside the institute. Also a big thank you to Daniel Martin, Daniel Wutke and Tammo van Lessen and Oliver Kopp for the preparation of and help with the Latex template for this thesis.

Thanks also go to several people at IBM research, for the discussions on service management and related approaches. I also want to thank Gerd Breiter and Michael Behrendt at IBM Böblingen for discussions opening my view on what was really needed in industry.

Last but not least I want to thank those people that made all this possible: First of all my parents, Monika and Hans who always supported me during school and university and who have made it possible that I could do this thesis. Thank you. A big thank you also goes to Anni, who had to suffer through a lot of weekends spent writing this thesis and weeks that I spent on conferences. She supported me all the time and brought me back on track when motivation was low. Thank you Anni for your support and love!

INTRODUCTION

1.1 Introduction, Problem Domain and Motivation

Enterprises today are faced with various challenges relating to their operational flexibility. In dynamic markets the ability to bring products to market rapidly and efficiently is crucial for the success of an enterprise. This rapid change and the introduction of new products require the business processes in an enterprise to be highly flexible. Today, business processes are often supported by corresponding IT-infrastructure. The IT-infrastructure should support the business processes, making them more efficient and less error-prone. However, the flexibility of business processes, and therefore the flexibility of an enterprise, is often hampered by the rigidity of the IT systems supporting the business processes. Even small changes in applications result in long implementation times, as IT departments are often understaffed, underfunded and therefore inflexible. Thus, IT is often seen as one of the main inhibitors of change in an enterprise [Car04].

As a consequence enterprises seek to reduce their IT costs by contracting parts of their IT to service providers. This *outsourcing* of (parts of) the IT is a common way to reduce capital expenditure (capex) and move costs to operational

expenditures (opex) as well as to concentrate on the core competences of the company [HLW94, MMB95, HH00].

Driven by the need of companies to outsource IT-infrastructure and even entire applications, different delivery models for software and the necessary infrastructure to run it has been emerging. One model that emerged is the *application service provider* (ASP) model [Tao01, Dew01, CS01]. In the ASP model a company outsources the entire application stack ranging from hardware, over middleware to software to the ASP [Tao01]. The ASP runs such a complete stack separately for each corresponding customer. In addition to providing the necessary hardware and software, the application service provider also provides the maintenance of the outsourced application.

Besides the ASPs, hosting companies emerged offering companies and end users hosting of Web sites and later Web applications on their servers.

Grids [FK04] as cross-institutional networks providing compute power and storage capacity, emerged. Grids allow defining *virtual organizations* that share computing resources. These computing resources can then be allocated for limited time by one of the partners often to execute a specific experiment. Similar concepts have been developed in the enterprise computing community under the name of *utility computing* [Par66, Rap04]. The vision of utility computing is the vision of providers offering compute resources in a similar manner to the way electricity or water providers (also called utilities) offer electricity or water. Customers can then use computing *on demand* [Fel03], i.e. when they need as well as in the quantity and the quality they need IT resources to perform a certain task in a business process.

One of the first and most popular offerings bringing the utility computing paradigm to end-users is Amazon's elastic computing Cloud (EC2) [Ama09, Vog08]. EC2 is an offering allowing customers to create arbitrary application stacks in the form of *machine images* and start and stop instances of these images on demand. S3 is an offering that customers can use to store data in Amazon's data center. Customers only pay for the compute power and storage they actually use. Two terms, *Infrastructure as a Service* (IaaS) and *Platform as a Service* (PaaS) have been coined to distinguish the degree up to which IT-infrastructure is provided. In the IaaS model only the lower layers (IT

resources such as storage or complete servers including the operating system) are supplied by the provider, while in the PaaS model complete application platforms are provided.

In parallel to the utility computing community - which aims at providing IT resources to be used to run other applications on top of it - another community has emerged. Driven by the drawbacks of the application service provider model, the Software as a Service (SaaS) [TBB03, Ma07, Nit09] model began to gain widespread acceptance. In the Software as a Service model, a provider offers the complete application stack to be used by his customers. Customers can sign up and sign off from an application on demand [DW07]. Typically customers can adapt certain (mostly functional) aspects of the SaaS application [Nit09] to their needs. How the application is implemented and details about the underlying infrastructure are transparent to the user. The main difference between the ASP model that also provides a complete outsourcing of the application stack is that the SaaS provider aims to host multiple customers on the same instance of an application. The main problem arising from hosting multiple customers (also called *tenants* in this context) on the same application stack is the problem of isolation of tenants. The application must therefore be *multi-tenant aware* [GSH⁺07, Ham07]. Multi-tenant aware means that the application must behave for each tenant as if this tenant was the only tenant of the application. In particular, data of one tenant must not be accessible to other tenants. In general, isolation requirements apply also to service level agreements (SLAs) [SAB⁺07]. SLAs, like data-privacy must be ensured on a per-tenant basis and tinkering with the software by one tenant must not influence the service levels of the software for other tenants.

Recently, these efforts to provide and use scalable IT-environments in a utility model has been summarized under the term *Cloud computing* [HAY08, LKN⁺09, YBDS08, Ley09]. The Cloud has been named after the shape in which the Internet often appears in architecture diagrams. Cloud computing incorporates as a service models such as SaaS, PaaS and IaaS as well as combinations thereof. Cloud computing is characterized by easy-to-use interfaces as opposed to Grid computing where adoption outside the academic community was often hampered by overly complicated models and interfaces [JMF09].

With the acceptance of Cloud computing and the underlying delivery models, more and more providers now offer infrastructure, platforms and services usable by application developers to build, host and compose applications.

Orthogonal to the advent of new delivery models, software engineering has moved away from building monolithic *application silos* that are hard or impossible to integrate with other applications and that replicate functionality found elsewhere. Component-based software engineering [BW96] proved successful in reusing components in new applications. Service-oriented architecture (SOA) as an architectural style emerged to deal with integration problems in and across enterprises [Pap03]. In a service-oriented architecture, business functions are realized as a shrink-wrapped self-contained piece of software – a *service*. Services can be recursively composed into higher-level services until the necessary functionality is achieved.

In this thesis utility computing and composite applications are combined. This combination enables the composition of *composite applications* out of components hosted at different providers in possibly different delivery models. The composition itself can be hosted again at a possibly different provider in a possibly different delivery model. Customers can then use these applications on demand without the need to provide their own infrastructure. Since the approach in this thesis focuses on composite applications, customers can plug-in their own services into applications hosted at one or several providers.

The method and implementation to define and provision these composite applications presented in this thesis has been named *Composite Application Framework* (Cafe for short). The vision of Cafe can be summarized in Figure 1.1. Cafe providers offer applications built by application vendors to customers using an *application template catalogue*. Customers interested in using an application offered at a provider then select an application from the *application portal* at a Cafe provider (Step 1). Next, the customer customizes the required quality and functional properties for the selected application (Step 2 and 3). These properties might be performance, scalability or other quality parameters as common in *Service Level Agreements* (SLAs) or functional aspects of the application. This includes customizations of the graphical user interface, customization of the business processes and customization of the data layer of the

application. Having done so, the provider derives the necessary infrastructure corresponding to the customized application (Step 4). The customized application is then set up at one or multiple providers' sites (Step 5). The customer can then use the application. Step 4 and Step 5 are collectively referred to as *provisioning* of the application and required infrastructure.

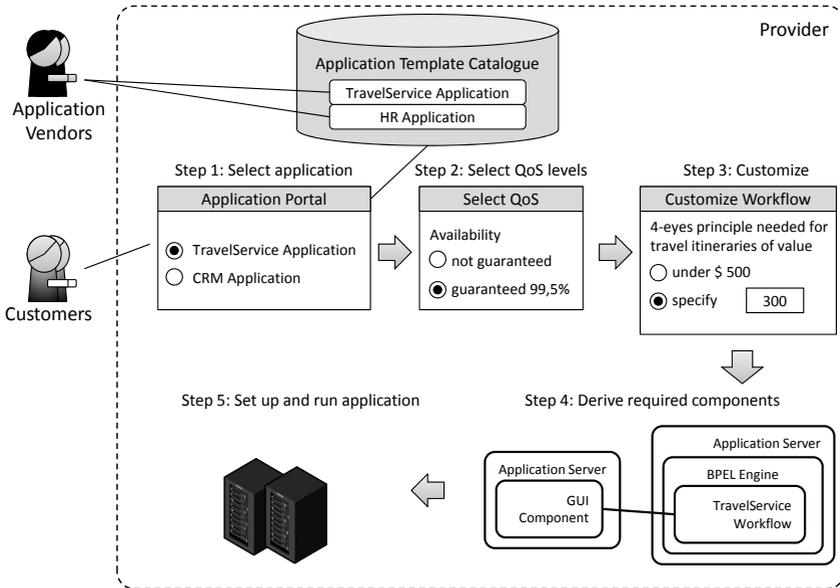


Figure 1.1: Cafe vision

1.2 Research Issues and Contributions

In the following section key research issues regarding the Cafe vision are described.

1.2.1 Application Metamodel and Component Dependencies

Cafe applications are modular applications that can be composed out of a set of components. When modeling and implementing such an application,

it is not known a priori on which provider individual components are later run. Therefore, it is needed to model and describe components and their dependencies on provider infrastructure components in a provider-neutral manner. This also includes the description of components that need to be supplied by a provider, such as servers or a database management system.

Contribution 1: A provider-neutral application metamodel enabling the modeling of components of an application, their dependencies among each other and their dependencies on the required provider infrastructure. Furthermore, a corresponding serialization format (*application descriptors*) is defined.

1.2.2 Separation of Application Vendors and Providers

One of the main unsolved problems hampering the adoption of Cloud computing is the vendor lock-in that customers currently experience when selecting Cloud providers [Dri09, Ope09, DKM⁺09, AFG⁺09]. Today, a lot of applications offered in the Cloud are provided by the same vendor that also implemented them. Examples of this are the Customer Relationship Management (CRM) application offered by Salesforce [Sal09b] or the office suite offered by Google Apps [Goo09a]. Customers therefore are dependent on this vendor, since they cannot obtain the same application from a different provider.

As a result of not being forced to stay with a single provider, a separation of providers and application vendors allows customers to follow a best-of-breed strategy on both application vendor and provider side when choosing outsourcing partners for their applications.

Also, providers and application vendors benefit from the separation of application vendors and providers, as it enables them to increase their customer base by offering a larger variety of applications at different providers. The consequence of separating providers and vendors is the need for a common format for the description of applications.

Contribution 2: A specification of a self-contained package format for Cafe applications. The package format is independent of the technologies used to implement and provide the applications. Furthermore, the package format together with the application metamodel is the basis for automatic provisioning

of the composite application it represents.

1.2.3 Defining Variability in Composite Applications

One goal for application vendors and application providers alike is to maximize the number of potential customers for their applications. Since different customers have different functional as well as non-functional requirements on an application, applications must be specified, designed, modeled and tested with variability in mind. Variability allows customers to adapt the application and thus it increases the number of customers. Variability affects all layers of the application from the GUI layer down to the business logic and data layers. Additionally, dependencies between different variability points in an application must be specified.

Contribution 3: A specification of a variability metamodel allowing the description of variability points in different components of an application as well as their dependencies. The variability metamodel is generic, i.e. it generalizes individual mechanisms to specify variability within the underlying implementation technologies used for an application. Furthermore, a serialization format for the variability model is defined (*variability descriptors*).

1.2.4 Customization of Applications

Customers selecting an application need to customize it to their particular needs. Customization means making decisions on alternatives for variability points while respecting the mutual dependencies between variability points. To support customers in this complex and error-prone task, tool support for customization is needed. Such a tool must ensure that the result of customization is *complete* and *correct*, i.e. the customized application template can later be transformed into a solution and can be run on the provider's infrastructure.

Contribution 4: Algorithms and corresponding implementations to generate customization workflow support from variability models ensuring *completeness* and *correctness* of the customization.

1.2.5 An Architecture to Automatically Provision and Manage Applications

When a new customer selects an application from an application portal, the required infrastructure must be provisioned at one or several providers so that the application can be set up on this infrastructure. Since different providers use different tools and APIs to set up infrastructures and applications, it is not possible today to describe provisioning scripts in a provider and provisioning-tool independent way.

Contribution 5: An architecture of a unification layer that syntactically and functionally unifies the APIs of different providers and their provisioning infrastructure. This unification layer is the basis for describing provider-independent provisioning scripts for applications.

1.2.6 QoS-Aware Provisioning

As shown in Figure 1.1, users can select the quality of service (QoS) attributes for an application. The provider must guarantee these QoS. Guaranteeing QoS consists of (i) setting up an appropriate infrastructure and (ii) adapting this infrastructure during runtime. While (i) has been solved in this thesis, (ii) is out of scope of this thesis.

Contribution 6: Algorithms and a corresponding implementation to generate provisioning scripts that ensure an appropriate set up of the infrastructure according to the selected QoS.

1.3 Organization of the Thesis

This thesis is organized as follows: Chapter 2 describes the necessary background and foundations of Cafe. This includes a discussion of architectural styles and composite applications, as well as a classification of software delivery models common today. Next, existing systems and related approaches are discussed and an evaluation of the state of the art concerning the requirements from the different stakeholders are given.

In Chapter 3 the development of Cafe application templates is described. The first part describes the overall development process for Cafe applications and

introduces relevant terminology to distinguish the different artifacts produced during different states of the development process. The second part of Chapter 3 introduces a component model for Cafe applications including an XML-based serialization of the component model. The third part of this chapter contains the definition of variability descriptors, a mechanism to annotate applications with variability. The chapter is finished by a description of a package format for Cafe applications containing the application components as well as the application and variability descriptors.

Chapter 4 describes how artifacts realizing customization tools based on generic middleware can be generated from the variability descriptors annotating a Cafe application. This is followed by an operational semantics for the variability descriptors enabling the translation from variability descriptors into a workflow model.

Chapter 5 then introduces an architecture and algorithms for a provisioning infrastructure for Cafe applications. The necessary Cafe provisioning and management infrastructure based on a service-oriented architecture is introduced. The need for a uniform interface for different existing provisioning infrastructures is motivated. It is illustrated how this unification is achieved through a common provisioning and management interface. This *Cafe provisioning and management interface (CPMI)* is illustrated in detail in the second section of the chapter. Having motivated the need for a common API, the algorithms for the generation of provisioning flows for Cafe applications are given. These provisioning flows can then be used to provision a Cafe application. In the fourth section of this chapter the generic plugin architecture and infrastructure is described enabling providers to plug in their own optimization algorithms into the provisioning infrastructure.

Having described the necessary infrastructure architecture we describe the prototypical implementation of the Cafe platform and the modeling tools in Chapter 6 in detail.

The thesis is continued by a validation of the presented concepts in general and in various settings in Chapter 7: The first setting is the automotive point-of-sales application eCCo where some of the concepts of Cafe have been applied to create a process-based sales application for car dealers. The second application

is the DecidR application, a Web-based platform for decision processes. We finish the validation section by describing how Cafe can be used to bootstrap its own platform and show additional use-cases where we applied Cafe to show its general applicability.

The last chapter of the thesis contains a conclusion and an outlook, summarizing the contributions of this thesis and describing how additional research work can enhance the approach and overcome some limitations.

BACKGROUND AND RELATED WORK

In this chapter the necessary background and related work to understand the following chapters is given. It is shown in which situations existing products, prototypes and research contributions must be extended to fully address the research issues given in Section 1.2.

The first section of this chapter deals with basic architectural styles, technologies and methodologies allowing the decomposition of applications into components and thus form the basis of this thesis. Different techniques are shown that support to decompose applications in terms of reusable application components as well as in terms of middleware components. In particular, component-based software development is described as the foundation of modularization of software. The concepts behind service-oriented architecture and one of its implementations – Web services – are introduced and explained to illustrate how applications can be distributed across different providers today. It is shown how component-based software development and service-oriented architectures form the basis for the composite applications that can be mod-

eled and provisioned using the Cafe methodology. Software product lines are introduced as a software-engineering discipline dealing with application families that can be configured differently for different purposes. It is shown how a software product line differs from a Cafe application template that is also annotated with variability.

Since Cafe applications deal with composite applications deployed at various providers in various delivery models, a classification of these delivery models and providers is presented. Similarities and differences of the individual delivery models are discussed. The discussion is done with regard to the aspects of multi-tenancy and outsourcing in the different models.

Then existing systems and related approaches to Cafe, from industry and academia are introduced and compared to the research challenges of this thesis. Having examined the relevant background and related work, a summary is given illustrating the need for the Cafe concepts and implementations given in the following chapters of this thesis.

2.1 Decomposition of Applications into Components

This section contains background on architectural styles and technologies used commonly to decompose applications into individual components. First, component-based software development is briefly introduced illustrating the way how the software-engineering community deals with the decomposition of applications into reusable components. It is shown how component-based software has fostered modularization and how it forms the basis for Cafe. Then, in the next subsection, service-oriented architecture (SOA) is introduced as the latest trend in software architectures. SOA allows to further modularize applications. Web services are then introduced as one specific implementation of a SOA. It is then shown how applications cannot only be decomposed into individual modules on the application level, but also into middleware and hardware components.

2.1.1 Component-Based Software Development

Decomposing software systems into individual modules for flexibility, reusability and comprehensibility [HC01] is a technique long proposed by the software engineering community [Par72b, Par72a, Cle95, BW96, KB98, HC01].

In component-based engineering, functionality is encapsulated in reusable *components*, also called *modules* [Par72a, BW96], used to assemble new applications out of existing components. This reduces the implementation effort for new applications which makes development of component-based applications less error-prone and faster. It also makes programs more understandable, as design decisions that only affect a part of the application can be hidden in a module or component [Par72a]. Commercial *off the shelf* components [Cle95, BW96] then allow to reuse and buy standardized components from other vendors and integrate them into new applications. This component-oriented development principle is based on the *reuse* of components. Reuse in this context indicates that the code is reused, thus the code is duplicated for every use of the component [LW05]. In case a component is contained in multiple applications, multiple instances of the code of the component are spread across these applications. Distributed component frameworks [KA98, PS98, Lau06], such as Microsoft's distributed component object model (DCOM) [Ses97], Sun's Enterprise Java Beans (EJB) [Ham97], or the Common Object Request Broker Architecture (CORBA) [V⁺97] and others [KB98, Lau06] enabled the building of distributed component-based applications built out of components across multiple machines and even across different organizations. These distributed component models require *middleware services* [Ber96] to run. These middleware services offer standardized services for the individual component frameworks [CLWK00] encapsulating often needed functionality such as, for example, the remoting capabilities of the framework enabling the distribution of the components. Software developers can then make use of these middleware services to facilitate application development. To run an application making use of these middleware services, they must be available and the application components must be deployed on these middleware services.

In addition to the distributed component frameworks several approaches

emerged describing components as reusable assets, most notably OMG's reusable asset specification (RAS) [OMG05a]. RAS allows to specify reusable assets along with dependencies on other assets and a general framework for describing the customizability of these assets.

Component-based software engineering constitutes an important enabler for Cafe, as it allows the modularization of applications into independent components that can then be reused either by copying the code of a component into a new application or by reusing already deployed components remotely. Cafe applications supports both models, i.e a Cafe application can include the code of reusable components or can make use of already deployed remote components.

2.1.2 Service-Oriented Architecture

Service-oriented architecture (SOA) [Erl05, Pap03] has been emerging as an architectural style to build applications out of existing building blocks, called services. Services encapsulate business functions prone to be reused. In contrast to object-oriented approaches, services have an *always on* semantics. Services therefore do not need to be instantiated before they can be used. Similar to services provided by utilities such as gas, electricity or water, services in a service-oriented architecture are simply there to be used. New applications can then be assembled by building new services and recursively composing new and existing services into higher level applications which then again can be provided to other applications as services. Services that cannot be decomposed further (or whose composition is hidden) are called *atomic services*. Services that are composed out of a set of other services are called *composite services*.

Services are characterized by a well-defined interface describing the functional aspects of a service. This interface is then implemented in any programming language. In addition to the description of the functional aspects of a service, the non-functional properties of a services can be described using *policies*.

Services are *published* by service providers in service catalogues [BCE⁺02, CDK⁺02] and can be discovered using functional and non-functional descrip-

tions of the desired service. Once a service has been discovered during this *find* step, a service consumer can *bind* against this service and use it. Binding can be *static* at design-time, or deployment-time. I.e., an application developer or deployer queries a service catalogue for available services and assembles them into a new application. Binding can also be *dynamic* at runtime where a *service bus* [Cha04] discovers available services based on their functional and non-functional descriptions when they are needed [KAB⁺04].

One specific property of a service-oriented architecture relevant for this thesis is that the architectural style does not make any assumptions on the implementation technology for services [Erl05]. Furthermore, an inherent property of a SOA is that services can be distributed across different machines, enterprises and continents [WCL⁺05, PG03]. This geographical and organizational distribution allows to assemble composite services out of (atomic and composite) services residing in any arbitrary setting around the globe.

In a service-oriented context *composite applications* are thus defined as applications that are composed out of a set of services. These services can be distributed across different machines or even providers. A composite application can be offered as a service again.

Web Services. The *Web service stack* [WCL⁺05] is a set of technologies and specifications that together define the technology for an implementation of a SOA. Services are called *Web services* in the Web service stack. Their interfaces are described using the *Web service description language* (WSDL) [W3C01]. Non-functional properties of Web services are described using a standard called *Web Service Policy* (WS-Policy) [W3C06]. The standard in the Web service stack for service catalogues is *Universal Description Discovery and Integration* (UDDI) [WCL⁺05].

Web services allow to implement service-oriented architectures on heterogeneous distributed infrastructures. Several frameworks exist to implement Web services using different programming languages such as Java, C#, C, PHP, and others. To be able to exchange messages between services in heterogeneous distributed environments, a messaging architecture is needed abstracting from the idiosyncrasies of different programming languages, hardware and middle-

ware platform. The *SOAP* [WCL⁺05] standard is such a messaging architecture. SOAP defines a format for XML-based messages as well as a processing model how such messages need to be dealt with.

2.1.3 The Role of Middleware in Composite Applications

With the increasing complexity of (distributed) applications, the role of the middleware on which these distributed applications are executed, becomes more and more important [Cha99, Emm00]. A wide range of middleware products is available today such as database management systems, application servers, transaction processing systems, message oriented middleware, workflow management systems etc. focused on providing middleware services for different aspects of a (distributed) application such as the data layer, transaction processing, messaging, clustering, distribution and workflows. Thus, applications are not only decomposed into different components forming the application, but also into middleware components providing important functionality for an application to be executed. Going one step further, to run an application not only middleware and application components are needed but also hardware with corresponding operating systems on which middleware and application components can be deployed. Similar to the components of an application modeled using, for example, UML component diagrams [OMG05b, LYC⁺98, CD01] or ACME [GMW00], the deployment relationships between applications, middleware and hardware can be modeled using UML deployment diagrams [OMG05b, Kob00]. Since Cafe focuses on applications that can be accessed via a network and are possibly distributed across different providers, these applications most likely require various middleware components to run. Even if they do not require middleware components, they require hardware and operating systems that are not part of the application to run on. Thus Cafe must support both modularization techniques: *horizontal* decomposition into application components, and *vertical* decomposition into application, middleware and hardware components.

2.1.4 Software Product Line Engineering

Software product line engineering is a discipline from software engineering aiming at reducing cost, speed up time-to-market and improve quality of software by modeling and developing similar software systems as a *software product line* [Par76, Boe99]. In software product line engineering the development process of an application is split into two phases [WL99, vdL02, PBvdL05]: The *domain engineering* phase where the core application, including the variability, is developed. The core application is also called *software platform* [ML97, PBvdL05]. The second phase is the *application engineering* phase where the software platform is customized to individual customer's needs, the customized platform is then called *application*. This is similar to the requirements for Cafe presented in Section 1.2. In Cafe an application vendor provides a *template* that roughly corresponds to a software platform. Customers can then customize the template to a *customer-specific solution* which corresponds to an application in software product line engineering.

However, the Cafe approach differs from a software product line in several aspects, the first aspect is the organizational structure:

In [Bos00a, Bos00b, Bos01, PBvdL05] different organizational structures for software product line are investigated and evaluated. However, all these structures have in common that both, the domain engineering and the application engineering are performed by the same organization. In the Cafe model the customization of the template is done by the end user which differs from the presented approaches for software product-line engineering. This implies that in product line engineering the variability in the product line is bound by a member of the same organization that has built the software

The second aspect is the definition of variability: In software product line engineering one approach to describe variability is to integrate the definition of variability in the respective development artifacts such as use case models [HP03], feature models [KLD02, FFB02] or implementation artifacts [BFG⁺02, VGBS01, vdMLA02]. A classification of such variability modeling techniques can be found in [SD07].

Approaches to define variability in business processes fall into the category

of variability models where variability is integrated into the development artifacts. WS-BPEL, for example, [OAS07a] allows specifying *abstract processes* containing *opaque tokens* that must be customized before the process can be deployed [MML08]. Other approaches, such as VxBPEL [SA08, KSSA09] or the approach presented in [LL07] extend WS-BPEL with a more sophisticated variability mechanism than the abstract processes. The configurable EPC (C-EPC) approach [RvdA07, RRVDAM06] includes variability in event-driven process chains (EPCs). The Provop approach [HBR08] focuses on the management of variants in process models by providing a set of operations (such as insert, delete, move or modify) to change process elements. The PESOA project [SP06] extends BPMN [OMG09] with the ability to model variants of a process model. Other approaches [GWJV⁺09, LRLS⁺07, vdADG⁺08] focus on configurable process models and how to preserve correctness of the process model during configuration. However, all these approaches are focused on the inclusion of variability on the process layer or even one particular workflow language ([SA08, KSSA09, LL07, GWJV⁺09, LRLS⁺07, vdADG⁺08]). For Cafe this is not enough as the variability must be on the application level and thus variability of the process layer is only a subset of the overall variability.

The problem of an approach where variability is integrated in the development artifacts is that variability is scattered across different artifacts. This makes the management of variability very complex. Additionally, as different artifacts are annotated with different variability models, describing dependencies between variability in different artifacts becomes hard [PBvdL05]. Therefore, *orthogonal variability models* such as the OVM (orthogonal variability model) [MHP⁺07, PBvdL05] have been introduced in the software product line engineering domain. The advantages besides documenting the variability in a central model are the reduction of complexity and size of the variability model, as only the variable parts are documented in it and not the common parts as they are in an integrated variability model.

For Cafe an orthogonal variability model is beneficial, since Cafe does not assume any specific implementation language and thus cannot rely on the fact that any possible implementation language offers an integrated variability mechanism. In addition, a mechanism to describe dependencies between the

variability of different artifacts is needed, i.e. variability in a process model can have effects on the GUI or even the deployment of additional services.

In [MMLP09] we show how OVM as an orthogonal variability model can in principle be used to model variability for SaaS applications. However, OVM has several drawbacks. First of all, it is focused on modeling variability for software product lines. However, one requirement for Cafe is that the variability is defined in a way end users can bind it and that the binding of variability results in executable code which is not the focus of OVM. Additionally, the customization of Cafe applications requires that during the customization not only pre-defined variants can be bound, but also free values can be entered. This is, for example, important if an end-user should be able to change the title of a Web-page. These *free alternatives* where end-users can enter free text or values are not supported in OVM. Therefore, a new orthogonal variability model is introduced in Section 3.4 that is similar to OVM but is geared especially towards Cafe applications. The new variability model includes free alternatives, pointers into executable code and arbitrary expressions to evaluate if an alternative can be bound or not, which is also not possible in OVM.

The third aspect in which Cafe differs from a software product line is the aspect of multi-tenancy. In a software product line each member of the product line contains one specific binding of the variability [PBvdL05, MMLP09]. In Cafe and SaaS applications a multi-tenant aware component can contain multiple different bindings of the variability at once for different tenants. For example, the background of a Web application can be red for one tenant while it is blue for another tenant and both are served from the same instance of the code. In Figure 2.1 this situation is depicted where two customers of a software product line are served by individual applications which are members of the software product line. The customers of a Cafe application can additionally be served by one instance of the application which has different customizations.

Additionally, the environment in which a member of the software product line is deployed later is known during the application development phase; therefore environment variability (i.e. variability related to a certain middle-ware component or operating system) can be bound during creation of the product line member. In Cafe applications this variability may be bound at

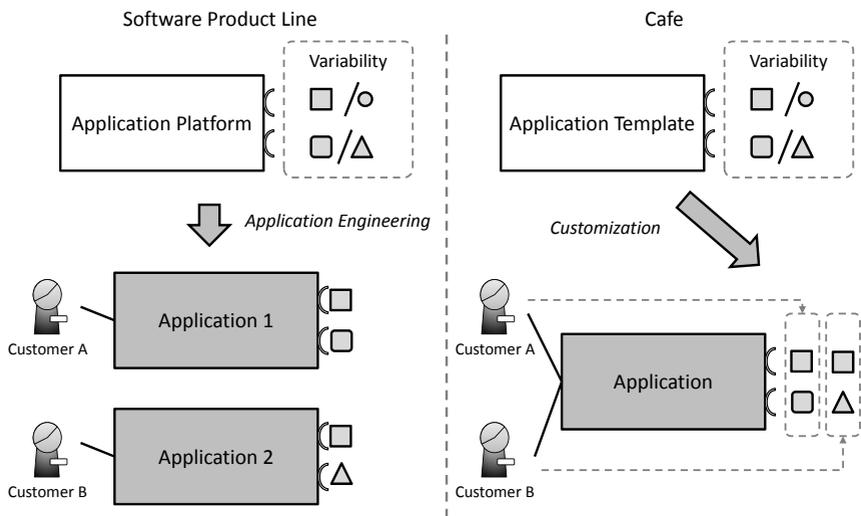


Figure 2.1: Two customers of a software product line, vs. two customers of a Cafe application

deployment or even runtime, as the concrete environment on which the tenant is deployed is not known beforehand.

2.2 Towards Utility and Cloud Computing

Cloud computing [HAY08, LKN⁺09, YBDS08, Ley09] has emerged lately as a new paradigm how to build and deliver IT services. Different definitions for the term *Cloud computing* exist [Gee08, VRMCL08, JMF09, HAY08, Vou08, FZRL08, BYV⁺09]. These definitions differ in various aspects such as the abstraction of interfaces or the pricing schemes. However, the open cloud manifesto [Ope09] defines a minimal set of principles that seem to be agreed upon in all the definitions. The key characteristics of the cloud according to [Ope09] are “*the ability to scale and provision computing power dynamically in a cost efficient way and the ability of the consumer (end user, organization or IT staff) to make the most of that power without having to manage the underlying*

complexity of the technology.”

This vision of on-demand usage of resources by users of the cloud follows the vision of IT as a *utility*. The *utility computing* [Rap04] paradigm aims at providing a *pay as you go* payment model for computing resources where computing resources can be provisioned *on demand* [Fel03]. The application model presented in this thesis follows this pay as you go paradigm and aims at providing applications on demand, i.e. end users can subscribe and unsubscribe from applications as they wish and these applications are then automatically provisioned and deprovisioned as needed. Another paradigm relevant for cloud computing is *autonomic computing* [KC03, ST05] where systems manage themselves according to predefined goals.

In computing intensive environments such as scientific applications and banking *Grid Computing* [FK04] has emerged as the computing paradigm to share computing resources such as servers and storage across different institutions in *virtual organisations* [FKT01]. Grid environments, however, are very generic and offer very complex interface hindering the adoption outside the established environments [JMF09, FZRL08, MKL09].

With the emergence of service-oriented architecture, grids and utility computing models, the outsourcing of parts of applications becomes easier as, for example, Web Service technology allows the distribution of the services of an application across multiple providers. Thus providers now offer services fulfilling certain often-needed functionality that can then be reused in applications by the clients of the provider *as a service* via standardized APIs.

2.2.1 As a service models

The *as a service models* differ in the degree up to which the application stack consisting of hardware, middleware and software components is outsourced [LKN⁺09, ALMS09], and which parts are shared between different customers and which parts are specific to a customer. *Software as a service* (SaaS) [TBB03, GSH⁺07] providers offer software that can be used by their customers without the need for a customer to care about the management of the application. The SaaS model is an evolution of the application service provider (ASP)

model [Tao01]. Providers not only offer services but also middleware and development platforms in the *platform as a service* (PaaS) model [Law08] and hardware and storage in an *infrastructure as a service* (IaaS) [Vog08] model. These platforms and infrastructure can then be used by customers to deploy applications on it.

In the following the two main aspects, outsourcing and multi-tenancy are investigated for each of these as a service models. Then existing examples for these models are given and it is investigated whether these examples, and the as a service model they represent, are suitable to deal with the research issues in Cafe as shown in Section 1.2.

Outsourcing. In the following the term *outsourcing cut* is used to denote where the cut between outsourced components and components that are taken care of by the client on premise, is made. The term outsourcing denotes that another organization or organizational entity is contracted to buy, run and/or maintain a particular component in the application stack. This organization or organizational entity can be a dedicated IT service provider or can be in the same organization as the client. Figure 2.2 shows which parts of the application stack are outsourced in which model. In the traditional on-premise model the entire application stack is run under the accountability of the company wanting to use the application. The ASP model and the SaaS model are the complete opposite, as the entire application stack is outsourced. Depending on the concrete definitions varying with every vendor, the IaaS model either provides bare infrastructure (in terms of virtualized servers) or infrastructure containing an operating system (such as basic Amazon machine images in Amazon EC2). Since the difference between IaaS and PaaS is again not commonly accepted, the following definition is assumed in the rest of this thesis: IaaS virtualizes above the hardware/virtualized server level, whereas PaaS provides additional middleware components such as application servers, databases and authentication services. PaaS therefore makes the outsourcing cut above the middleware, whereas some commonly used application functions such as authentication or logging might already be part of the platform.

Outsourcing can be transitive, i.e. a company wishing to use an application

can outsource it to an SaaS provider who in turn outsources the necessary infrastructure to run the application to an IaaS provider. Thereby the SaaS provider acts as a client of the IaaS provider and a provider for the company using the application.

Multi-Tenancy. The second criterion after which the different delivery models are classified, is the *multi-tenancy* cut. This cut shows which part of the application stack is run in multi-tenant mode (i.e. one instance is offered to multiple tenants). By one instance, one logical instance is meant that can be clustered for performance or availability reasons. Natively, the on-premise delivery model is not multi-tenant aware. An application is typically provisioned and run for one client on this client's premise. The same holds true for the ASP model in which the entire application stack is separately provisioned for each customer. In the SaaS application the application itself is multi-tenant aware and the underlying infrastructure is shared by all tenants.

Again, as with the outsourcing cut, the IaaS and PaaS models are situated in between. The multi-tenancy cut is made here on the hardware and middleware. In the IaaS model several tenants are spread over a common infrastructure often through virtual servers that can be shifted and moved between the actual hardware to master varying load demands. In the PaaS model virtual infrastructure is offered to tenants, i.e. a DBMS or a business process engine is shared between multiple tenants. The underlying infrastructure is shared between the tenants too. In the SaaS model the entire application stack up to the application is shared between the tenants.

Figure 2.2 shows the different delivery models and the grade to which their application stack is outsourced and natively multi-tenant aware. Note that different providers can combine different delivery models, thus a SaaS provider can provide a SaaS application on the infrastructure of an IaaS provider.

The analysis of the different outsourcing and multi-tenancy cuts shows the need of a model for composite applications that can possibly be hosted on any of these as a service models. Such a model must include the capability to model the different levels of the application stack, as well as which parts can be shared with other customers and which parts can be outsourced. In the

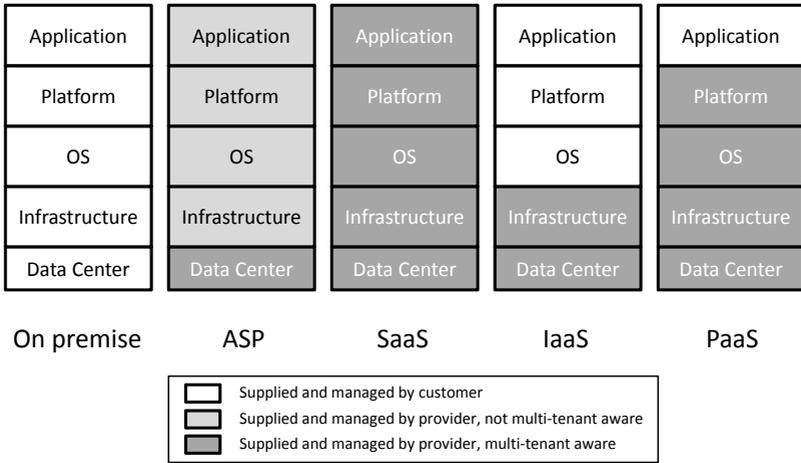


Figure 2.2: Classification of as a service models

following, examples for the individual as a service models are given and it is investigated how and if these examples can be included as providers in Cafe applications.

2.2.2 Example SaaS Application

In this subsection, the Salesforce CRM application [Sal09b] is described, as it is a typical SaaS application that is well-known and can be easily integrated as a component in Cafe applications via its Web service interface. In addition, Salesforce not only offers their CRM as a SaaS application but it can be extended by building components for the proprietary Force.com platform. Other SaaS applications, such as SAPs business by design [SAP09] or teleconference applications such as Cisco’s WebEX [Cis09], or more consumer oriented office applications such as Google Apps [Goo09a] and many more also exist. However, to understand the examples in later parts of the thesis, the Salesforce application is briefly introduced here.

Salesforce. Salesforce offers a CRM (customer relationship management) product in a software as a service model [Sal09b]. Tenants can subscribe to

this application and can instantly use it. Nevertheless Salesforce allows the customization of the application by allowing users to customize data fields, workflows and other aspects of the application. A tenant can customize various aspects of the application from data fields to workflows. Salesforce is a good example of a multi-tenant aware application, as Salesforce explicitly serves multiple tenants from the same instance of the application.

To integrate the Salesforce application with other applications, a Web service interface (described in WSDL) is provided. The Web service interface is offered in two variants, the *partner WSDL* being the same for all tenants and the tenant-specific *enterprise WSDL*. The enterprise WSDL contains the customizations a tenant has made (i.e. workflows, different data schemas etc.) whereas the partner WSDL is generic for all tenants and can thus be reused by another multi-tenant aware application offered for multiple tenants of the Salesforce application.

Customers can extend the Salesforce platform by building applications on the force.com platform. These applications can be tenant specific or multi-tenant aware so that they can be used by other tenants too. The force.com platform offers all necessary middleware and integration technology to run applications integrating with the Salesforce application. Therefore, the force.com platform is a good example for a platform offered as a service (PaaS).

2.2.3 PaaS Examples

Platform as a service is a delivery model for software where a provider provides a platform on which clients can deploy their own code using the platform. Common platform services [GSH⁺07] include security and authentication services as well as other middleware services such as databases, workflow engines or portals. Often PaaS providers also offer programming support through special libraries and development environments.

Microsoft Azure. Microsoft's Windows Azure services platform offers Microsoft .net related services in a PaaS model [Cha09, Mic09]. These services are clustered in five categories. The first category (“.Net Services”) offers access control, service bus and workflow services built upon Microsoft's .net framework.

The second category (“SQL Services”) offers highly scalable data services built upon the SQL Server database management system.

The third category (“Live Services”) offers services geared towards the creation and deployment of social Web 2.0 applications. These services include Mesh services (for data synchronization across different devices and applications), identity services, directory services, user-data storage services as well as geospatial services and search services. Other services offered are communications and presence services that can be used by customers to communicate among each other using messaging systems.

The fourth category of Azure services (“SharePoint Services”) offer access to SharePoint capabilities in the Cloud.

The fifth category of Azure services (“Dynamics CRM Services”) will offer access to functionality of Microsoft’s Dynamics CRM system that are reusable in third-party applications deployed on-premise or in the Cloud, in contrast to the other categories this category does not fall into the category of PaaS offerings but is a SaaS offering as a whole application is offered to the customers.

In summary Azure can be seen as a PaaS offering with some SaaS parts, providing the functionality of certain Microsoft applications in the cloud. Developers can use these platform services to built applications upon these services. However, applications built upon the Azure platform depend heavily on this platform, as there are no other PaaS providers that, for example, provide similar “Live services”. Another disadvantage is that the platform is centered around the .net framework and other Microsoft products. There is, for example, no possibility to run JEE applications on the Azure platform.

Force.com. Force.com [WB09] is Salesforce’s platform as a service offering. Force.com allows users to create applications based upon the Salesforce platform. These applications can either connect to the Salesforce application and thus extend the Salesforce application with custom functionality or can be stand-alone. Through the force.com platform third party providers can offer their applications that are then hosted on force.com’s cloud infrastructure similar to the Salesforce application itself. Currently over 800 applications from third party providers have been developed for the force.com platform.

However, again as with the Azure services platform the force.com platform is a proprietary platform. Currently only Salesforce is offering the force.com platform. Additionally, the force.com platform is built around the proprietary Apex language that can only be executed on the force.com platform. Other programming languages commonly used for enterprise applications such as BPEL, Java, or C# are not supported. Apex code is executed in what Salesforce calls a “virtual, virtual machine” separate to each tenant. Therefore, the Apex code is by default multi-tenancy aware and the code from one tenant does not interfere with the code from other tenants as the virtual, virtual machine also provides, for example, data isolation.

Google AppEngine. Google’s AppEngine [Ciu09] is one of the most-cited examples of a PaaS offering. The AppEngine platform consists of a set of services users can employ to build their own applications. These services include data services (i.e. a block-structured database), UI services and authorization services (linked to Google’s own authorization system). Google’s AppEngine allows users to create new applications in the Python or Java programming language. To create these applications, developers can make use of the platform services provided by Google while they are free to program their applications in the Python and Java programming language. However, both, the Python and the Java AppEngine do not allow certain operations to be called, such as IO operations or multi-threading which makes the AppEngine a restricted Python or Java environment.

Since AppEngine applications are written in the Python or Java language they can be ported to any environment supporting this programming language. However, the platform specific services of the AppEngine might not be available on other platforms. To enable testing on local machines, Google supplies an AppEngine SDK including a limited version of the AppEngine that can be installed locally and provides implementations for all APIs that users can use in the real system. There are some installations of this limited AppEngine at Amazon EC2. This setup allows users to transfer their AppEngine application from Google’s servers to their local data center or EC2, however, without exploiting the elasticity of the Google environment.

Mashup Tools. *Mashup tools* have emerged as Web-based tools enabling end-users to rapidly compose new small Web applications out of existing Web applications [YBCD08, Mer06]. Several mashup tools exist on the market today. Examples for Mashup platforms are Yahoo Pipes [Fag07], Microsoft Popfly [Gri08] or Google's Mashup Editor [Goo09b] among various others, although the latter two examples seem to loose support.

Despite the similarity of the different mashup platforms, there is no standardized package format for these mashups. Mashups cannot be transferred from one platform to another, introducing a dependency on one of the providers. There are typically no guaranteed service levels offered by the mashup providers. In addition, mashup environments focus on simplicity rather than on completeness or extensibility [YBCD08]. Thus they are more focused on the simple aggregation of data from various Web sites than on multi-tenant aware pre-defined configurable applications as Cafe is.

Business Process as a Service Environments. The business process as a service market has been growing lately with offerings such as The Process Factory [Cor09a] Appian anywhere [App09] or Run My Process [Run09]. These tools allow end-users to define their business processes and host them on the platform supplied by the vendor. All providers also include a set of basic services that can be orchestrated with these platforms to create new applications. All applications offer a Web-based modeling tool in which the processes can be modeled and associated forms for user input can be designed and automatically deployed.

These tools differ from the Cafe approach, as they focus on the orchestration of existing services and all employ proprietary technology meaning that processes can only run on the respective platform and cannot be moved from one platform to another. Furthermore, once a process has been designed it cannot be annotated with variability points that can then be bound by other customers. Thus the process itself is the variability. Such business process as a service environments could be integrated into Cafe as a platform to deploy business processes on. However, as they employ proprietary technology, processes deployed on such a platform cannot be moved on another platform.

Furthermore the services in the platform cannot be employed outside of the platform in other applications.

2.2.4 IaaS

In this subsection different offerings providing computing resources in an infrastructure as a service delivery model are investigated and evaluated for their use in Cafe.

Amazon EC2. Amazon with its elastic compute cloud (EC2 [[Ama09](#)]) is the prominent example for an infrastructure as a service provider. EC2 allows users to create *Amazon Machine Images (AMIs)*, i.e. server images containing the necessary operating system and middleware to run a certain application. These server images can be stored on Amazon's simple storage service (S3 [[Ama](#)]). When a user requires the middleware and software installed in a certain image, the user can simply start one or more instances of that image on Amazon's infrastructure. Amazon then bills the user based on the hours of operation of each instance, as well as the network traffic transferred to and from the instance.

Amazon EC2 gives the user the flexibility to host any application and middleware on Amazon's servers needed to perform a certain task. The computing power can be dynamically adapted to the current needs by adding and removing instances. Again users are dependent on Amazon's EC2 platform as the machine images cannot be easily deployed somewhere else. However, first tool kits are emerging allowing the conversion of EC2 AMIs to VMWare [[VMw08](#)] images runnable on any platform supporting the VMWare workstation or player products. Conversions to the open-source XEN format are also available. Also, there are frameworks emerging such as OpenNebula [[SMLF09](#)] enabling the management of multiple cloud infrastructures in one framework as well as the deployment of virtual images in multiple clouds. However, there is currently no way to easily migrate Amazon EC2 AMIs to another platform. The applications and middleware already available on Amazon's EC2 or in a framework managed by OpenNebula can be used as components in Cafe applications. For example, a workflow engine on Amazon EC2 can be a component in a Cafe

application on which a workflow must be deployed. Since Amazon offers an API to dynamically provision AMIs, the Cafe system can integrate this EC2 API to provision the respective AMIs required to run an application.

GoGrid. GoGrid is similar to Amazon EC2 in offering machine images that users can start and stop on demand. However GoGrid only offers a limited set of predefined images and users cannot add new images. Again, as with EC2, GoGrid's server images cannot be transferred to another hosting infrastructure (e.g. EC2). Similar to EC2, hardware, middleware and software resources available on GoGrid and any other IaaS provider can be integrated as components in Cafe applications. However, as GoGrid, for example, has a different API than Amazon EC2, an abstraction above these APIs is needed in Cafe enabling the integration of resources from any of these providers in Cafe. Then a concrete provider can be selected during the provisioning based on cost or other functional or nonfunctional properties.

Hosting Providers. Web-Hosting providers offer infrastructure (namely Web servers) as a service on a more long term basis than the IaaS providers. Web hosting providers either offer *root servers* allowing customers to modify everything beginning at the operating system, or *managed (virtual) servers* containing a predefined operating system where customers can install any middleware and application they might need. Often Web hosting providers also provide already installed middleware such as an Apache Web server and MySQL databases that customers can use to build their Web applications. Most of the time these Web servers and databases are standard software tools so that users can move their Web applications from one Web hosting provider to another. Cafe applications can contain the resources offered by a hosting provider as a component. Since most hosting providers do not offer APIs with which resources can be provisioned on demand, external provisioning engines are needed that enable the provisioning of Cafe components on the resources provided by these providers. For example, a Web-component could be provisioned on a Web-space already acquired from a hosting provider, by copying the application to that Web-space via FTP.

2.2.5 Application Portals

One goal of Cafe as shown in Figure 1.1 is to enable customers to select an application from a portal that is then automatically provisioned. In the following some application portals are examined with regard to their applicability for Cafe.

MySMBStore. MySMBStore [Ete09] is a Web-based store for Web-based applications from different providers. MySMBStore targets small and medium sized businesses that require applications such as E-Mail, CRM tools or customer feedback applications. Customers can try and buy applications directly from MySMBStore's portal. The applications are hosted at the respective provider's site, as MySMBStore is only re-selling these applications. The advantage is a unified provider for several applications which results in a single point of contact for payments and customer support. MySMBStore differs from Cafe as it does not separate the provider and application vendor roles, as applications are still provided by the company that originally built them or by a specific provider and are not packaged to be run by other providers. Thus MySMBStore is an aggregation of the provider's portals and does not offer a specific packaging format and model to describe applications that can be deployed on any suitable provider. Additionally, MySMBStore does not allow sharing components of an application across different applications.

Multiple SaaS provider such as Salesforce [Sal09b] allow the subscription to their services or extensions of their services in a portal, however, these portals do not allow the subscription to other services other than those of the respective provider which is also the vendor.

2.2.6 Cloud Management Portals

Cloud management portals such as Elastra [Ela09] and Rightscale [Rig09] offer cloud management services on top of IaaS providers. Elastra provides special languages used to describe the infrastructure components of an application such as application servers, databases and their connections. However, when deploying applications based on these models concrete hosts must be

manually assigned to these modeled components, and application components not included in the list of pre-defined components must be manually installed on top of the provisioned middleware components. 3tera [3te09] offers a similar approach. In the 3tera Web portal users model applications based on pre-defined virtual images that are then deployed in a data center. As opposed to Cafe, all of these approaches focus on the management of resources and applications assuming that both are modeled and managed within the target environment of the respective provider. Applications modeled using the Cafe models cannot only be deployed at one provider but at any provider supporting the Cafe models. Cafe also offers a variability model that is more powerful than the model of simple independent configuration parameters offered in the above approaches. The Cafe variability model allows the modeling of variability across different components and with complex dependencies also for components of new applications and not only for a set of predefined components. Furthermore, none of the presented approaches allows the modeling of explicit multi-tenancy of an application.

2.2.7 Provisioning Engines

In the systems management domain *provisioning engines* [LGDK05, BTD⁺, CSL03, EMME⁺06] are used to automate systems management tasks relating to the deployment of new infrastructure resources such as servers, middleware and software. In the Cafe scenario such provisioning engines can be used to set up the necessary middleware as well as application resources to run a Cafe application for a new customer once he has subscribed to this application. Several commercial and open source provisioning products exist on the market today. These engines can be classified into three categories: image distribution engines, general purpose and special purpose provisioning engines.

Image Distribution Engines. Image distribution engines are provisioning engines that can automatically install machine images (such as OVF [DMT09] VMWare images [VMw08], XEN images [Wil07] or Amazon EC2 AMIs [Ama09]) on a set of servers. Most of the time these image distribution engines come with a user interface that allows dragging and dropping images from an image

catalog on one or more servers in the server pool. Advanced image distribution engines (such as VMWare ESX [Wal02]) allow the migration of images from one physical server to another during runtime and the dynamic starting and stopping of instances of images. Internally Amazon EC2 employs such an engine (the open source XEN hypervisor [But06]) to assign the EC2 AMIs to physical computing resources.

Another example of such an engine is the open source OpenQRM [Ope10b] project. The OpenQRM engine allows to define machine images it can then provision on servers in a server pool. Therefore, each server must be made available to the system. Provisioning can then be done with the help of a command line API or a Web-based portal.

Other examples such as the IBM Tivoli Systems Automation Manager (TSAM) [IBM10] can be used as advanced image distribution engines or make use of virtualization products such as VMWare or Xen. TSAM, for example, allows provisioning of services from a *service catalog* to one or more nodes in a server pool. In addition to standard images, complex topologies can be defined spanning over multiple servers and contain points-of-variability such as variable IP addresses that are filled during the provisioning.

Image distribution engines can be used in Cafe to provision basic components, such as application servers, database management systems or even entire application stacks including application components such as workflows or Web services. However, these engines are not enough to realize the full Cafe scenario as they lack the flexibility needed for Cafe. For example, a Cafe application might already be partially provisioned because another customer is already using it and parts of the application can be shared. Therefore, a Cafe application can typically not be represented as a single image, however, individual components of a Cafe application can be seen as configurable images that, together with components that must be deployed on them, form the application. Thus image distribution engines are tools realizing an important functionality required by Cafe. Since multiple different image distribution engines exist at different providers it is important that the Cafe architecture can deal with a multitude of these engines and the provisioning actions for an application are independent on the concrete engine in use. In Chapter 5 it is

shown how image distribution engines and other provisioning engines can be orchestrated to provision an application.

General Purpose Provisioning Engines. In contrast to image distribution engines, general purpose provisioning engines deal with the installation of pre-defined software packages on already provisioned and running servers. These packages can range from operating systems, over middleware to complete applications spanning multiple machines. In Cafe such engines can be used to provision new components on already existing components. For example, a general purpose provisioning engine at a provider could be used to install a BPEL engine on an already running application server because the application required by a customer, needs such a BPEL engine.

A general purpose provisioning engine is, for example, Sun Microsystems' N1 Service Provisioning System [Hum06, Sch08]. In this system, users can define software packages with associated provisioning and management plans. A user can, for example, specify an Apache Tomcat server package consisting of all files necessary to run an Apache Tomcat server. The user can then associate the package with additional provisioning plans describing the necessary actions the provisioning system has to execute to install the software. Also IBM's Tivoli Provisioning Manager, for example, contained in IBM TSAM is such a general purpose provisioning engine used to provision IBM applications such as IBM's WebSphere application server.

Special Purpose Provisioning Engines. Special purpose engines typically come with complex middleware products such as application servers or complex applications. These special purpose engines can then provision one single product often in a multitude of configurations. For example, IBM dynamic infrastructure (IBM DI) for SAP [IBM09] can provision a complete SAP environment. Other plug-ins for this special purpose engine can provision complex WebSphere application server topologies and the corresponding IBM WebSphere Process Server BPEL engine.

Other special purpose provisioning engines are shipped with complex products such as operating systems or middleware and offer *silent installs*. These provisioning engines cannot be used to provision arbitrary applications but can

be used to solve very specific, complex provisioning tasks.

2.2.8 Topology Modeling and Provisioning

In [KEK⁺09] the authors describe a tool for the virtual solution composition and deployment in infrastructure clouds. The work of this project [EMME⁺06, AEK⁺07, KEK⁺09] focuses on the modeling of infrastructures composed of infrastructure components available in one or several clouds at different providers. In [KEK⁺09] the composition of *virtual appliances* into a *virtual solution model* is described. A deployment architect then transforms the virtual solution model into a *virtual deployment model*. This virtual deployment model is specific for one or several clouds, as it contains concrete implementations for the virtual appliances. A planning component then generates deployment plans out of the virtual solution model to deploy the virtual appliances on the appropriate cloud infrastructure components. Some of the concepts presented in [KEK⁺09] can be leveraged by the Cafe application model presented in this thesis. This is namely the description of deployment dependencies between infrastructure components, the *topology* of the combination of components. The approach to model deployment dependencies presented in this thesis is similar to the approach presented in [EMME⁺06, AEK⁺07, KEK⁺09], as it allows to describe which properties of another component is necessary to configure another component using it. However, the approach presented here includes these dependencies in the overall variability model of the application. This means that constraints on deployment dependencies can be in turn dependent on functional variability of the application. Other differences of the application model presented here in comparison to that of [KEK⁺09] are that in the Cafe model components can be described on the infrastructure level as well as on the application level, whereas the concepts presented in [KEK⁺09] focus on the composition of virtual appliances and thus the IaaS and lower levels. Furthermore, the Cafe deployment model natively includes multi-tenancy patterns for individual components (cf. Section 3.3.6). The Cafe approach does not make any assumption on how infrastructure components are provided, as it allows to integrate arbitrary provisioning engines that are then called by the

provisioning environment (cf. Section 5.2).

Furthermore, the Cafe model natively includes a description of the multi-tenancy aspects of an application on all levels. I.e. it is possible for application components as well as infrastructure components to describe whether they can be shared or not.

In addition, in Cafe the selection of concrete resources is done automatically, whereas in [KEK⁺09] it is done by a solution architect. The approach is geared more towards very specific complex setups of infrastructure for complex systems where composition is done at virtual appliance level, whereas the Cafe approach is geared towards standardized components across different providers that are composed at application level.

The approach of [EMME⁺06] also employs a semantically rich metamodel which requires the complex modeling and configuration of individual infrastructure components before they can be deployed using the techniques presented in [EMME⁺06, AEK⁺07]. Thus modeling such components and applications is a very complex task. Cafe aims at providing an easy-to-understand general purpose metamodel that can be used to describe arbitrary cross-component application templates and can use the models of [EMME⁺06, AEK⁺07] as implementations of individual infrastructure components of a Cafe application.

Using workflows to perform provisioning is also investigated in [KB04]. In [KB04] the predecessor of WS-BPEL, BPEL4WS is used to execute various provisioning actions on a IBM Tivoli based infrastructure. In essence, BPEL is used as an orchestration language and the BPEL engine as an orchestration engine to execute services that are provided by a provisioning engine. The approach presented in [KB04] however, assumes that a provisioning workflow is explicitly defined to perform one specific task (i.e., to set up an application server). The approach presented in [EMME⁺06] as well as our approach allows to generate provisioning scripts [EMME⁺06] or provisioning workflows (Cafe) out of a model of the infrastructure. In [SH94, EMME⁺06] artificial intelligence (AI) planning techniques are used to derive the provisioning workflow and the operations to invoke.

2.2.9 Composite Services and Applications

In [BSD03, MSB04, BDS05] the Self-Serv environment is described allowing the automated provisioning and orchestration of composite services. The presented approach is based on P2P communication and is geared towards dynamic environments in which parts of the services to be orchestrated are on reliable infrastructure and parts are not. In [S⁺06] a framework for such composite services is introduced enabling users to define customer-specified personal compositions that can then be executed in a decentralized fashion by executing them on a tuple space. This approach differs from the Cafe approach in various aspects as it has a different focus than Cafe. It is geared towards dynamic environments with transient resources, whereas Cafe is geared towards more stable environments. Furthermore, [S⁺06, BSD03, MSB04, BDS05] assume that the services to be composed are already in place and running, whereas Cafe can provision composite applications from scratch. Multi-tenancy is not covered in [S⁺06, BSD03, MSB04, BDS05], as they assume that all services are shared and the composition is specialized for one particular user. From that point of view this approach is similar to that of a business process orchestrating a set of services into a higher-level service. In their adaptive service composition model described in [S⁺06], the composition is described as a *process schema* that can be extended, personalized and reused by end users. The process schema can thus be configured to a personalized composite service that can then be enacted by the corresponding framework.

The approach presented in [RLM⁺09] relies on a domain-specific language (the Vienna Composition Language VCL) to describe compositions. It is similar to the approach of Cafe, as it aims at providing these compositions as a service again. The approach is also similar to [S⁺06], as it is focused on composition of services which are automatically discovered in the environment given a set of QoS parameters. The composition is then transformed into a workflow. Variability is not explicitly modeled but compositions can be customized by changing the QoS and features of the composition. The Cafe approach is more general as it is application centered rather than process centered and therefore also allows to describe all other artifacts of the application (including GUIs,

infrastructure components, requirements on the provider infrastructure etc. and their variability).

Several research projects have focused on the QoS-aware composition of services [RLM⁺09, CDPEV05, KSBB08, JRGMO5]. However, they all focus on the automatic composition of already provisioned services [RS05]. The Cafe approach is different, as it focuses on the QoS-aware deployment of composite applications that can be composed out of components already deployed at a provider and components that need to be deployed.

In [LYJP09] the SIMPLE tool is introduced. This tool allows to create integration solutions derived from *solution templates* [FYL⁺08]. Integration solutions are derived from the templates by filling variability points. However, the approach is aimed solely at the generation of integration solutions and thus is limited in scope. For example, the variability mechanism does not allow complex dependencies and the building blocks for templates are collaborations (representing workflows) and connectors connecting the collaborations as well as screen views [FYL⁺08]. Thus SIMPLE contains customizable components but only for integration solutions and not for arbitrary applications. Furthermore, it does not allow to describe the required infrastructure to deploy the integration solution, which is an essential part of Cafe.

2.2.10 Cloud Interoperability

The Reservoir project [RBL⁺09] aims at providing a *federated cloud* environment where applications can be distributed across different providers. The Reservoir model distinguishes between *infrastructure providers* owning infrastructure resources and *service providers* providing services on top of these infrastructure resources. The Reservoir approach is similar to the Cafe approach, as it allows to define *Service applications* across multiple clouds. However, the Reservoir approach differs from the Cafe approach, as service providers are not separated from application providers, i.e. one service application is provided by exactly one service provider whereas in Cafe one application can be offered by multiple providers. Therefore, Reservoir does not contain a generic model to describe applications in a provider independent manner. In addition, Reser-

voir does not contain a generic variability model. It is assumed that service applications are customized by a specific tool before a new tenant for one of these service applications is provisioned. Thus the focus of Reservoir as shown by its architecture in [RBL⁺09] is more on the definition of a federated cloud environment than on the definition of provider-independent applications, their variability and provisioning, as it is the focus of Cafe. In future work the federated cloud of Reservoir could serve as an good execution environment for Cafe and could be plugged into the unification layer as another cloud.

2.2.11 Package Standardization Efforts

In this subsection related work from various areas describing the packaging of applications and related infrastructure is investigated.

Open Virtualization Format. The *Open Virtualization Format* (OVF) [DMT09] provides a common standardized format for virtual machines. Several key players in virtualization such as VMWare, XenSource, Symantec, Sun, Intel and IBM are involved in the standardization activities. OVF defines several items such as a package structure, virtual disk formats and a descriptor format (called the *OVF descriptor*) providing metadata about the package.

While OVF is a necessary step in the direction of standardization of virtual machine images it only fulfills a small part of the requirements of Cafe. OVF is geared toward a standardized format for images distributed by image distribution engines and therefore does not specify applications and their variability explicitly. Such an image could contain an application which is already installed on the necessary middleware and operating system making deploying an application on existing middleware impossible.

Service Component Architecture. The *Service Component Architecture* (SCA) is a set of specifications describing a package format for applications built following a service-oriented architecture. SCA allows specifying composite SOA applications developed in different programming languages and includes a basic mechanism for configurability of the components forming such an application. The SCA Assembly Model Specification [Ope07] defines how an

SCA composite application (called *composite*) is built from individual artifacts (called *components*). An SCA component represents a business function. A component provides *services*. These services can be used by other components to call this component. Components can call other components thus consuming their services via *references*. A component reference is connected to a component service via a *wire* which points to a service. Similar to components, composites can contain services and references. Services of components contained in the composite can be promoted as a service of the composite that can be called by requesters from outside the composite. Similar to component services, component references can be promoted to composite references. Composite references denote that the reference is served by a service outside the composite. A composite including all its components is described in an XML document as specified by the SCA assembly model specification. An SCA component is a configured instance of an SCA component implementation. This means that several components can use the same implementation but configure it differently. A component implementation is a concrete implementation of a business function. SCA provides a set of predefined implementation types referring to specific implementation technologies such as Java, BPEL, C++, Spring or Java EE EJB technology. The SCA assembly model is recursive by allowing composites to be implementations of components in other composites. Implementations (including composites) can have properties. Using properties, an implementation can be configured externally by specifying property values in the component that uses an implementation. In particular, different components can use the same implementation but configure it differently by specifying different property values. SCA specifies how properties are specified and accessed, for example, in Java and BPEL code. The SCA assembly model specification also describes how SCA artifacts and other artifacts (such as code files) are packaged and deployed. The central unit of deployment in SCA is a contribution. A contribution is a package containing implementations, interfaces and other artifacts necessary to run components. The SCA specification specifies an interoperable package format for contributions. The package format proposed by the SCA assembly model specification is a Zip file, however, other packaging formats are explicitly allowed. The `sca-contribution.xml`

file lists the runnable SCA composites within the contribution, i.e. those composites that can be installed in the SCA domain the contribution is installed into. An SCA domain represents a complete runtime configuration, potentially distributed over a series of runtime nodes. Additionally, an SCA domain defines the boundary of visibility for all SCA mechanisms [Ope07]. In particular, this means that artifacts (such as services) from one contribution are visible to artifacts (such as a reference) from another contribution if both contributions reside in the same SCA domain and are made explicitly visible through imports and exports. In [MLP08] and [Arn09] the applicability of SCA to describe customizable SaaS applications has been investigated and extensions to SCA to extend it with the description of multi-tenancy of individual components and an extended variability model have been proposed. However, even this extended SCA does not allow describing the requirements on the underlying infrastructure and only allows modeling applications that can later be run on a SCA infrastructure and is thus not suitable for Cafe.

Enterprise Archives. The *Enterprise Archive* (EAR) [Sun08] file format is a packaging format for Java EE applications. An ear file is a Zip file containing several artifacts, also called *modules* of the application. Different types of modules exist, such as Web modules, .jar files, EJB modules and resource adapter modules. An enterprise archive also contains a metadata directory with one or several deployment descriptors. These deployment descriptors contain metadata such as transaction settings, naming, configuration etc. of the application. Enterprise archives are deployed in Java EE compliant application servers. These application servers then install the contained modules in the appropriate containers.

Given the requirements for Cafe, .ear files lack several requested features: First of all, they are very specific to Java EE environments. There is no possibility to include other packages (for example .net assemblies) as modules in the enterprise archive. Furthermore, there is no variability mechanism to describe variability in the different modules of the application. The .ear file format also does not support the inclusion of other layers than the application layer. There is limited support to describe requirements and configuration for

the underlying middleware (i.e. the application server) via the deployment descriptors. A deployment descriptor, for example, allows to define security roles for all components in the application. However, there is no possibility to specify which concrete middleware components are needed to run the application and there is no way to describe required nonfunctional properties such as QoS parameters for them. Additionally, there exists no mechanism to describe whether certain modules of the application are multi-tenant aware or not.

In summary .ear files can be used as a packaging standard for components in Cafe developed for a Java EE environment. Since the focus of the .ear package is a different one than the one of the Cafe packages, the two can be used in conjunction.

Application Packaging Standard. The *Application Packaging Standard (APS)* [SWS07] is a standard package format for Web applications proposed by SWSOft. APS defines a zip-based package for Web applications and a descriptor file format containing relevant metadata for the application. The descriptor includes general information about the packaged application such as an icon, screen shots a name and a description as well as information about the application vendor. Furthermore, information about available languages, entry points into the application (in form of relative URLs) and URL mappings are described in the descriptor.

The descriptor also contains links to configuration scripts that are executed upon installation or update of the application. Also, application settings and their data types can be specified in the package descriptor. Corresponding tooling can then display controls for these settings allowing users to fill them with specific values. The package descriptor also contains requirements on the infrastructure posed by the application. These requirements can be particular runtime environments (such as PHP or .NET) as well as requirements on databases (such as on a special version of MySQL).

The application packaging standard is geared towards Web applications and in particular PHP, ASPNET, CGI and similar types of applications. A basic point-of-variability mechanism is contained in the package descriptor in the form of

the application settings but no complex dependencies between the points-of-variability can be expressed. However, applications cannot be decomposed into components and requirements can only be stated for the entire application and not for individual components. Especially there is no way to specify individual components as being shared among different tenants, as the concept of multi-tenancy is not contained in the APS standard. Furthermore, there is no concept of individual components or services of an application which is needed in Cafe. The APS website [SWS07] offers a portal from which several applications described in APS can be downloaded.

Evaluation of Packaging Standards. Regarding the separation of application vendors and application providers, several standards for the packaging of (composite) applications have been examined. Table 2.1 shows the different packaging standards and how they support various properties required for Cafe. The first property are the supported layers in the application stack. To be suitable, all layers from infrastructure to application components should be supported which is not the case with any of the formats. An orthogonal variability model is not supported by any of the formats, some of them contain an internal variability model which can be referenced by an external orthogonal variability model. Requirements on the infrastructure can be included in APS and OVF but not explicitly in ear files and SCA composites. Multi-Tenancy patterns are not supported by any model.

	OVF	SCA	APS	EAR
supported layers	virtual images	composite apps	Web applications	Java EE application components
variability points	yes (specific)	via properties	yes (specific)	not explicit
infrastructure requirements	yes	not explicit	yes	not explicit
multi-tenancy patterns	no	no	no	no

Table 2.1: State of the art evaluation of packaging formats

As a conclusion, none of the investigated formats is directly suitable to be

used to target all research challenges outlined in Section 1.2. The main reason for this is that these formats are targeted at a very specific set of applications or a specific layer in the application stack, such as virtual machines. Thus a more general approach to defining components, their relations and the cross-component variability is needed.

2.3 Summary and Conclusion

In this chapter background, related work and existing systems related to the research issues of Cafe were investigated. Several technologies and architectural paradigms exist that can be taken into account when building, customizing and provisioning Cafe applications. For example, existing packaging and architecture modeling techniques can be partially used when modeling and packaging Cafe applications, however none of them is general enough to tackle arbitrary applications. Thus, these techniques and models influence the Cafe application metamodel in Section 3.3. In software product line engineering, variability needs to be defined, thus variability models such as OVM from software product line engineering influence the variability metamodel in Section 3.4. Several provisioning engines and environments are in use today to provision different components and infrastructure. How these provisioning engines are integrated into Cafe is described in Chapter 5.

DEVELOPMENT OF CAFE APPLICATION TEMPLATES

Similar to traditional applications sold to customers and then installed on their premises, applications in Cafe (called *Cafe applications* from now on) are developed using a development process including requirements engineering, specification, design, implementation, test and deployment. However, given the nature of Cafe applications, the development process varies from that of traditional on-premise applications. Also the artifacts produced during the development process vary. For example, a Cafe application must be packaged so that it can be offered by any provider. This chapter describes the development process and artifacts necessary to develop a Cafe application.

The first section of this chapter describes the roles involved in the development of Cafe applications. The roles are compared to the roles in traditional on-premise application development as well as the development of software product lines. Section 3.2 introduces the overall Cafe development process and the artifacts produced during individual phases of this development process.

Section 3.3 defines the formal metamodel for the *Cafe application model*.

The application model is used to define *Cafe application templates*. Cafe application templates are used to model the components of an application and the requirements they pose on the provider infrastructure. A serialization of the application model is given in form of *application descriptors*. The definition of variability is an important part of the development process for Cafe application templates. Section 3.4 contains a formal definition of the Cafe variability metamodel as well as a serialization format for the variability metamodel, the *variability descriptors*.

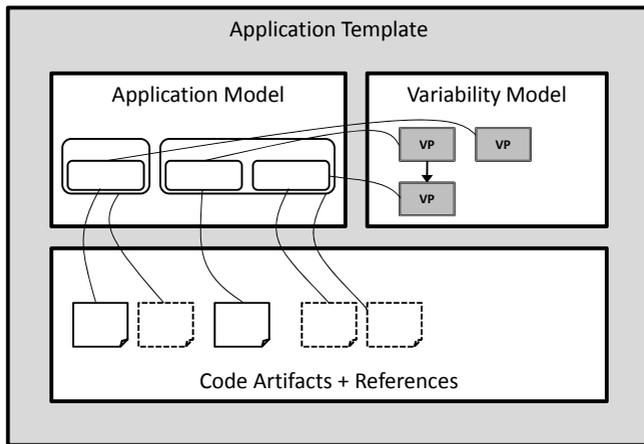


Figure 3.1: Ingredients of an application template

Figure 3.1 shows the ingredients of an application template. Despite the application model and the variability model in form of their respective descriptor files the template contains the code for some of the components in the application. For those components for which the code is not included the template contains references whose realizations must be supplied by the provider.

Having modeled and developed the components of an application as well as their variability and implementations, the application is ready to be packaged. The package format for Cafe applications, the *Cafe application archive* (.car) format, is introduced and defined in Section 3.6.

In Section 3.7 the individual phases of the development process for the development of Cafe application templates are described in detail. It is shown how the previously introduced artifacts are created and refined in the individual phases of the process.

3.1 Roles in Cafe

In this section the roles involved in Cafe are introduced. This is done by describing the roles in traditional on-premise scenarios and then expand them and add additional roles as required for Cafe. The roles involved in Cafe vary from those involved in traditional on-premise software scenarios. In a traditional on-premise application two main roles are involved: the first is the *application vendor*, the company selling the on-premise software. The second role is that of the *customer* running and using the software. In case an application requires customization on the customer's side, often another role, the role of the *customizer* is introduced. This is the case, for example, for complex ERP applications (such as SAP systems) that require a considerable amount of customization and adaptation [SH00]. The role of the customizer is often filled by consulting companies (that may be divisions of the application vendor). For such applications two main phases in the development process exist, the development of the basic application and the customization phase.

In software product line engineering also two main phases exist during the development of an application for a customer. The *platform engineering phase* where the application platform including the variability is developed and the *application engineering phase* where the variability is bound. The variability of the application platform is normally bound before a new member of the product line (an application) is delivered to the end user. As shown in [Bos00a, Bos00b, BMK⁺01, Bos01, PBvdL05] different organizational structures exist separating platform and application engineering inside an organization. Thus two separate roles are defined, one responsible for the platform engineering and one responsible for the application engineering. In some organizational models these roles can be filled by the same part of the organization, however, in all models both roles are filled by the same organization, namely the application

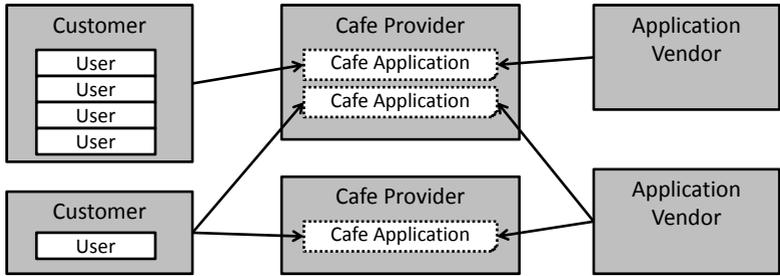


Figure 3.2: Roles in Cafe

vendor.

In outsourcing scenarios often another role is introduced: The role of the *application service provider* [Tao01], that hosts and maintains the application for a customer. This provider role is also present in SaaS application scenarios where the role is called *SaaS provider* [GM02, CC06].

3.1.1 Roles in Cafe

Given the brief analysis of different roles in different scenarios in different software development and delivery models, the main roles important for Cafe are shown in Figure 3.2 and described below. These distinct roles can be played by different organizational entities or one organization or person can have several roles at once.

Application Vendor. The *application vendor* in Cafe is similar to the application vendor described previously in the on-premise scenario. The Cafe application vendor sells software that can be offered at any Cafe provider. Customers can then subscribe to and unsubscribe from the application. Thus, as shown in Figure 3.2, no direct relationship between customers and application vendors exist.

Cafe Provider. A *Cafe provider*, or provider in short, is a company offering Cafe applications. The applications hosted by a Cafe provider can be developed in-house or licensed from an application vendor. The Cafe provider thus offers

a set of Cafe application templates. Customers can then select one or more templates to which they want to subscribe.

Customer. A *customer* of a Cafe application is a person or an organizational entity such as a company subscribing to a Cafe application. Once a customer has subscribed to a Cafe application, the customer can customize the application. Customization is optional, since there might be applications not requiring customization by a customer. The customer, albeit not being involved in application hosting and the development of the basic application template, thus plays an important role in the development of a Cafe application (cf. Figure 3.3). The term *customizer* is used in this thesis to denote a particular person at a customer performing the act of customization of an application template.

User. A *user* of a Cafe application is a person or program using the application. A single customer typically has one or more users. For example, a company subscribing to a Cafe application can have multiple employees who use the system. A user can also be an application, i.e. a different application using the Cafe application, for example, via an API.

3.2 High-Level Cafe Development Process and Terminology

The Cafe application development process contains several phases distinguishing it from development processes for traditional on-premise applications. In this section, the Cafe application development process is described in detail. The focus of this thesis is on the actual development (i.e. architecture and implementation as well as deployment) of such applications. Other phases of a traditional development process such as requirements engineering, specification, testing or documentation are thus only briefly touched in the textual description where needed. Figure 3.3 shows the high-level Cafe application development process, the roles involved and artifacts produced during the individual phases. These are explained below.

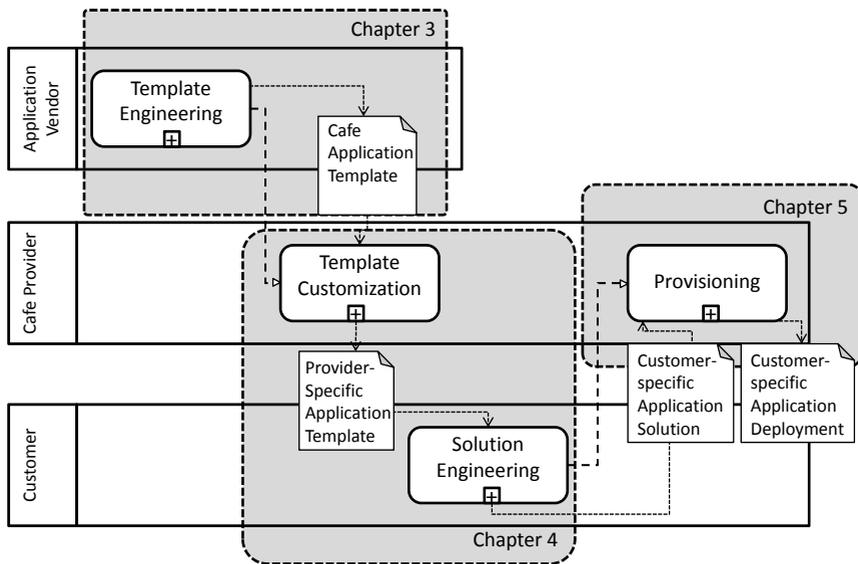


Figure 3.3: High-level application development process

3.2.1 Template engineering phase

During the template engineering phase the application template is developed. This phase is under the responsibility of the application vendor. The artifact produced during this phase is called the *application template*. The application template contains the necessary artifacts to provision and run the application as well as a definition of the variability which is not yet bound. An application template contains several artifacts: First of all, it contains the code necessary to run the application as well as references to components that must be supplied by a provider. It also contains an *application descriptor* that is a serialization of the *application model* describing all the components and their dependencies of the application template. The application template also contains a *variability descriptor* which is a serialization of the *variability model* of the application template. The variability model contains the variability of the template, i.e., where and how it can be customized. The template engineering phase is

described in detail in Section 3.7.

3.2.2 Template customization phase

In this phase the template is refined by a provider to adapt it to the particular environment of a provider. The output of the template customization phase is a partially bound variability model of the template. All variability binding already performed in the template customization phase must not and cannot be performed by the customer during the next phase, the solution engineering phase. Providers can thus limit the choices of their customers by already binding some variability points of a template. Template customization is described in more detail in Section 4.4.

3.2.3 Solution engineering phase

In the solution engineering phase, the variability of an application template is bound by a customer to adapt the Cafe application according to that customer's requirements. The output of this phase is a *customer-specific application solution*. A solution is a template in which all variability that can be bound during the solution engineering phase, is bound. Thus the solution contains a *customization* of the variability model of the corresponding template. The customization of the variability model contains the values for the variability as it has been bound by the customer. A solution can still contain variability that must be bound during the provisioning of the application, i.e. not all variability points must be bound prior to provisioning. The solution engineering phase is described in detail in Section 4.5

3.2.4 Provisioning phase

In the provisioning phase performed by the provider, a solution is *provisioned* on the infrastructure of the provider. Provisioning includes the binding of variability (such as concrete endpoints) that can only be bound at provisioning time and runtime as well as the physical deployment of components. The output of this phase is an *application deployment*, i.e. a provisioned application

solution ready to be used by the requesting customer. The provisioning phase is described in detail in Chapter 5.

3.3 Cafe Application Metamodel

As motivated in the analysis of related work, existing metamodels to model and package composite applications, such as SCA, APS or EAR are not expressive enough to define arbitrary Cafe applications or have a different focus than required for Cafe. This section therefore contains a definition of two metamodels to describe application templates and solutions. The first metamodel is the Cafe application metamodel to model the artifacts required for a Cafe application. The second one is the Cafe variability metamodel to model the variability in a Cafe application. The Cafe application metamodel does focus on the description of the components in a Cafe application. It is primarily geared towards providing a metamodel suitable to model the different deployment artifacts as well as the required infrastructure necessary to run a Cafe application. The application model can then be combined with the Cafe variability metamodel to describe application templates including the variability that must be bound. The two main goals of these two metamodels is to be expressive enough that given the code included in the application model and the description of the variability and the required environment, corresponding tools such as customization workflows and provisioning workflows can be derived from the variability and infrastructure requirements, respectively, and can automatically set-up the application and prompt users for necessary decisions. The second main goal for these models is to be general enough to model arbitrary application templates where components are developed in any programming language which distinguishes the metamodels from specific metamodels for special types of applications and variability as shown in Section 2.2.11 and Section 2.1.4.

3.3.1 Short Overview

Figure 3.4 gives an overview of the most important entities and relations of the Cafe application metamodel. A formal definition of all artifacts of the Cafe application metamodel is given below. Here, only a short textual overview of the most important entities is given: An *application model* consists of a set of *components*. An application model itself is a special component and can be recursively used as a component in other application models. Components have an associated *component type*. Components can have a *deployed on* relationship to another component, indicating that the component must be deployed on the other component. Components are implemented by an *implementation*. An implementation is of a specific implementation type. An implementation can be supplied with the component, then a set of *files* is defined realizing the implementation. Files, realizing an implementation consist of a set of *building blocks*. The implementation of a component can also be of the implementation type *provider supplied* indicating that the application model developer relies on the *provider* to supply such a component.

The first part of the metamodel to be described is the Cafe application metamodel below. The variability metamodel is described in Section 3.4. The names of the attributes as well as the descriptions in parenthesis of the associations in the UML diagrams correspond to the functions in the mathematical metamodel below.

3.3.2 Example Application Model

To illustrate the purpose of the Cafe application metamodel an example of an application model which is an instance of the Cafe application metamodel is given below and shown in Figure 3.5. The example application model consists of a *GUI* component, that must be deployed on an *Application Server* component which must be of component type *JEE Application Server*. The implementation of the *Application Server* component must be supplied by the provider, while the implementation of type *Java Web Application* is supplied for the *GUI* component in form of a *.war* file. The application model also consists of a *Workflow* component. The *Workflow* component has to be deployed on

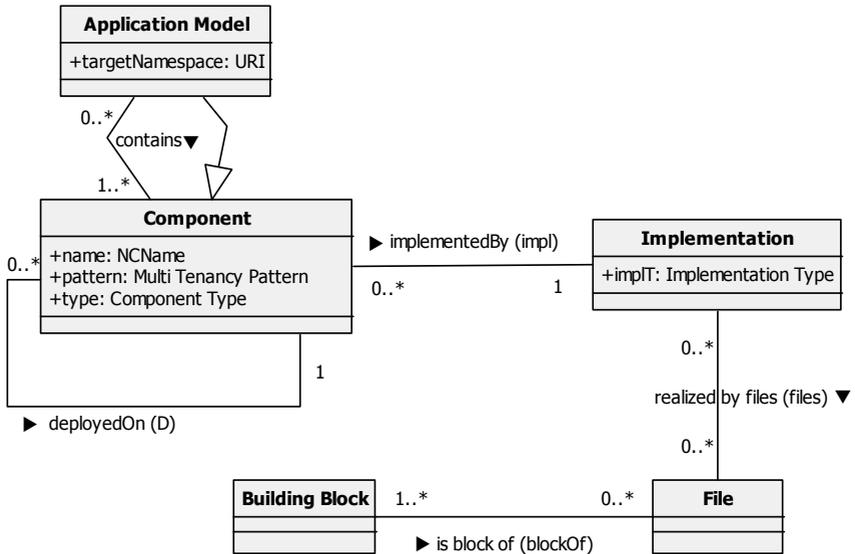


Figure 3.4: Cafe application metamodel

a *BPEL Engine* component of type *Apache ODE* that in turn must be deployed on a *Servlet Container* component of type *Apache Tomcat*. Both, the *Servlet Container* component and the *BPEL Engine* component must be supplied by the provider. The implementation of the *Workflow Component* is supplied with the application in the form of an implementation of type *BPEL*. The *workflow* component orchestrates two services represented by the *SMS* component and the *CRM* component. The *SMS* component must be of type *SMS Sender* and must be supplied by the provider.

Components can also be implemented by a reference to a component provided by a third-party. The implementation type of such components is *external*. These components are called *external* components from now on and are referenced via an EPR. The difference between provider supplied and external components is that external components are not under the control of the provider, they must exist to be used in the application without any intervention by a provider. The *CRM* component is a component supplied by an external

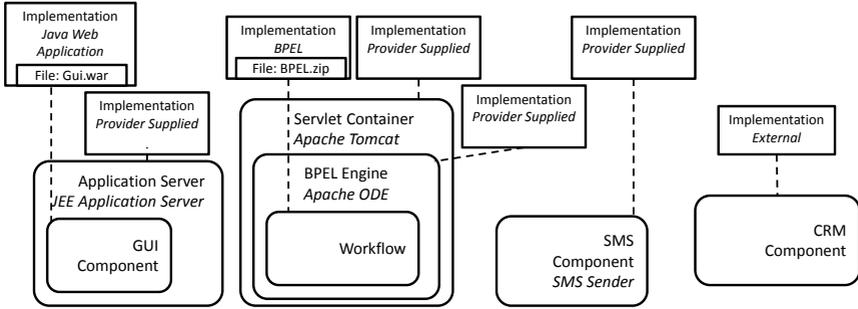


Figure 3.5: Example application

third party and is thus of implementation type *external*.

3.3.3 Formal Definition

Having introduced the Cafe application metamodel briefly, a formal definition of the artifacts of the Cafe application metamodel are now given. In the following formalizations π_i is used as a projection on the i -th element of a tuple and $\rho(A)$ denotes the powerset of the set A . The following definition briefly introduces the elements of the application metamodel which are then described in detail in the rest of this section.

Definition 1 (Application Model). *The set of application models $AM = \{\dots, a_i, \dots\}$ defines the set of all application models. An application model $a \in AM$ is a tuple*

$$a = (C, I, F, B, D, type, impl, implT, files, blockOf, p)$$

where

- C is a set of components,
- I is a set of implementations,
- F is a set of files,
- B is a set of building blocks,
- D is a set of deployment relations,

- *t* type is the (component, component type) map,
- *impl* is the (component, implementation) map,
- *implT* is the (implementation, implementation type) map,
- *files* is the (implementation, file) map,
- *blockOf* is the (building block, file) map,
- *p* is the (component, multi-tenancy pattern) map

3.3.4 Components

Composite Cafe applications are assembled out of *components*. Thus, a component is the most important entity in the Cafe application metamodel.

Components can be anything from application-related components such as Web services, BPEL Workflows, Web applications to infrastructure related components such as Web servers, application servers, databases or workflow engines. The description of infrastructure components is the main difference to component models such as SCA, APS and UML component diagrams, as well as models for reusable assets such as RAS, where components are always only parts of the application and the infrastructure cannot be described. Thus in the Cafe metamodel components are units of deployment instead of units of functionality which may be overlapping in some cases but not in all.

Definition 2 (Components). *The finite set $C = \{\dots, c_i, \dots\}$, defines the non-empty set of all components in an application model $a \in AM$.*

A component is of a *component type*. For example, a component realizing a JEE application server is of type *JEE Application Server*, whereas another component is of type *Database Management System* or more specific *IBM DB2*. Component types are important, as they enable to specify components that must be supplied by a provider. For example, an application vendor can specify that an application must be deployed on an application server component of type *JEE Application Server*. The provider must then supply a component of the same type *JEE Application Server* on which the application is then provisioned. Component types abstract from concrete component implementations and thus allow to exchange components of the same component type. This is especially

important if a provider offers multiple components of the same type, e.g. an application server hosted in the data center of the provider and one outsourced to a third party.

To define the customization options of a certain component a component type optionally defines an *abstract variability model* (cf. Section 3.4.4).

Components defined as sub-components inside an application model that cannot be used outside this application model do not have to have an explicit component type. These components are called *internal components* from now on. These components can have an empty component type assigned. No explicit component type is necessary, as these components cannot be exchanged for different components of the same type. They can also be used in other application models, however, they are only usable as part of the whole application model and not separately as individual components.

Definition 3 (Component Types). *The set $CT = \{\dots, t_i, \dots\}$ defines the set of all possible component types available. Component types are not defined within an application model but are explicitly defined outside application models enabling the reuse across different application models.*

The function $type : C \rightarrow CT \cup \{\perp\}$ assigns a component type to a component. If \perp is assigned to a component, this component is marked as internal component.

Figure 3.6 shows the graphical representation of components as it will be used throughout the rest of the thesis. The name of the component is written in normal font, whereas its component type is written in italics.

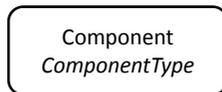


Figure 3.6: Component with type

3.3.5 Component Implementations

A component is implemented by an *implementation*. An implementation is of an *implementation type*. An implementation can be realized by one or more

concrete files using a concrete programming language. E.g. a UI component is implemented using HTML and PHP files, whereas a component representing a business process is implemented using BPEL, WSDL and deployment descriptor files. An application server component can, for example, be implemented by an OVF package file. The implementation type defines which type of implementation is used. For example, for a business process implemented using BPEL and WSDL files, as well as a deployment descriptor file, the implementation type is *BPEL*. For an application server component contained in an OVF package, the implementation type is *OVF image*. The implementation type is important as components define which components of which implementation types can be deployed on them (see Definition 16).

All known programming languages are suitable to realize a component implementation. In addition, a component can be of the implementation type *provider supplied*, i.e. the provider must somehow have an implementation for the component, as the implementation is not supplied with the application template. For example, one would not want to include the code for an application server component within a CRM application. Since an implementation type of *provider supplied* indicates that the provider must supply a component of that type, provider supplied components in an application model pose an implicit requirement on the provider infrastructure, namely that a component of the specified type is available. Providers can advertise the available component types to enable application vendors to design their application models in a compatible way so that only component types are used that are available at a provider.

Once an ecosystem of component types has been established, *profiles* could be agreed upon, grouping a set of component types required at a provider to support the profile. For example, providers supporting the JEE profile would be required to host a JEE compliant application server, a database and message-oriented middleware. Whereas providers supporting a LAMP (Linux, Apache, MySQL, PHP) profile [LW03], would be required to provide components of type *Apache Web Server* deployed on a *Linux* component with PHP support and components of type *MySQL DBMS*. A lot of the major Web hosting providers today implicitly fulfill this LAMP profile. Application vendors can then develop

applications requiring one or more of these standardized profiles to run.

The Cafe application metamodel allows the recursive aggregation of application models, thus a provider supplied component may also be implemented by another application model.

Definition 4 (Implementation). *The set $I = \{\dots, i_j, \dots\}$ defines the set of all implementations of components in an application model $a \in AM$.*

The function $impl : C \rightarrow I$ assigns an implementation to a component.

Definition 5 (Implementation Types). *The set*

$$IT = \{\dots, it_i, \dots\} \cup \{\text{providerSupplied}, \text{external}\}$$

is the set of all implementation types of components in an application.

The function $implT : I \rightarrow IT$ assigns an implementation type to an implementation.

The left part of Figure 3.7 shows a component and its associated implementation including the implementation type. In the rest of this thesis, the implementation is only shown in figures if it is required, otherwise an implementation is implicitly assumed. In the middle and right parts of Figure 3.7 shortcuts for components with an implementation type of provider supplied and external are shown. The implementation type is placed in italics below the component type.

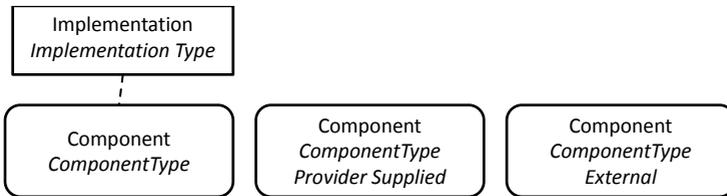


Figure 3.7: Component with associated implementation

Definition 6 (Files making up an implementation). *The set $F = \{\dots, f_i, \dots\}$ is the set of files realizing the implementations of all components of an application model $a \in AM$. The function $files : I \rightarrow \rho(F)$ assigns a (possibly empty)*

set of files to an implementation. In case an application is provider supplied or external, the set of implementation files is empty, i.e., $files(i) = \emptyset$ in case $implT(i) \in \{\text{providerSupplied}, \text{external}\}$. One file may be required for implementations of several different components.

A file consists of a set of *building blocks*. A building block is one part of a file that can be uniquely referenced. In an XML file, for example, a building block is an XML node [BPSM⁺00] which means it can be an XML element, an XML attribute or character data. A building block can be a complex XML element with several sub-elements and attributes in case of an XML file. In a properties file containing key-value pairs, a building block is such a key-value pair.

Definition 7 (Building blocks of a file). *The set $B = \{\dots, b_i, \dots\}$ is the set of building blocks that make up the files in the implementation of the components of an application.*

The function $blockOf : B \rightarrow F$ assigns building block to a file which means that the building block is contained in the file.

Definition 8 (Building blocks of a component). *Given a component c , the set*

$$B(c) = \{b \in B \mid blockOf(b) \in files(impl(c))\}$$

contains all building blocks in all files implementing the component.

Figure 3.8 shows an exemplary BPEL implementation of a component. The implementation of type *BPEL* contains three files, a WSDL file, a BPEL file and a deployment descriptor. These three files consist of a set of building blocks which are XML nodes (attributes, elements and text nodes) in this case as all three files are XML files.

Constraints for Components and Implementations.. The following constraints apply to components and their implementations:

Constraint 1 (Required Implementations). *A component must have an implementation, i.e.*

$$\forall c \in C : \exists i \in I \wedge impl(c) = i$$

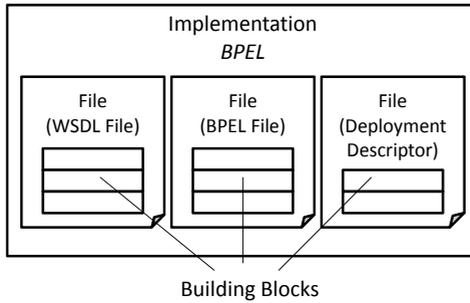


Figure 3.8: Relations between implementation, files and building blocks

Constraint 2 (Empty Implementations). *An implementation can only be empty if it is of type provider supplied or if it is of type external, otherwise it must contain at least one file.*

$$\begin{aligned}
 \forall i \in I: & \text{files}(i) \neq \emptyset \\
 & \vee \text{implT}(i) = \text{external} \\
 & \vee \text{implT}(i) = \text{providerSupplied}
 \end{aligned}$$

3.3.6 Multi-Tenancy Patterns

One important property of SaaS applications is their support for multi-tenancy. As discussed in the related work in Section 2.2.1, applications are either fully multi-tenant aware (such as SaaS applications) or none at all (for example applications in the ASP model). In the Cafe application metamodel vendors can specify at a component level if a component is shared between different customers of an application or not. This is necessary, as Cafe applications can be composed out of already existing components that either support multi-tenancy or not. In [MUTL09, MLU10] we give an analysis of several different services that exist and can be used to compose new applications and whether or not they support multi-tenancy. Sometimes it is also very difficult to implement a component so that it supports multi-tenancy. This is particularly difficult for middleware components. Currently, for example, no BPEL engine of a major

vendor supports multi-tenancy out of the box. Therefore, it is impossible to deploy BPEL process models in a single-instance multi-tenant aware manner. In [ALMS09] we investigate how current BPEL engines must be modified to support multi-tenancy. However, this requires a fundamental change of the architecture of such an engine. To ensure applicability even in these cases, Cafe must support multi-tenant aware components as well as components for which a separate instance must be deployed for each customer. To indicate whether a component can be shared between several customers or not, a multi-tenancy pattern is assigned to every component. In the following three basic patterns are described for components relating to whether they support the single instance multi-tenancy model or whether they require multiple instances to run multiple tenants. The patterns are described in a pattern form similar to the pattern forms in [GHJV95, Fow02] and especially [HW03]. A detailed analysis of multi-tenancy patterns for services in a service-oriented architecture can be found in [MUTL09]. In this thesis, not only services and their combinations are examined as in [MUTL09], but arbitrary components which can also be middleware or hardware components instead of services, following the horizontal and vertical composition of multi-tenancy patterns introduced in [MLU10].

Each pattern below is described with the following properties: (i) a title, (ii) an icon, (iii) a context describing the problem that can be solved using the advice described in the pattern, (iv) a short question summarizing the problem the pattern solves, (v) a description of the forces making the problem a problem, (vi) the result when the solution is applied, as well as (vii) relations to other patterns, (viii) examples on how to implement the pattern and examples for components using this pattern.

Single Instance Component.



Figure 3.9: Single instance component icon

Context: A component must behave the same for all customers using it. If the component must be updated, it should only be updated once for all customers.

How can I implement a component so that it can be reused by several customers and has the same behavior for all customers?

Forces: A component should have the same behavior for all customers. It is not necessary that the component is aware which customer currently uses it. In particular, customers do not need to be isolated, since the component does not have any customer-specific data or behavior. The component should only be deployed once for all customers and updates to the component should automatically be reflected for all customers using it.

Solution: Use a *single instance component* (cf. Figure 3.9) that is deployed once for all customers. The component can then be used by all customers and can be updated once for all customers.

Result: A component in the single instance component pattern is deployed once for all customers. After deployment of the single instance, the component is available for all customers. Since the component cannot distinguish between customers, isolation of customers is not guaranteed. Modification of the components for all customers is easy by using this pattern as the component can be updated once for all customers. In case the interface of such a component is changed all customers must be notified and adapted, otherwise their applications may break. Components following the single instance patterns can be developed just like any traditional component as no special care regarding multi-tenancy is required. Regarding scalability issues, the single instance component can be deployed multiple times and customer load can be balanced over these instances by a load-balancer.

Next: The single instance component does not distinguish between customers. In case customer-specific behavior or isolation is required while still only one instance is deployed the single configurable component pattern should be used. The multiple instances component pattern (cf. section 3.3.6) is another alternative. In this pattern customer-specific behavior is realized by deploying a

different component for each customer that can be customized to the respective customer's needs.

Examples: An example for a single instance component is a Web service offering customer-independent data such as stock quotes or the current weather. An example for such a service is the GlobalWeather service from WebserviceX.NET [Gen09]. A BPEL engine can also be deployed in this pattern, to be used by different customers deploying their processes on it. However, the engine would not be able to distinguish between the different customers and would not be able to isolate them.

Single Configurable Instance Component.

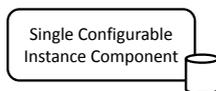


Figure 3.10: Single configurable instance component icon

Context: A component should behave differently for different customers. Customers must be isolated. It should be easy to add new customers to the component. Changing the non-customer-specific parts should be done at once for all customers similar to modifying a component following the single instance pattern (cf. Section 3.3.6). The load of different customers should be balanced and it should not be necessary to deploy the complete application stack for each customer.

How can a component which is shared among customers have customer-specific behavior?

Forces: A component should be tenant-aware, i.e. it must be aware which customer invoked it. Additionally, customers must be isolated despite being served by the same instance of the component. The component can show customer-specific behavior. Adding new customers should not require a redeployment of the component.

Solution:

Use a single configurable instance component that is only deployed once but can be configured differently for each customer. In [CC06, GSH⁺07, SZG⁺08] *tenant configuration data* is used to configure a SaaS application on a per-tenant basis.

Result: A *single configurable instance* (cf. Figure 3.10) component uses tenant configuration data to provide customer-specific behavior. A single configurable instance component must be invoked under a *tenant context*. The tenant context must contain the necessary data so that the component is aware on behalf of which customer it was invoked. Depending on this knowledge the component then must retrieve the correct configuration data for the customer. Different patterns exist how configuration data can be stored. It can be stored local to the component or in a central configuration database for the entire application [CC06, GSH⁺07].

Regarding the isolation of customers, a single configurable component must ensure this, e.g. with data-isolation techniques in the database layer. Updating of the non customer-specific part of the component can be done once for all customers, because all customers use the same instance. Updating the parts having customer-specific configuration is harder as it requires re-definition and redeployment of the configuration data of all customers, especially when new configuration options are introduced that must be bound [MMLP09].

Since one requirement for the single configurable instance components is the isolation of customers, this multi-tenant awareness must be an integral part of the design, architecture and implementation of the component. Building such multi-tenant aware components is more complex than building non tenant-aware components. In [CC06, AGJ⁺08, JA07] the development of multi-tenant aware database schemas is described. In [MLP08] we describe how SCA properties can be leveraged to build multi-tenant aware services. We describe multi-tenancy in service-oriented architectures and their implications on services in detail in [MUTL09]. In [GSH⁺07] and [CZZ⁺09] frameworks facilitating the development of multi-tenant aware applications are described. However, these frameworks focus on the multi-tenancy enablement of entire applications. While the Cafe approach can make use of such frameworks for

individual components, not all components of a Cafe application model must be multi-tenant aware.

Regarding scalability the single configurable instance component has the same advantage as the single instance component since the load of multiple customers can be balanced over one instance. In case additional instances of the component are required to cope with increasing demand, configuration data must be replicated to these instances.

Next: In case tenant isolation and tenant-specific behavior are not required, the single instance component pattern can be used. Components using this pattern require less effort during development as multi-tenancy must not be obeyed. Furthermore, they can be more easily replicated when multiple instances are required due to high load, whereas the single configurable instance pattern requires the overhead of accessing and replicating configuration data. SaaS providers such as Salesforce thus sometimes use a combination of the single configurable instance and multiple instances patterns (see Section 3.3.6). Salesforce, for example, has multiple instances of their application (deployed in multiple locations) where each instance serves a multitude of tenants [Sal09a]. However, each tenant is permanently assigned to one particular instance of the application. This reduces the replication of the complex configuration data to multiple instances and reduces the number of tenants per instance.

Examples: A real life example for a single configurable instance component is the Salesforce Web service (see Section 2.2.2). This Web service behaves differently depending on the configuration data for each individual tenant.

Another example, albeit on a different level than Salesforce are platforms offered in a PaaS model such as Google's AppEngine. One single instance of the platform serves several tenants but isolates tenants in terms of data access and performance [Ciu09].

On the IaaS level, Amazon EC2 is another example for a single configurable instance component. Multiple tenants can run their AMI instances on the same (virtualized hardware). Tenants are isolated, due to the assumption that one tenant cannot access and manipulate the virtualization layer and thus manipulate the virtual machines of other tenants.

Multiple Instances Component.



Figure 3.11: Multiple instances component icon

Context: Customer-specific behavior is required for a component. However, the component cannot be realized as a single configurable instance component, for example, because the underlying middleware does not support it or the effort cannot be spend to re-implement it.

How can I realize customer-specific behavior and serve multiple customers when my component cannot be implemented in a single configurable instance mode?

Forces: Several reasons prevent using the single configurable instance component pattern to implement a component that must have customer-specific behavior and serve multiple customers. Reasons include middleware that does not support multi-tenancy.

Another reason could be that a component is already implemented and cannot easily be changed to a single configurable instance pattern, as it would require a complete redesign and reimplementation. In the application service provider model, this was one of the main problems as the applications to be served by the ASP were mostly not multi-tenant aware.

Sometimes location and regulatory reasons prevent the sharing of instances of components. For example, European regulations require that personal customer data can only be outsourced to countries with adequate data protection laws [CGRG06] which would prevent European companies from using components for European companies not deployed in such countries .

Solution: Use the *multiple instances component* (cf. Figure 3.11) pattern, when customer-specific behavior is required and it is not feasible to use a single configurable instance component.

Result: In the multiple instances component pattern at least one instance of a

component is deployed for each customer. Therefore this pattern is similar to the ASP model where also one instance of a whole application is deployed to serve a single customer.

By deploying a proportional amount of components corresponding to the number of customers of the application, it is ensured that each customer is served from the component. Since only one customer is served from one instance of a component this component can be widely customized.

Updating a multiple instances component can be very hard, as instances of components may be spread across different data centers, may be heavily customized or may be even deployed on different platforms. To facilitate the update of such components, a similar approach as the approach with single configurable instance components (see Section 3.3.6) can be taken. Installation of components can be automated similar to the automation of the deployment of the configuration data of the single configurable instance component. To prevent heavy uncontrolled customizations and thus making the update of a component impossible, templates and corresponding variability can be introduced similar to that in the single configurable instance component. Instead of only deploying the configuration data, the configuration data is used to automatically generate a new instance of the component and then redeploy it. This facilitates updating of the parts not affected by variability.

The development of multiple instance components is similar to the single instance components and does not require special thoughts about multi-tenancy.

However, deploying one instance per customer has some drawbacks. Since most middleware components (such as BPEL engines and application servers) do not support multi-tenancy, deploying one instance of the middleware per component requires a redeployment of the entire application stack for each customer and thus constitutes a significant overhead, as load cannot be balanced over several customers easily.

Next: In cases where components can be realized in a single configurable instance pattern (see Section 3.3.6), this pattern can be used to build services having customer-specific behavior. In case a single instance of a single instance or single configurable instance component is not enough to serve all customers,

they can be combined with the multiple instances pattern. This means that multiple instances of a component are deployed. Each of these instances then serves a fixed set of customers thus load can still be balanced across different customers, but configuration data must not be shared between all instances of the component. However, this requires a smart selection and distribution of the customers to prevent that all customers requiring high-load at the same time are put on one instance where other instances are idling because they only have low-profile customers.

Examples: One prominent example for a multiple instance component are standard applications such as a SAP system or any other ERP application. Such applications are separately installed for each company and are used on their own application stack.

Another example are the Amazon machine images at Amazon EC2. Each tenant starts the image of an application that the tenant wants to use. But since all instances are started from the same instance, updating the instance and restarting it for all tenants will update it at once for all tenants. The Amazon EC2 example shows that the same application stack can support multiple multi-tenancy patterns. While the virtualization layer of EC2 is in single configurable instance mode, the AMIs on top of it are in multiple instances mode.

Assigning Multi-Tenancy Patterns to Components.

Definition 9 (Patterns). *Each component in a model is assigned one of the three multi-tenancy patterns: (i) single instance, (ii) single configurable instance or (iii) multiple instances. The set*

$$P = \{singleInst, singleConf Inst, multipleInst\}$$

defines the set of available multi-tenancy patterns. The function $p : C \rightarrow P$ assigns a multi-tenancy pattern to a component.

3.3.7 Deployment Graph

As shown in Figure 3.5 some components must be deployed on other applications to run. In the example shown in that figure the BPEL engine is deployed on an application server and a workflow is deployed on the BPEL engine.

To express such deployment relationships, the *deployment relation* is introduced. The deployment relation connects two components via a directed connection and states that the source component must be deployed on the target component. In the example in Figure 3.5 there is, for example, a deployment relationship between the workflow component and the BPEL engine component as well as another one between the BPEL engine component and the Servlet container component.

Definition 10 (Deployment Relation). *The relation*

$$D \subseteq C \times C$$

defines all deployment relations between components in an application model. The relation is directed, where $\pi_1(d)$ represents the source component deployed on the target component $\pi_2(d)$ of a deployment relation $d \in D$.

Each component can only be deployed on a maximum of one other component and thus have only one deployment relationship in which it is the source component. Thus the following holds true for all deployments: $\forall d_1, d_2 \in D : \pi_1(d_1) = \pi_1(d_2) \Rightarrow d_1 = d_2$.

The deployment relationship is modeled as a relation and not as a containment function in this thesis because in that way the model is open to extensions where one component can be deployed on multiple other components, although this case is not further treated in this thesis. The terms relation and relationship are used interchangeably in the following in case no confusion can arise by using either of the terms. In case one component must be deployed on multiple components, this component must be modeled as multiple separate components. However, these can have the same implementation.

Figure 3.12 shows two components B and C that have a deployment relationship with a third component A . In the figure these two components are

contained in the third component.

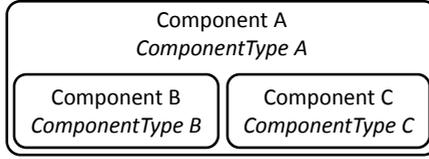


Figure 3.12: Two components contained in a third component

Components and their deployment relations span a graph, the *deployment graph*. Since a deployment relation is directed, the graph is also directed. The deployment graph is acyclic, indicating that it is not allowed to deploy a component on another that is transitively deployed on the first component. In fact, in this phase the deployment graph essentially is a forest (of trees), however, for extensibility it is modeled as a graph.

Definition 11 (Deployment Graph). *The acyclic Graph $DG = (C, D)$ which consists of components and their deployment relations is called the deployment graph of an application.*

To deploy a component on another component, this component must be available, i.e. the component *requires* this component to be present. The component that must be present to deploy a component is called *required component*. Note that the required component is only the component on which a component must directly be deployed. For example, the required component of the Workflow component in the example in Figure 3.5 is only the BPEL engine and not the underlying application server.

Definition 12 (Required Component). *For a component $c \in C$ the set*

$$\vec{c}(c) = \{cd \in C \mid \exists d \in D : c = \pi_1(d) \wedge cd = \pi_2(d)\}$$

defines the set of required components which are directly required for the deployment of component c . The set can be empty and can at most have the cardinality of one in this thesis, because one component can only be deployed on one other component.

In case a component c does not have any required components, i.e. if $\vec{C}(c) = \emptyset$, it is either a standalone component such as a server or that the application modeler does not impose any requirements on how this component should be deployed. Typically provider supplied components do not require other components, as the deployment of those component is left to the provider. In case a component that has an implementation type of *provider supplied* does have a required component, this required component is seen as a constraint for the provider. Allowing application developers to model a required component for a provider supplied component enables to model scenarios such as the one depicted in Figure 3.5 where the BPEL engine (which is provider supplied) must be provisioned on a different application server than the one hosting the GUI component.

The set of *transitively required components* for a component c is defined as the set of components that are *reachable* from component c via deployment relationships. Reachable means that there exists a path through the deployment graph connecting any component in the set of transitively required components to the component c via outgoing deployment relations from component c . In practice, this is the set of components that must be available before component c can be deployed on them.

Definition 13 (Transitively Required Components). *The set of transitively required components for a component $c \in C$ is defined as the set of components to which a path of deployment relations exist from component c :*

$$C^+(c) = \{cd \in C \mid \exists d_1, d_2, \dots, d_{n-1}, d_n \in D : \\ \pi_2(d_n) = cd \wedge \pi_1(d_1) = c \wedge \pi_2(d_{k-1}) = \pi_1(d_k) \text{ for } 2 \leq k \leq n\}$$

3.3.8 Atomic and Composite Components

In general two types of components can be distinguished: *atomic components* and *composite components*. Atomic components are components that do not have a deployment relation with other components.

Definition 14 (Atomic Components). *The set of atomic components is defined*

as:

$$C_{atomic} = \{c \in C \mid \neg \exists d \in D : \pi_1(d) = c\}$$

Composite components are components that depend on another component to be executed, i.e. that have one deployment relation to another component.

Definition 15 (Composite Components). *The set of composite components is thus defined as:*

$$C_{composite} = \{c \in C \mid \exists d \in D : \pi_1(d) = c\}$$

Component types that can be part of a composite component, i.e., allow other components to have a deployment relationship with them are called *provider component types*. To specify which other components can be provisioned on a provider component type each provider component type specifies a set of implementation types that it can host. For example, the component type *Servlet Container* specifies that it can host components that are implemented using the implementation type *Java Web Application*.

Definition 16 (Allowed implementation types for deployments on a component of a component type). *The map $ad : CT \rightarrow \rho(IT)$ returns a set of implementation types for a component type. The semantics is the following: Components implemented using one of the returned implementation types can have a deployment relationship on components of the component type. Thus this definition restricts the possible deployment relationships. Constraint 3 formally defines all allowed deployment relations between components restricted by this map.*

Definition 17 (Provider Component Types). *A provider component type is defined as a component type where the set of allowed deployment implementation types is not empty.*

$$CT_{provider} = \{ct \in CT \mid ad(ct) \neq \emptyset\}$$

The following constraint must then hold for all deployment relations:

Constraint 3 (Deployment relations must be allowed deployment relations). For each deployment relation the provider component type must allow the implementation type to be deployed on that component type. A special case is if the implementation type of a component is provider supplied and that component has a deployment relation on another component. This is allowed if the provider component also has an implementation type of provider supplied.

$$\forall d \in D : \text{implT}(\text{impl}(\pi_1(d))) \in \text{ad}(\text{type}(\pi_2(d))) \cup \{\text{providerSupplied}\}$$

3.3.9 Serialization of the Cafe Application Metamodel

To exchange Cafe applications between application vendors and providers, a format in which the Cafe application metamodel is serialized is required. In the following the *Cafe application descriptor* is introduced. A Cafe application descriptor is an XML document containing the same information that is contained in an instance of the Cafe application metamodel. The syntax of the XML document is defined by the *Cafe application descriptor schema*. The complete Cafe application descriptor XML schema can be found in [Mie10a]. Listing 3.1 shows a pseudo schema for the application model serialization.

Listing 3.1: Application model pseudo schema

```

1 <applicationModel targetNamespace="URI"
  name="String" type="URI"
3  pattern="multipleInstances | singleConfInstance | SingleInstance"
  xmlns="http://www.iaas.uni-stuttgart.de/cafe/schemas/
  applicationDescriptor">
5  <component name="NCName" type="URI" ?
    pattern="..."> +
7    <component name="NCName"
      pattern="...">*
9    <implementation type="URI">
      <file>String</file>*
11  </implementation>

```

```

13     <component/>
14     <implementation type="URI" />
15 </component>
16 <implementation type="URI">
17     <file>String</file> *
18 </implementation>
19 </applicationModel>

```

Listing 3.2 shows the XML-serialization of an example application model. The application model shown in Listing 3.2 contains three components. A *Workflow* component that must be deployed on a provider supplied *Workflow Engine* component of component type *ApacheOde*. In addition to that, an external *CRM* component is defined in the model. The *Workflow* component is implemented by an implementation of type *BPEL*. The implementation contains one file that is referenced via a relative path. The *CRM* component is defined with an implementation type of *external*. Therefore, an EPR is annotated to that component indicating where this component can be accessed.

Listing 3.2: Application model example XML

```

2 <applicationModel targetNamespace="http://t1"
3   name="Template1" type="t3:WorkflowAppType"
4   pattern="multipleInstances"
5   xmlns:t1="http://t1" xmlns:v1="http://v1"
6   xmlns:cf="xmlns="http://cafe" xmlns:j1="http://j1"
7   xmlns:t2="http://t2" xmlns="http://cafe"
8   xmlns="http://www.iaas.uni-stuttgart.de/cafe/schemas/
9     applicationDescriptor">
10 <component name="WorkflowEngine" type="t2:ApacheOde"
11   pattern="multipleInstances">
12 <component name="Workflow"
13   pattern="multipleInstances">
14 <implementation type="j1:BPEL">
15 <file>/component1/workflow.zip</file>
16 </implementation>
17 </component>
18 </component/>

```

```

16     <implementation type="cf:providerSupplied" />
    </component>
18 <component name="CRMComponent">
    <implementation type="external">
20     <epr>...</epr>
    </implementation>
22 </component>
    <implementation type="t1:sampleApplication">
24     <file>applicationDescriptor.xml</file>
    </implementation>
26 </applicationModel>

```

3.4 Cafe Variability Metamodel

Variability of a Cafe application is an essential part of Cafe, since the variability describes which parts of the application can be customized and what are the allowed values for customization. In this section the variability metamodel for Cafe is described in detail. At first the requirements for the Cafe variability mechanism are stated. Then the *variability metamodel* is introduced. A formalization of the main concepts of the variability metamodel is given and its serialization in form of *variability descriptors* is described. In contrast to the application descriptor that was given at the end of the introduction of the application metamodel, the individual serializations of elements of the variability descriptor are given when the elements are introduced because the variability metamodel is considerably more complex than the application model.

3.4.1 Requirements for a Variability Mechanism

To describe variability in Cafe, a mechanism is required, allowing Cafe application vendors to specify variability in Cafe application templates (cf. Figure 3.1). For example, there might be variability in the user interface (i.e. the modification of a text, logo or background color) as well as in the process layer (i.e. the choice of a set of predefined process models) or the database layer

(i.e. custom database fields). Since all these layers can be implemented in different programming languages, an orthogonal variability model is needed. Thus the approach must be technical enough, such that application vendors can use the mechanism to specify variability at a code level. This means that the bound variability model and the code contained in the application template must be expressive enough such that deployable and executable code can be generated by combining these two. Despite this requirement which mandates dealing with concrete implementation formats, the variability concept must be flexible enough such that it can be used across different artifacts that realize the implementation of different components in an application template.

Thus a variability mechanism is needed that allows defining variability points and their dependencies in all kinds of artifacts of the application template. In Section 2.1.4 related approaches such as the orthogonal variability model (OVM) are discussed and several shortcomings have been shown. In this section, the similarities and differences to OVM and other variability models are highlighted during the definitions of some of the artifacts of the Cafe variability model.

In addition to the requirements of concreteness and implementation technology independence, the mechanism must abstract from the concrete technology in use such that customers can customize an application without knowing the implementation details of this application.

Furthermore, it must be possible to define complex dependencies between different variability points. This is important in order to express complex customization options such as “alternative B1 for variability point B can only be taken in case alternative A1 at variability point A has already been taken”.

3.4.2 Short Overview

Variability in Cafe application templates is expressed through a *variability model*. The variability model is an instance of the variability metamodel. An *abstract variability model* is a variability model that is specified by component types to define the variability for a component type. Variability models can be imported into other variability models. A variability model consists of a

set of *variability points* defining the variability and a set of *variability point dependencies* describing the dependencies between variability points. Variability points must be bound during one concrete *phase* during the set-up of an application. A variability point is optionally associated with a concrete artifact that it configures via a *locator*. A variability point also defines a set of *alternatives*, that describe alternative values for that variability point. Different types of alternatives exist, such as *free alternatives* that allow customizers to enter arbitrary values or *explicit alternatives* that pre-define the values that can be selected at a variability point. Alternatives can be restricted by *enabling conditions* that define which alternatives are enabled under a certain condition. Imported variability points can be refined by *variability point refinements* that, for example, restrict the number of alternatives that can be selected. Figure 3.13 shows the main elements of the variability metamodel. The individual concepts are described in detail in the sections below.

3.4.3 Example Variability Model

Figure 3.14 shows an example variability model for the example application presented in Figure 3.5. Note that the example presented here does not contain all variability that would be necessary to actually deploy the example application. The purpose of this example variability model is to introduce the main concepts of the variability metamodel. The example variability model contains four variability points: The *ApplicationTitle* variability point, the *BPELEngineHostName* variability point, the *SetWorkflowEPR* variability point and the *JEEVersion* variability point. It also contains one variability point refinement, the *JEEVersion* variability point refinement which refines the *JEEVersion* variability point.

The *ApplicationTitle* variability point must be bound during the solution engineering phase by a customer, as indicated by the circle with the “S” in the top right corner. This variability point contains two alternatives, the *StandardTitle* and *CustomTitle* alternatives. The *StandardTitle* alternative is an explicit alternative which contains the value of the standard title “SMS Sending Application”. The *CustomTitle* alternative is a free alternative, where

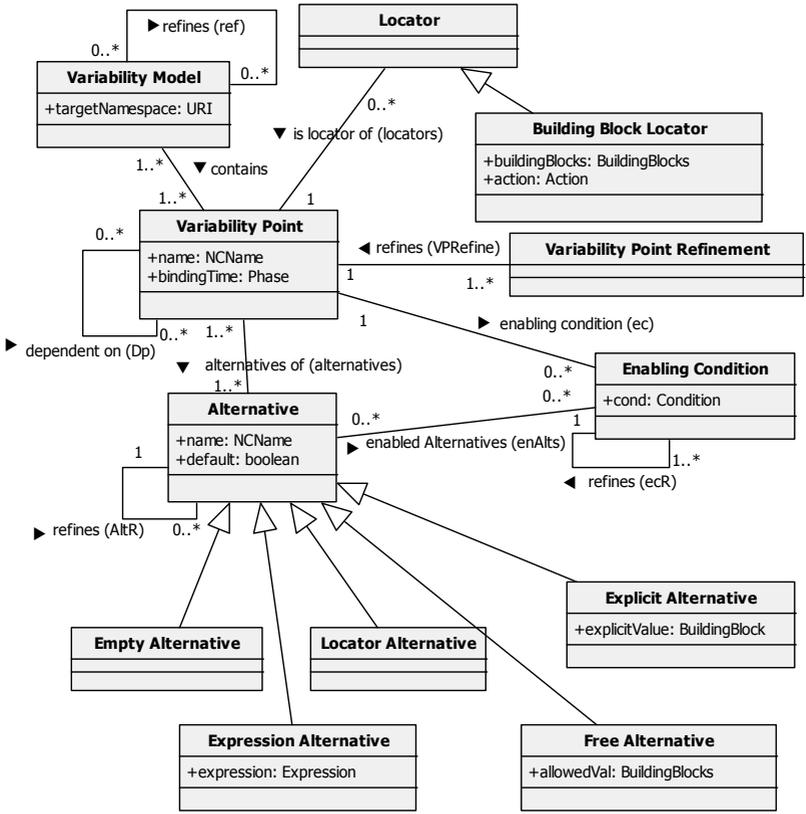


Figure 3.13: Variability metamodel

a customer can enter his own title for the application. The *ApplicationTitle* variability point contains one XPath locator that points into a deployment descriptor file (`/WEB-INF/web.xml`) in the `Gui.war` file where the title of the application can be configured. As shown in Figure 3.5 the `Gui.war` file is part of the implementation of the *GUI* component, thus this variability point targets the GUI component.

The *BPEngineHostName* variability point is imported from the abstract

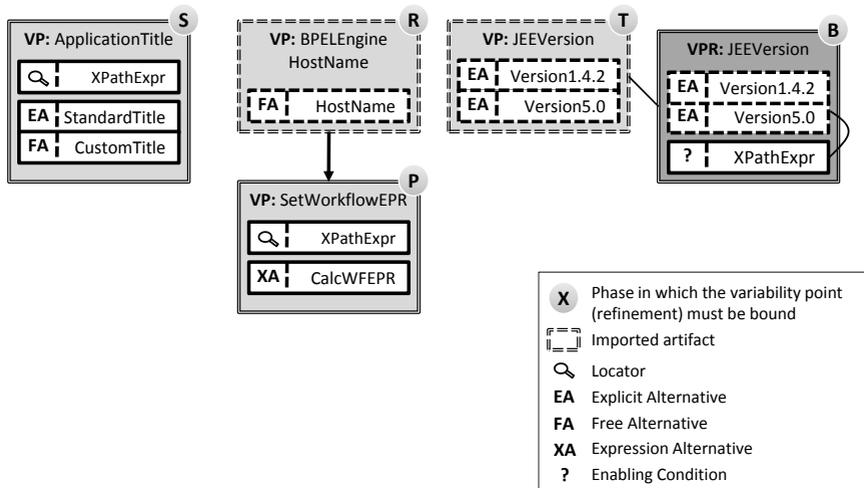


Figure 3.14: Example variability model

variability model of the *Apache ODE* component type and thus targets the *BPEL Engine* component in the example application. It is bound at runtime (as indicated by the “R”). The *BPELEngineHostName* variability point contains one free alternative, the *HostName* alternative, that is bound by the provisioning engine after the component has been provisioned and its concrete host name is known. This variability point does not contain a locator, as it does not configure any document directly.

The *SetWorkflowEPR* variability point depends on the *BPELEngineHostName* variability point and must be bound at pre-provisioning time. It specifies one alternative, the *CalculateWorkflowEPR* (*CalcWFEP*) alternative, containing an XPath expression that concatenates the selected value of the *BPELEngine-HostName* variability point with the relative path under which the workflow component is available at that host. This variability point also contains a locator that sets the EPR with the calculated value in the workflow.wsd file in the *Gui.war* file. Thus it affects the *GUI* component of the example application at pre-provisioning time.

The *JEEVersion* variability point is imported from the abstract variability model of the *JEE Application Server* component type. It defines two explicit alternatives *Version1.4.2* and *Version5.0*. As the example application can only run on JEE Version 5.0 it refines the *JEEVersion* variability point with the *JEEVersion* variability point refinement. In this refinement a refinement enabling condition is specified in form of an XPath expression that always evaluates to true and only enables alternative *Version5.0*. The variability point refinement is specified to be bound at component binding time, whereas the *JEEVersion* variability point must be bound at template customization time. The variability point refinement implicitly states that only components of type *JEE Application Server* that were customized to support JEE Version 5.0 can be used for this application.

3.4.4 Formal Definition

Definition 18 (Variability Model). *Let VM be a set of variability models, then a variability model $V \in VM$ is formally defined as a tuple*

$$V = (VP, VPR, Dp, L, Alt, EC, Co, E, ref, locators, alternatives, ec, cond, enAlts, expression, buildingBlocks, roles, VPrefine, AltR, eCR)$$

where

- *VP is a set of variability points*
- *VPR is a set of variability point refinements*
- *Dp are dependencies between variability points*
- *L is a set of locators*
- *Alt is a set of alternatives*
- *EC is a set of enabling conditions*
- *Co is a set of conditions*
- *E is a set of expressions*
- *ref is a (variability model, variability model) refinement map*
- *locators is a (locator, variability point) map*

- *alternatives* is an (alternative, variability point) map
- *ec* is an (enabling condition, variability point) map
- *cond* is a (condition, enabling condition) map
- *enAlts* is an (alternative, enabling condition) map
- *expression* is an (expression, alternative) map
- *buildingBlocks* is a (building block locator, building blocks) map
- *roles* is a (variability point, role) map
- *VRefine* is a (variability point, variability point refinement) map
- *AltR* is an (alternative, alternative) refinement map
- *ecR* is an (enabling condition, enabling condition) refinement map

Additional Properties of a Variability Model. Besides the formalized properties above, a variability model has additional properties that are explained in Table 3.1. Only those additional properties that are necessary for the understanding of the concepts are shown in the tables in order to emphasize their importance.

Property	Type	Explanation
Name	NCName	The name of the variability model
Description	String	A short description that describes the variability model
TargetNamespace	URI	The XML target namespace for which all elements in the variability model are defined

Table 3.1: Additional properties of a variability model

XML serialization of the variability model as a variability descriptor. A variability model is serialized as a variability descriptor which is a special XML format to describe the variability model for an application template. Listing 3.3 shows the pseudo schema for a variability descriptor.

Listing 3.3: Variability descriptor pseudo XML schema

```

1 <variabilityModel targetNamespace="URI" name="NCName"
2   xmlns="http://www.iaas.uni-stuttgart.de/cafe/schemas/
   variabilityDescriptor">
3   <description>...</description> ?
4   <import>...</import> *
   <variabilityPoint>...</variabilityPoint> +
5   <dependency>...</dependency> +
6 </variabilityDescriptor>

```

Abstract Variability Models. Two types of variability models can be distinguished. *Concrete variability models* (also called variability models for short) and *Abstract variability models*. A concrete variability model is a variability model that describes the complete variability for a component (which can be an application template). An abstract variability model describes the *variability interface* of a component type. The variability interface describes which variability is offered in general by a component type. Thus all components which are of that component type must support this variability. Abstract variability models and concrete variability models can contain all entities of the variability metamodel. Since the Cafe application metamodel allows the recursive aggregation of components, the variability metamodel must also allow this. Note that abstract variability models can also refine other abstract variability models. Thus a mechanism is needed to import the abstract variability models of aggregated components into the variability model of the aggregating application template. Also a mechanism is needed so that a concrete variability model can refine an abstract variability model. Thus a (concrete) variability model can be standalone or can *refine* one or more abstract variability models. Refinement means that variability points of the imported variability models can be modified following a strict set of rules what can be modified and what not. These rules are given in this section as well as in Section 3.4.4 and especially Section 3.4.5.

Figure 3.15 shows an application model containing one internal component (*Component 1*) that must be deployed on a provider supplied component

(Component 2). Component 2 is of component type C which has an associated abstract variability model. The application model itself is of the component type A which in turn has an abstract variability model. The variability model for the application model is thus comprised out of a part that *refines* the abstract variability model for component type C, for example, by refining certain alternatives. It also contains a part that refines the abstract variability model for component type A which means that it contains all artifacts of that abstract variability model and extends them with additional artifacts needed to adapt this abstract variability model to the concrete context of this application template. The variability model for the application template can also contain other variability points that are not refinements of any imported abstract variability model. How this is done in detail and which artifacts are allowed during the refinement is explained below.

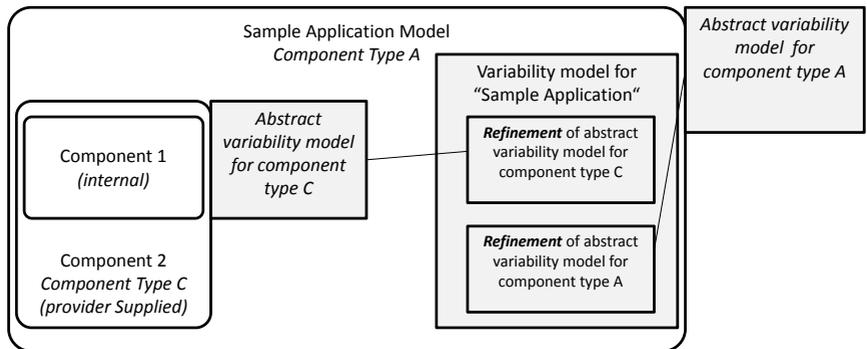


Figure 3.15: Refining variability models

Refining Variability Models. As shown in Figure 3.15, variability models can refine other variability models. The semantics of such a refinement is the following: In case a variability model refines another variability model, it automatically contains all variability points of the refined variability model and their dependencies. For each of these imported variability points the choices that can be taken at the variability point can be restricted in the refining variability model. Formally the refinement mechanism is defined as a reference

to a set of other variability models.

Definition 19 (Refinement of a Variability Model). *The map $ref : VM \rightarrow \rho(VM)$ assigns a set of variability models to a variability model that this variability model refines. Note that this is an informal definition and that each artifact that can be refined is defined in the definition of the respective artifact.*

The concrete semantics of a imported variability model imported via the refinement mechanism is as follows: All artifacts declared in the imported variability model are added to the respective sets of the refining variability model. Therefore, all following set definitions are defined to include the artifacts imported from other variability models via refinement. To avoid name clashes an additional mechanism is needed.

Listing 3.4 shows the XML serialization of the refinement mechanism. Each imported variability model is imported via its name space as indicated by the `namespace` attribute. The name space is then bound to a concrete variability descriptor via the `location` attribute. The optional `referencingNamespace` attribute maps the imported namespace to a name space that is unique in the importing variability descriptor. This is necessary as for example abstract variability models of one component type can be refined multiple times for different components of that type and thus must be imported multiple times. To be able to reference artifacts of an abstract variability model that is refined several times, a separate name space can be given to each of the imported abstract variability models. The `component` attribute references the component (via its QName as defined in the application descriptor) for which the variability model is imported.

Listing 3.4: Import pseudo schema

```
1 <import namespace="URI" location="String "  
   referencingNamespace="URI" ?  
3   component="QName" />
```

Associating abstract variability models with component types. As said in Section 3.3.4 component types have an associated abstract variability model.

Definition 20 (Abstract variability model of a component type). *The function $avm : CT \rightarrow VM$ associates a variability model with a component type.*

Variability Points. The most important entity of the Cafe variability model is the *variability point*. In software product line engineering the concept of a *variation point* [JGJ97, BFG⁺02, PBvdL05] is well-known. Jacobson et al. [JGJ97] define a variation point as “one or more locations at which the variation will occur”. Pohl et al. [PBvdL05] define a variation point as “a representation of a variability subject within domain artifacts enriched by contextual information.” As a generalization a variation point can be seen as one or more locations in a software product line that are variable and that need to be bound when creating a new product line member.

A variation point as used in software product line engineering corresponds to a location in a Cafe application template where that template must be customized. To reflect this similarity, but distinguish it from the variation point, the term *variability point* is used in Cafe template engineering.

A variability point defines one or more locations and possible values for that locations of a Cafe application template that need to be customized. Customization can be done by a customer or the provisioning infrastructure before the application can be used.

Definition 21 (Variability Points). *The finite set $VP = \{\dots, vp_i, \dots\}$ is a set of variability points. The set $VP_{vm} = \pi_1(vm)$ is the set of variability points of a variability model $vm \in VM$. This includes the variability points imported via the refinement mechanism from other variability models. Thus, the following condition always holds true for a given variability model $vm \in VM$: $\forall w \in ref(vm) : \pi_1(w) \subseteq \pi_1(vm)$, i.e., the set of variability points of any imported variability model is always a subset of the set of variability points of the importing variability model.*

Variability Binding Times and Roles. Variability models describe variability of application templates that can be bound by different roles such as the provider or a technical person or a marketing person on the customer side. Also variability can be bound in different phases during the life-cycle of a

component. For example, variability regarding the quality of service as well as functional variability (such as the decision which type of payment is allowed, or the choice of a background color) is bound at solution engineering time. Variability such as the concrete definition of the endpoint reference to use, may be bound at pre-provisioning time or even at runtime.

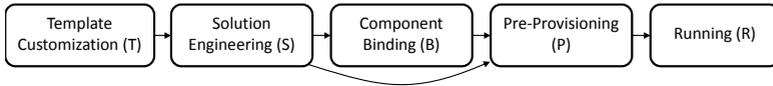


Figure 3.16: Phases during the provisioning of a component

Regarding the overall Cafe development process (cf. Figure 3.3), five basic phases can be defined for each component: The template customization, solution engineering, component binding, pre-provisioning and running phase. These five phases in the life-cycle of a component are defined as follows, the characters in parenthesis denote the abbreviation for the respective phase that will be used in the remainder of this thesis:

- *Template Customization (T)* denotes that the application template containing the component is configured by a provider to be offered by that provider.
- *Solution Engineering (S)* denotes that the application template containing the component, and thus the component is currently customized by a customer during the solution engineering phase.
- *Component Binding (B)* denotes that a provider supplied component is currently bound against a concrete provisioning service that can provision such a component or an already provisioned component. Only provider supplied components can be in that state.
- *Pre-provisioning (P)* denotes that the component is customized and has an assigned provisioning service or is assigned to a component already provisioned. However, the component is not yet provisioned.
- *Running (R)* denotes that the component is already provisioned and running.

Figure 3.16 shows the ordering of the aforementioned phases.

Definition 22 (Phases in which variability points can be bound). *The set*

$Phases = \{template\ customization, solution\ engineering, pre-provisioning, component\ binding, running\}$

defines the phases during the provisioning of a component.

Definition 23 (Variability bind times). *The function $bindT : VP \rightarrow Phases$ assigns a binding time to a variability point. The semantics is that the variability point must be bound during the assigned phase.*

Definition 24 (Variability points that must be bound in a phase). *The set of variability points $VP(p)$ that must be bound in a phase $p \in Phases$ is defined as follows:*

$$VP(p) = \{vp \in VP \mid bindT(vp) = p\}$$

Definition 25 (Variability binding roles). *The set $RL = \{\dots, rl_i, \dots\}$ is the set of all roles that can be required to perform a customization.*

The map $roles : VP \rightarrow \rho(RL)$ assigns a set of roles allowed to perform the binding to a variability point.

The set of roles contains a subset containing all roles that are performed by the customer. The set $RL^{Cust} \subseteq RL$ contains these roles. Examples for such roles could be *UI Expert* for someone that must customize UI related variability points of an application, *Domain Expert* that must customize functional variability of the application or *QoS Expert* that customizes the QoS properties of the application.

To further qualify and group variability points, they can be tagged with arbitrary tags. Such tags could, for example, be *SLA* to denote SLA-related variability points or *GUI* to denote GUI-related variability points. The tags serve only informational purposes and providers can exploit it to mine customizations to derive information such as which customers have the same GUI settings or SLA settings. Tags are not formally defined here.

Property	Type	Explanation
Name	NCName	A name, uniquely referencing the variability point in the namespace of the variability model
Description	String	A short description that describes the variability point
Tags	List of Strings	A set of tags associated to the variability point that can be used to group variability points in tools

Table 3.2: Additional properties of a variability point

Additional Properties of a Variability Point. Besides the formalized properties above, a variability point has additional properties that are explained in Table 3.2.

XML Serialization of a Variability Point. Listing 3.5 shows the serialization of a variability point. The description and documentation elements have been omitted from the listing as well as from the formalization below. They can be found in the XML Schema in [Mie10b]. Note that the `alternative` tag represents the super type from which all alternatives and their tags shown in Listing 3.8 are derived. A variability point has a name attribute that must be unique among all variability points in the namespace it is declared in.

Listing 3.5: Variability point pseudo schema

```

1 <variabilityPoint name="NCName" role="QName"
    bindingTime="templateEngineering | solutionEngineering |
    componentBinding | preProvisioning | running">
    <tag>Process Variability</tag> *
3 <locator>...</locator> *
    <alternative>...</alternative> +
5 <enablingCondition>...</enablingCondition> *
</variabilityPoint>

```

Variability Point Refinements. A variability point refinement refines one variability point, i.e, it limits the choices that can be bound at that variability point.

Definition 26 (Variability point refinement). *The set $VPR = \{\dots, vpr_i, \dots\}$ denotes a finite set of variability point refinements. The set $VPR_{vm} = \pi_2(vm)$ is a set of variability point refinements in a variability model $vm \in VM$ including those imported from other variability models.*

The variability point refinement to variability point map $VPrefine : VPR \rightarrow VP$ assigns a variability point to a variability point refinement that this variability point refinement refines.

Serialization of variability point refinements. Listing 3.6 shows the pseudo-XML schema for the serialization of variability point refinements.

Listing 3.6: Variability point refinement pseudo schema

```

<variabilityPointRefinement name="NCName">
2  <originalVariabilityPoint>QName</originalVariabilityPoint>
   <tag>...</tag>*
4  <locator>...</locator>*
   <explicitAlternative>...</explicitAlternative>*
6  <expressionAlternative>...</expressionAlternative>*
   <emptyAlternative>...</emptyAlternative>*
8  <locatorAlternative>...</locatorAlternative>*
   <freeAlternative>...</freeAlternative>*
10 <enablingCondition>...</enablingCondition>*
</variabilityPointRefinement>

```

The refined variability point is referenced in the importing variability model via its QName.

Locators. To specify locations in a Cafe application that are affected by a variability point or a variability point refinement, a variability point or variability point refinement can contain one or more *locators*. A locator specifies what should be done with the value obtained during the binding of a variability point. Different types of locators exist. Two types of locators are defined in this thesis. The *user defined locator* and the *building block locator*. The user defined locator indicates that some specific functionality outside the variability model must be implemented specifying what should happen with the value

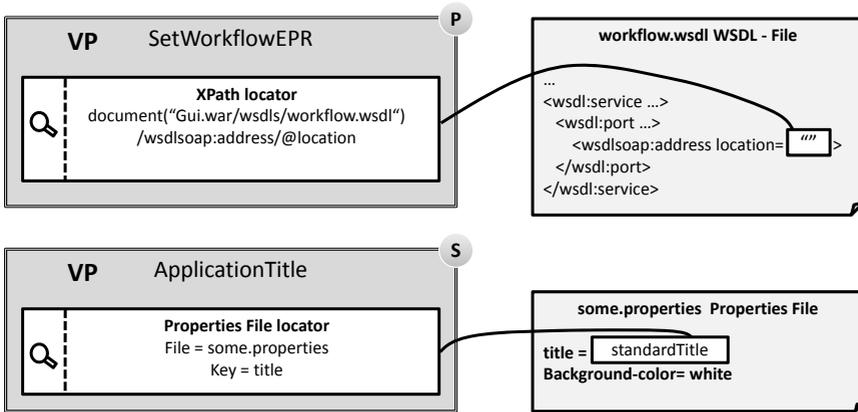


Figure 3.17: Examples of an XPath and a properties file locator

obtained from the variability point binding. A building block locator points into a concrete file of an implementation, therefore the format of a locator is very specific to the format of the artifact. For example, a building block locator pointing into an XML document can specify the exact location in the XML it points to via an XPath [W3C99] expression. A building block locator pointing into a properties file with key-value pairs specifies the name of the key to denote the concrete location it points to. For this formalization it is enough to denote that a building block locator points into a building block of a file of an implementation of a component. For the definition of building blocks, files and implementation see Section 3.3.5 and Figure 3.8.

Figure 3.17 shows an example of an XPath and a properties file locator pointing into an XML file and a properties file via an XPath expression and a key. Any type of building block locator can be specified for arbitrary documents. In this thesis two types of building block locators are described:

- *XPath locator* is a building block locator to point into arbitrary XML documents. An XPath locator always begins with a `document()` function that points into the file where the variability is located. The second part of the locator is the XPath location path that points to one or more locations

in the document. The semantics of pointing to multiple locations is that those locations are bound simultaneously when the variability point containing the locator is bound. Exactly one document and location path string is allowed per locator.

- *Properties file locator* is a building block locator to point into properties files that are built following a (key,value) scheme. The properties file locator references the document to point into and a name of the key where the value should be changed.

Definition 27 (Building Block Locators). *Formally a building block locator is a (file, set of building blocks) tuple specifying the exact location of one or more building blocks in one file. However, given the definition of building blocks, each building block is assigned to exactly one file, therefore the file part can be omitted in the formal definition and the locator can be defined as a subset of the building blocks of an application and an action. The action denotes if the building blocks referenced by a locator must be replaced during the customization or if the values selected during customization must be placed in front of or behind the referenced building blocks.*

The set $BL = \{\dots, bl_i, \dots\} = \rho(B) \times \{\text{replace, before, after}\}$ defines the set of building block locators pointing to any set of building blocks used to implement an application template or component and the associated action that must be taken during customization.

Definition 28 (Locator). *The set $L = \{\dots, l_i, \dots\} \cup BL$ defines the set of locators that can be used for variability points and variability point extensions to define what should be done with the value selected during the binding of that variability point.*

The function $\text{locators} : VP \cup VPR \rightarrow \rho(L)$ assigns a set of locators to a variability point or variability point refinements. Locators can be associated with variability point refinements to adapt them to concrete implementations of components. A variability point in an abstract variability model typically does not specify any locator since the locator is very specific to how a component is implemented. Locators are then typically added in variability point refinements in the variability model of the component that is of the component type defining the

abstract variability model.

The following conditions must hold for all building block locators:

$$\forall l \in BL, \forall b_1, b_2 \in \pi_1(l) : blockOf(b_1) = blockOf(b_2)$$

i.e. each building block locator can only contain building blocks that are contained in one file.

XML serialization example. Listing 3.7 shows the example serialization of an XPath locator pointing into the title of an XHTML document. Note that an XPath processor which later processes the XPath must be aware of the namespaces used in the XPath expression (such as the `xhtml` namespace in this case).

Listing 3.7: XPath locator example XML

```
1 <locator action="replace">
  <xPathLocator xmlns:xhtml="...">
3   document("sample.html")//xhtml:title
  </xPathLocator>
5 </locator>
```

Alternatives. Similar to the variation point in software production line engineering, a variability point contains the possible variants that can be selected by a customizer to bind that variability point. However, our model is different from the variant model, for example, used in OVM [PBvdL05, PM06]. While OVM and other variability models such as FORM [KKL⁺98] or FODA [KLD02] from software product line engineering allow selecting several variants to bind one variability point, the Cafe variability model only allows to select one variant. To make this distinction explicit we call variants *alternatives* in the Cafe context, as they are mutually exclusive. The reason for only allowing one alternative to be selected is that the value of this alternative must be copied to where the locator points to. Since the variability models in software product line engineering operate on a higher level than the code level on which the Cafe variability model operates, multiple variants can be selected.

Definition 29 (Alternatives). *An alternative is one possible value that a customizer can select to bind a variability point. The set $Alt = \{\dots, a_i, \dots\}$ is the set of all alternatives of a variability model including those imported from other variability models.*

The function $alternatives : VP \cup VPR \rightarrow \rho(Alt) \setminus \emptyset$ assigns a non-empty set of alternatives to a variability point, or a variability point refinement. One alternative can be used in multiple variability points. Thus they are declared in a variability model and not at a variability point.

An alternative must have a name that is unique among all alternatives in the namespace of a variability model.

Alternative Types. Two general types of alternatives can be distinguished. The first type of alternatives allows the customizer to choose between one of a predefined set of optional values, whereas the second type allows the customizer to enter own values.

Alternative types that allow the user to choose between predefined values are: *Explicit* alternatives, *Expression* alternatives, *Empty* alternatives and *Locator* alternatives. The alternative type to enter a free value at a variability point is the *Free* alternative. Variability models such as OVM do not offer free alternatives only the choice between predefined variants [PBvdL05, MMLP09]. This is due to their different focus which is on composing different features in an application architecture rather than customizing pre-defined application templates. The different alternative types are explained below

Definition 30 (Types of Alternatives). *The set of alternatives A of a variability model, including imported variability models, is the union of the sets of the sets of alternatives of different types.*

$$Alt = Alt^{Explicit} \cup Alt^{Expression} \cup Alt^{Empty} \cup Alt^{Locator} \cup Alt^{Free}$$

Where $Alt^{Explicit}$ denotes the set of explicit alternatives, $Alt^{Expression}$ the set of expression alternatives, Alt^{Empty} the set of empty alternatives, $Alt^{Locator}$ the set of locator alternatives and Alt^{Free} the set of free alternatives.

Explicit Alternative. An explicit alternative $exa \in Alt^{Explicit}$ contains a building block that is predefined in the template engineering phase. The building block is the same type of building blocks as used in the definition of implementations of components (see Section 3.3.5). Several explicit alternatives can be used to offer a decision between multiple values during customization.

Definition 31 (Value of an explicit alternative). *The function $explicitValue : Alt^{Explicit} \rightarrow B$ assigns an explicit value in form of a building block to an explicit alternative.*

Two examples illustrate the explicit alternative approach. In the first example, the customizer can choose between two mutually exclusive activities to bind a variability point in a BPEL process. Therefore, two explicit alternatives containing the complete BPEL code of the corresponding activity are specified. Another explicit alternative at another variability point could specify a value for the background color of a Web application.

Empty Alternative. An empty alternative $ema \in Alt^{Empty}$ denotes that the location in the artifact the locator points to can be replaced with the empty string. This alternative is a shortcut for an explicit alternative with an empty value. An example for an empty alternative is the removal of an activity from a BPEL process. To specify this an empty alternative is used.

Definition 32 (Value of an empty alternative). *The function $emptyValue : Alt^{Empty} \rightarrow \{\perp\}$ returns an empty value for each empty alternative.*

Expression Alternative. An expression alternative $exp \in Alt^{Expression}$ contains an expression that points to a building block in a file or at an already bound variability point. The result of the expression is one value which could be a set of building blocks that can be chosen by the customizer to fill the associated variability point. The expression must be evaluated when the variability point is presented to the customer to be bound. Therefore, the value for the expression alternative is computed dynamically and can reflect the binding of previous variability points. As with the locators, multiple expression languages are allowed. The standard way used in the Cafe prototype to describe expressions is XPath.

Definition 33 (Expressions). *The set*

$$E = \{\dots, e_i, \dots\}$$

is the set of all expressions used in expression alternatives of a variability model including those imported from other variability models.

An expression references a set of building blocks. The function `buildingBlocks` : $E \rightarrow \rho(B)$ returns all building blocks referenced by an expression.

The function `expression` = $Alt^{Expression} \rightarrow E$ assigns an expression to an expression alternative

An example for an expression alternative could be the value of an already bound variability point which is expressed in the Cafe prototype through a special XPath function, the `selectedValue(variabilityPoint)` function. More on special XPath functions can be found in Section 3.4.4.

Locator Alternative. A locator alternative $la \in Alt^{Locator}$ implicitly points to the value of the location(s) the locator(s) of the variability point refer to. The semantics of the locator alternative is the following: if a locator alternative is selected at a variability point, the locations in a document targeted by the locator(s) of that variability points are not changed.

Definition 34 (Building blocks referenced by a locator alternative). *The set of building blocks B_{la} referenced by a building block locator alternative la is therefore defined as follows:*

$$B_{la} = \{b \in B \mid \exists vp \in VP : b \in locators(vp) \wedge la \in alternatives(vp)\}$$

Free Alternative. A free alternative $fa \in Alt^{Free}$ is an alternative that allows the customizer to enter a value that is not predefined, i.e. the customizer can define a new building block that replaces the building blocks the locator(s) of that variability point refer to.

Definition 35 (Value entered at a free alternative). *The function `freeValue` : $Alt^{Free} \rightarrow B \cup \{\perp\}$ returns the building block entered at a free alternative. In case none has been entered \perp is returned.*

Free alternatives can be annotated with a condition that restricts the input, for example, to instruct a customizer that the value to enter should be in a specific range and of a specific type. Restrictions can be in any expression language. Again the default language is XPath. In case of an XPath restriction, the special variable $\$input$ (cf. Listing 3.8) is used to access the input a customizer has made in the XPath expression. The XPath expression must return true, otherwise the input must be rejected.

Since it is not the task of this thesis to formalize expression languages, the set of allowed values is defined as a set of discrete building blocks. In reality this is, for example, the set of all strings or the set of all values between 0 and 15 or all BPEL activities, depending on the restriction.

Definition 36 (Restriction of values allowed at a free alternative). *The free alternative building block map $allowedVal : Alt^{Free} \rightarrow \rho(B)$ assigns a set of building blocks that are allowed at the free alternative.*

Combination of alternative types. For one given variability point, several alternatives can be specified. These alternatives can be of any of the above types and can be an arbitrary combination of alternatives of the different types. A variability model can, for example, contain a variability point with three alternatives, one locator alternative, one explicit alternative and one free alternative. This variability point is used to denote that the customizer can choose between two predefined values (the one referenced by the locator and the one referenced by the explicit alternative) and a user-defined value (described by the free alternative).

Definition 37 (Possible values for an alternative). *The map $possVal : Alt \rightarrow \rho(B)$ returns the set of building blocks referenced by an alternative. It is defined as follows, depending on the alternative type for a given alternative $a \in Alt$.*

$$\text{possVal}(a) \mapsto \begin{cases} \text{explicitValue}(a) & a \in \text{Alt}^{\text{Explicit}} \\ \text{buildingBlocks}(a) & a \in \text{Alt}^{\text{Expression}} \\ B_{l_a}(a) & a \in \text{Alt}^{\text{Locator}} \\ \text{allowedVal}(a) & a \in \text{Alt}^{\text{Free}} \\ \perp & a \in \text{Alt}^{\text{Empty}} \end{cases}$$

Definition 38 (Building blocks referenced by locators of a variability point). For a given variability point $vp \in VP$, the set

$$B^{\text{Loc}}(vp) = \{b \in B \mid b \in \pi_1(l) \wedge l \in \text{locators}(vp) \cap BL\}$$

contains all building blocks that are referenced by one of the locators of the variability point.

Definition 39 (Variability points affecting a component). Thus all variability points affecting an (internal) component $c \in C$ can be defined as the set of variability points whose locators point at least to one building block of a file in an implementation of component c :

$$VP(c) = \{vp \in VP \mid \exists b \in B^{\text{Loc}}(vp) : b \in B(c)\}$$

Serialization of Alternatives. Listing 3.8 shows an example serialization of all alternative types. Note that alternatives have a `name` attribute that is unique within each variability point to reference them from within a variability model. The description attribute of the alternatives has also been omitted for clarity from the example here and the formalization. It can be found in the complete schema in [Mie08].

Listing 3.8: Alternatives example XML

```

1      <!-- explicit alternative containing a bpel scope definition -->
3      <explicitAlternative name="alt1">
4          <bpel:scope xmlns:bpel=... />
5      </explicitAlternative>
6      <!-- expression alternative that concatenates the value of
7          another variability point with a string -->
8      <expressionAlternative name="alt2" language="...XPath..." ...>
9          concat(selectedValue("tns:epr"), "/app")
10     </expressionAlternative>
11     <!-- empty alternative -->
12     <emptyAlternative name="alt3" />
13     <!-- locator alternative -->
14     <locatorAlternative name="alt4"/>
15     <!-- free alternative with a restriction that prevents a
16         definition of a scope with the name startScope, the \$input
17         variable points to the input a user has made-->
18     <freeAlternative name="alt5">
19         <restriction language="...XPath...">
20             \$input//bpel:scope/@name != "startScope"
21         </restriction>
22     </freeAlternative>

```

Additional Properties of Alternatives. Besides the formalized properties above, an alternative has additional properties that are explained in Table 3.3.

Variability Point Dependencies. As shown above, a variability model includes a set of variability points. During the customization, a customizer sometimes has to make choices that depend on other choices that he made at another variability point [BFG⁺02, PBvdL05, JB04]. To model variability points that are dependent on other variability points, the concept of a *dependency* is introduced.

Definition 40 (Dependency). A dependency describes a directed link between

Property	Type	Explanation
Name	NCName	A name, uniquely referencing the alternative in the namespace of the variability model
Description	String	A short description that describes the alternative to be used to display to a user during customization
Default	Boolean	Indicates whether an alternative is the default alternative and is pre-selected during customization

Table 3.3: Additional properties of an alternative

two variability points denoting that the second variability point must be bound after the first one. The set

$$Dp \subseteq VP \times VP$$

defines the set of dependencies between variability points of a variability model including those that have been imported from other variability model.

Since variability point refinements always refer to a variability point, all dependencies must refer the original variability point and not the refinement. This ensures that all dependencies leading to and starting from the refined variability point, are kept and cannot be removed. Since it is allowed to introduce new variability points in variability models that refine other variability models, it is also allowed to introduce new dependencies. These dependencies can also target imported variability points. In particular, it is also allowed to introduce new dependencies between variability points imported from different variability models. It is, however, not allowed to remove any dependencies imported from other variability models. Removing dependencies would alter the possible sequence in which variability points must be bound and could thus implicitly introduce new alternatives. This is the case as enabling conditions could evaluate to true that would not evaluate to true in the refined variability model. In addition, removing dependencies may cause problems in evaluating enabling conditions as required values such as the selected value of another

variability point might not be available.

Given the requirements above, dependencies must always connect variability points and not variability point refinements. Thus, when adding dependencies to imported variability points they must be added to the variability points and not the refinements.

Definition 41 (Variability Graph). *A variability graph*

$$VG = (VP, Dp)$$

is a directed acyclic graph of variability points (nodes) and dependencies (edges).

Serialization of Dependencies. Listing 3.9 shows the serialization of a dependency. Dependencies reference their source and target variability points via a QName. Thus, they can also target variability points that are declared in imported variability models.

Listing 3.9: Dependency example XML

```
1 <dependency>
2   <source>tns:VP1</source>
3   <target>tns:VP2</target>
4 </dependency>
```

Definition 42 (Independent Variability Points). *An independent variability point is a variability point in a variability model, that is not a target of any dependency declared in the variability model. The set of independent variability points $VP^{Independent}$ of a variability model is therefore defined as follows:*

$$VP^{Independent} = \{vp \in VP \mid \forall d \in Dp : \pi_2(d) \neq vp\}$$

There might be multiple independent variability points in one variability model.

Definition 43 (Variability points a variability point depends on). *Given a*

variability point $vp \in VP$, then the set

$$VP^{\leftarrow}(vp) = \{v \in VP \mid \exists d \in Dp : \pi_2(d) = vp \wedge \pi_1(d) = v\}$$

contains all variability points on which the variability point vp has a direct dependency.

Definition 44 (Transitive variability point dependencies). *Given a variability point $vp \in VP$, then the set*

$$VP^{*\leftarrow}(vp) = \{v \in VP \mid \exists d_1, d_2, \dots, d_{n-1}, d_n \in Dp : \pi_2(d_n) = vp \wedge \pi_1(d_1) = v \wedge \pi_2(d_{k-1}) = \pi_1(d_k) \text{ for } 2 \leq k \leq n\}$$

contains all variability points on which the variability point vp depends on transitively.

Enabling Conditions. A dependency between two variability points denotes that one variability point must be bound after another one. Simply linking variability points using dependencies is not always sufficient. Sometimes more elaborate dependencies need to be described than the temporal one imposed by the dependency introduced above. One example is the situation where variability point B contains several alternatives B_1, B_2, B_3 and B is dependent on A that itself contains a set of alternatives A_1, A_2 . To express conditions such as “in case alternative A_1 is chosen in A only alternative B_2 can be chosen in B ” a more sophisticated dependency mechanism is needed. Therefore, *enabling conditions* are introduced that specify which alternatives are enabled at a variability point given the evaluation result of a condition.

Definition 45 (Set of Enabling Conditions). *The set $EC = \{\dots, ec_i, \dots\}$ is the set of all enabling conditions in a variability model including those imported from other variability models.*

The function $ec : VP \cup VPR \rightarrow \rho(EC)$ assigns a potentially empty set of enabling conditions to a variability point or variability point refinement. Not assigning any enabling conditions to a variability point is a shortcut that has the following semantics: In case no enabling conditions are assigned to a variability point all

alternatives at that variability point are automatically enabled.

An enabling condition is defined as a condition and a set of enabled alternatives. The semantics of the enabling condition is the following: if the condition evaluates to true, the alternatives contained in the set of enabled alternatives are enabled for the variability point which means that a customer can select one of these alternatives to bind the variability point.

Definition 46 (Condition in an enabling condition). *The set $Co = \{\dots, co_i, \dots\}$ is the finite set of conditions declared in a variability model including those imported from other variability models. A condition co is defined as a boolean function over its input data. Input data can be any building block $b \in B$ or an alternative $a \in Alt$ that has been selected at a previous variability point. (For a concrete definition of the selected alternative of a variability point see Definition 76 in Section 4.2.2). Thus the input data for a condition is defined as:*

$$inputData : Co \rightarrow \rho(B \cup Alt)$$

Thus each condition co is defined as a boolean function in its input data.

$$co : inputData(co) \rightarrow \{\text{true}, \text{false}\}$$

The function $cond : EC \rightarrow Co \cup \{\perp\}$ assigns a condition or \perp to an enabling condition. Assigning \perp instead of a condition denotes that this is a catch all enabling condition. Catch all enabling conditions evaluate to true automatically and thus have the same meaning as else branches in if-elseif-else statements in traditional programming languages such as Java.

Definition 47 (Enabled alternatives of an enabling condition). *The function $enAlts : EC \rightarrow \rho(Alt)$ assigns a potentially empty set of enabled alternatives to an enabling condition. The semantics is that these alternatives can be chosen if the corresponding condition evaluates to true. In case no alternative is enabled for a variability point during the customization, the variability point is in state disabled which means that no action is taken for that variability point. In Section 4.2 the operational semantics for variability models, including the states, are*

described in detail.

Enabling conditions can be *human* or *automatic*. Human enabling conditions are those enabling conditions that contain one or more enabled alternatives. Automatic enabling conditions are those where only one alternative is enabled which does not require a decision to be taken. Thus automatic enabling conditions are those where only one alternative is enabled that cannot be a free alternative. Free alternatives are not allowed in automatic enabling conditions as they always require an input and thus cannot be automatically bound without further information.

Definition 48 (Automatic enabling conditions). *The set*

$$EC_{auto}(vp) = \{e \in ec(vp) \mid |enAlts(vp)| = 1 \\ \wedge enAlts(vp) \cap (Alt^{Free}) = \emptyset\}$$

defines the set of automatic enabling conditions of a variability point.

Enabling conditions are ordered. The semantics of the ordering are the same as in if-elseif-else statements in programming languages.

Definition 49 (Set of all orders of enabling conditions for a variability point). *There exists a strict total order over the set of enabling conditions $ec(vp)$ of a variability point. The set*

$$\Omega_{VP_{ec}} = \{\prec_{ec(v)}^v \mid v \in VP\}$$

contains all strict total orders over the enabling conditions of a variability point.

Definition 50 (Set of previous enabling conditions). *Given an enabling condition $c \in ec(vp)$ of a variability point $vp \in VP$ the previous enabling conditions $EC^-(vp)$ are all those that are before c in the ordering of enabling conditions.*

$$EC^-(vp, c) = \{e \in ec(vp) \mid e \prec_{ec(vp)}^{vp} c\}$$

The i -th enabling condition of a variability point vp is then defined as $ec_i(vp)$. For all i -th enabling conditions of variability points the following holds: There

exist $i - 1$ elements in the set of previous enabling conditions.

$$\forall vp \in VP, \forall ec_i(vp) \in EC : |EC^{\leftarrow}(vp, ec_i(vp))| = i - 1 \quad (i \in \mathbb{N})$$

Let $vp \in VP$ be a variability point then $first_{ec}(vp)$ is the first enabling condition in that variability point, i.e. there exists no other enabling condition that is before $first_{ec}(vp)$ in the ordering of the enabling conditions of variability point vp .

$$\forall vp \in VP : \neg \exists e_2 \in ec(vp) : e_2 <_{ec(vp)}^{vp} first_{ec}(vp)$$

Serialization of enabling conditions. Listing 3.10 shows the serialization of enabling conditions. Note that two functions are defined as part of the Cafe variability descriptors that can be used in any XPath expression in an application descriptor: `selectedAlternative(vp)` and `selectedValue(vp)`. These functions return the selected alternative and selected value of a variability point. The variability point is referenced by its QName. Thus they can be used to define conditions on variability points a variability point depends on. These functions can be used in enabling conditions or expression alternatives.

Listing 3.10: Enabling condition example XML

```

1 <enablingCondition name="ec1">
2   <condition language="...XPath...">
3     selectedAlternative("tns:VP1")="tns:Alt1"
4   </condition>
5   <enabledAlternative>
6     tns:Alt3
7   </enabledAlternative>
8   <enabledAlternative>
9     tns:Alt5
10  </enabledAlternative>
</enablingCondition>

```

3.4.5 Enabling conditions and new alternatives for refinement variability points

To refine variability points from imported variability models, alternatives and enabling conditions can be added to variability point refinements. These additions, however, must not (i) allow the selection of values under a condition where this is not possible in the original variability point and must not (ii) introduce new values that can be selected during the customization that cannot be selected in the original variability point under the same condition. These conditions are necessary as otherwise variability point refinements could be specified that would lead to invalid customizations of the refined variability model. For example, it would be possible to customize a variability point refinement with a value that is not at all or only under different conditions a valid value for the refined variability point.

Rule (i) and (ii) are ensured as follows: Enabling conditions for refinement variability points must always reference an enabling condition of the original variability point. The refining and the refined enabling condition are then connected via a logical and. This ensures that the refining enabling condition only evaluates to true if and only if the refined enabling condition evaluates to true. Alternatives declared in a refinement variability point must also be linked to an alternative in the original variability point that they refine. These new alternatives are only allowed to offer a subset of the values of the alternative they refine. The new enabling condition can only enable alternatives that offer a subset of the values the alternatives of the old enabling condition offered. For example, a free alternative could be refined with an explicit alternative offering one value that is in the set of allowed values for the free alternative.

Definition 51 (Alternative refinement). *The alternative refinement map*

$$AltR : Alt \rightarrow Alt$$

assigns an alternative from the refined variability point to an alternative of the refining variability point refinement. Such a map must be present for all variability points in all variability point refinements. Multiple alternatives can

refine one alternative.

$$\forall vr \in VPR \forall ar \in alternatives(vr) \exists a \in alternatives(VPrefine(vr)) : AltR(ar) = a$$

The values allowed for the alternative refinement must be a subset of those allowed for the alternative it refines.

$$\forall vr \in VPR \forall ar \in alternatives(vr) : possVal(vr) \subseteq possVal(ar)$$

Definition 52 (Alternatives associated to a variability point and its refinement). Given a variability point $vp \in VP$, the set $Alt_{ref}(vp)$ is the set of all alternatives associated to this variability point or any of its refinements. It is defined as:

$$Alt_{ref}(vp) = \{a \in Alt \mid a \in alternatives(vp) \vee (a \in alternatives(vpr) \wedge VPrefine(vpr) = vp)\}$$

Definition 53 (Enabling condition refinement). The enabling condition refinement map $ecR : EC \rightarrow EC$ assigns an enabling condition to another enabling condition that the source enabling condition refines. An enabling condition can be refined by multiple other enabling conditions. Such a map must be present for all enabling conditions in all variability point refinements.

$$\forall vr \in VPR \forall ecr \in ec(vr) \exists ec \in ec(VPrefine(vr)) : ecr(ecr) = ec$$

A refinement enabling condition can only enable a subset of the alternatives of the enabling condition it refines. It can also enable refinement alternatives that refine one of the alternatives that was originally enabled.

$$\forall vr \in VPR \forall ecr \in ec(vr) : enAlts(ecr) \subseteq enAlts(VPrefine(vr)) \cup \{a \in Alt \mid AltR(a) \in enAlts(VPrefine(vr))\}$$

Refinement alternatives and enabling condition serialization. Listing 3.11 shows a variability point refinement that refines a free alternative (alterna-

tive *Alt5* as shown in Listing 3.8)) and an enabling condition. The original alternative and enabling conditions are referenced via their QName.

Listing 3.11: Refinement enabling conditions and alternative example XML

```

<variabilityPointRefinement ... >
2  <explicitAlternative name="altR1" refining="tns:Alt5">
    ...
4  </explicitAlternative>
    <enablingCondition refining="tns:ec1" name="ec1r">
6      <condition language="...XPath...">
            selectedAlternative("tns:VP1")="tns:Alt1"
8      </condition>
            <enabledAlternative>
10             tns:Alt3
            </enabledAlternative>
12    </enablingCondition>
</variabilityPointRefinement>

```

Constraints on variability models.

Constraint 4 (Enabled alternatives in enabling conditions). *For each enabling condition assigned to a variability point only alternatives of that variability point may be in the set of enabled alternatives.*

$$\forall vp \in VP \forall ec \in ec(vp) : \\ enAlts(ec) \subseteq alternatives(vp)$$

Constraint 5 (Else conditions in enabling conditions). *A condition for an enabling condition can be defined as \perp indicating that this is the catch all enabling condition that is used if all other conditions evaluate to false. At most one enabling condition per variability point may have a catch all condition.*

$$\forall vp \in VP \forall ec_1, ec_2 \in ec(vp) : \\ cond(ec_1) = \perp \wedge cond(ec_2) = \perp \Rightarrow ec_1 = ec_2$$

Constraint 6 (Enabling conditions referencing selected alternatives). *In case a*

condition in an enabling condition of a variability point vp_1 contains an alternative in its input data set, i.e. it depends on the selection of a certain alternative at another variability point vp_2 , variability point vp_1 must be dependent on variability point vp_2 . Otherwise a situation might occur where the enabling condition cannot be evaluated because an alternative has not been selected at the other variability point. Thus the following condition must hold true for all pairs of variability points:

$$\forall vp_1, vp_2 \in VP \forall ec \in ec(vp_1) : \text{inputData}(\text{cond}(ec)) \cap \text{Alt}(vp_2) \neq \emptyset \\ \Rightarrow vp_2 \in VP^{* \leftarrow}(vp_1)$$

3.4.6 Variability Models and Component Dependencies

Variability models not only describe the variability of individual components but can also be used to describe implicit dependencies between these components.

Since the Cafe variability metamodel is concerned with allowing to describe complex variability dependencies across multiple components, dependencies can exist between variability points of variability models associated with different components. Variability points must be bound during different phases of the Cafe application development process as indicated by the phase associated to a variability point. As a result, different dependency patterns between components induced through the combination of their variability models can exist.

Figure 3.18 shows two basic patterns. The upper part of the figure shows dependencies through variability points that must be bound during the same phase. Application model T contains two provider supplied components: component A and component B . They both supply an abstract variability model (AVM) that contains one variability point (VP): VP A respective VP B . Both variability points must be bound during solution engineering as indicated by the circle marked with “S” in the upper right corner. The variability model (VM) of the application model T refines both variability points into variability point refinements (VPR) VPR A and VPR B respectively. It also introduces a dependency between VPR A and VPR B . This may be necessary, as the value

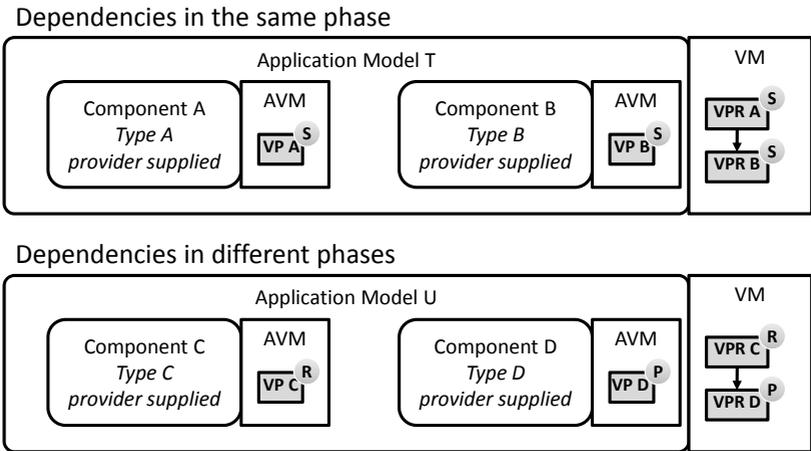


Figure 3.18: Dependencies between components induced by variability

for VPR B depends on the binding of VPR A. The semantics of this is that the component B can only be customized after component A has been customized.

The second part of Figure 3.18 shows a dependency induced by the variability model (VM) of application model U. It is different from the example in the upper part of Figure 3.18 as it connects two variability points that are in different phases. In this case the variability point of component C (VP C) must be bound during the running phase, whereas the variability point of component D (VP D) must be bound during the provisioning phase. The variability model of application model U refines both of these variability points into VPR C and VPR D respectively and introduces a dependency between those two. The semantics of this dependency as shown in Figure 3.18 now is that the variability point of component D can only be bound after that of component C has been bound. The variability point of component C is only bound in the running phase and that of component D must be bound before provisioning (in the pre-provisioning phase). Thus component D can only be provisioned after component C has already been provisioned and is already running. This dependency is thus important when determining the order for the provisioning

of components during the provisioning phase.

In the rest of this thesis, either the notation shown in 3.18 is used to show component dependencies induced by variability or the shortcut notation shown in Figure 3.19. In Figure 3.19, the two semi-circles attached to the components denote variability points and the arrow between the variability points denotes a dependency. The circle with the phase shortcut attached to the variability points denotes in which phase the variability point is bound. Thus Figure 3.19 shows that component C has a variability point that must be bound during the running phase and depends on the binding of the variability point of the component D that is bound during the pre-provisioning phase.

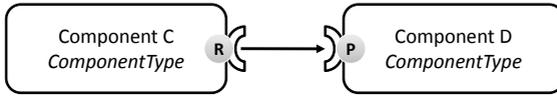


Figure 3.19: Shortcut notation for component dependencies induced by variability

Definition 54 (Component dependencies induced by variability models). A component $a \in C$ in an application template $t \in AM$ is said to be dependent on another component $b \in C$ in that application template t if at least one variability point of component a is dependent on a variability point of component b . The set of component dependencies $Cd \subseteq C \times C$ is thus defined as a subset of all component pairs that fulfill the following properties: A dependency exists between at least one variability point associated to one component and another variability point associated to the other component. A variability point can either be in a (imported abstract) variability model $avm(c)$ of a component c and thus be associated with that component, or it can have a locator pointing into a file of an implementation belonging to a component and thus be in the set of variability points $VP(c)$ affecting component c (cf. Definition 39). Thus the set of component dependencies is defined as follows:

$$\begin{aligned}
Cd = \quad & \{cd \in C \times C \mid \exists vpb \in VP \exists vpa \in VP^{*-}(vpb) \wedge \\
& (vpa \in avm(\pi_1(cd)) \cup VP(\pi_1(cd))) \\
& \wedge (vpb \in avm(\pi_2(cd)) \cup vpb \in VP(\pi_2(cd)))\}
\end{aligned}$$

Table 3.4 shows the dependencies that are allowed between variability points that belong to variability models of different components. The table must be read as follows. Variability Point B (VP B) is dependent on variability point A (VP A). The labels for the columns and rows correspond to the shortcuts of the phases as shown in Section 3.4.4. Thus **T** denotes the *template customization* phase, **S** the *solution engineering* phase, **B** the *component binding* phase, **P** the *pre-provisioning* and **R** the *running* phase. A “+” in a cell means that a dependency of the form VP B depending on VP A is allowed if VP B is in the phase indicated by the row, whereas A is in the phase indicated by the column of the cell. A “-” in a cell indicates that this combination is not allowed. Summarizing Table 3.4 variability points are always allowed to depend on variability points in one of the previous phases. In addition, variability points of the pre-provisioning phase are also allowed to depend on variability points of the running phase of other components. However, as indicated by the “+*” in Table 3.4 some restrictions apply for this case. To be able to define these restrictions first the sets of components that a component depends on in the pre-provisioning phase must be defined.

Definition 55 (Components that a component depends on in the pre-provisioning phase). *The set of components $C_{prov}^+(c)$ that a component $c \in C$ depends on in the pre-provisioning is thus defined as the set of components with at least one variability point vpa that must be bound in the running phase. Additionally, at least one variability point vpb from the variability model of c that must be bound during pre-provisioning must have a variability point dependency on vpa . The set of variability points from the variability model of a component c are either defined by the set of variability points $\pi_1(avm(type(c)))$ of the abstract variability model of the component type (Definition 20), or by the set of variability points $VP(c)$ associated to this variability point in the variability model of the*

application (Definition 39).

$$\begin{aligned}
C_{prov}^+(c) = & \{co \in C \mid \exists vpb \in VP \exists vpa \in VP^{*\leftarrow}(vpb) : \\
& vpa \in \pi_1(avm(type(co))) \cup VP(co) \\
& \wedge vpb \in \pi_1(avm(type(c))) \cup VP(c) \wedge bindT(vpa) = \text{running} \\
& \wedge bindT(vpb) = \text{pre-provisioning}\}
\end{aligned}$$

Given this definition of the $C_{prov}^+(c)$ set, the set of component dependencies during provisioning Cd_{prov} can be formally defined as follows:

$$Cd_{prov} = \{cd \in C \times C \mid \pi_2(cd) \in C_{prov}^+(\pi_1(cd))\}$$

Constraint 7 (Acyclicity of variability point dependencies between running and pre-provisioning phase). *Dependencies between variability points of the pre-provisioning phase of one component on variability points of the running phase of another component are only allowed if no (transitive) cycle is introduced between the dependencies of pre-provisioning variability points and running variability points of those components.*

$$\forall cd_1, cd_2 \in Cd_{prov} : \pi_1(cd_1) = \pi_2(cd_2) \Rightarrow \pi_2(cd_1) \neq \pi_1(cd_2)$$

Constraint 8 (Variability point dependencies between running and pre-provisioning phase and deployment relations). *Dependencies between variability points of the pre-provisioning phase of one (source) component on variability points of the running phase of another (target) component are only allowed if the source component is not in the set of transitively required components of the target component.*

$$\forall cd \in Cd_{prov} \forall d \in D : \pi_1(cd) = \pi_2(d) \Rightarrow \pi_2(cd) \notin C^+(d)$$

		VP A				
		T	S	B	P	R
VP B	T	+	-	-	-	-
	S	+	+	-	-	-
	B	+	+	+	-	-
	P	+	+	+	+	+*
	R	+	+	+	+*	+

Table 3.4: Allowed dependencies between variability points of different components

3.4.7 Modeling Requirements on Provider Supplied Components with Variability

The goal of modeling an application template with the Cafe application model is to produce a description of the application template that can be used by a provider to provision the application. So far components, their properties, their deployment relations and their variability is modeled. The model as presented above implicitly contains a set of requirements that the provider must fulfill. These requirements are called *implicit requirements* as they are not modeled explicitly. These implicit requirements can be derived from various aspects of the model:

- Component types of provider supplied components constitute a requirement for the provider. In case a component has an implementation type of provider supplied it is assumed that the provider can supply a component of that component type.
- Multi-tenancy patterns for provider supplied components indicate in which pattern a certain component must be supplied by the provider and thus constitute a requirement on the infrastructure.
- Deployment relations constitute a requirement, since they indicate that a certain component must be deployable on another component.

Despite these implicit requirements, the Cafe model allows modeling *explicit requirements* complementing the implicit requirements. These explicit requirements are used to select suitable *provisioning services* or *already provisioned*

components for provider supplied components once a template is customized and can be deployed. A formal definition for already provisioned components and provisioning services can be found in Section 3.5. For now it is sufficient to know that provisioning services can provision new components and already provisioned components are components that have been provisioned for the use in other applications for other customers.

Given the Cafe application metamodel, there are two general approaches to describe explicit requirements. The first solution would be to introduce a new *requirement* entity that would allow to model requirements explicitly. This is done, for example, in [UML09] and [KEK⁺09]. The second possibility is to treat the requirements on the infrastructure as variability points: Each component type defines an abstract variability model that defines the variability of components of that component type. These abstract variability models can contain variability points that must be bound during the template customization phase. In case a provider offers a component that is of a component type whose variability model contains such variability points, the provider must bind these variability points before offering the component. The component that is then offered expresses its capabilities through the already customized variability points. Thus these variability points are called *capability variability points* of a component from now on.

The second approach is taken in Cafe as it (i) allows to express complex dependencies between requirements and capabilities of one component and between different components without inventing yet another dependency mechanism, and (ii) is easy to integrate with the variability points already present in the variability model of an application template.

Thus, a template that contains a component of a specific component type in its application model also imports the capability variability points of that component type in its variability model. However, in the template these variability points are treated as variability points that must be bound during the component binding phase. They must be bound by the component that is selected for a provider supplied component. Thus the capability variability points of a component type serve as *requirement variability points* in the application template. To refine the requirements that are posed on provider supplied

components, requirement variability points can be refined just like any other variability point. Thus the requirement variability points and their refinements for a component type constitute the requirements suitable components must fulfill. How this is ensured is described in detail in Section 5.3.3.

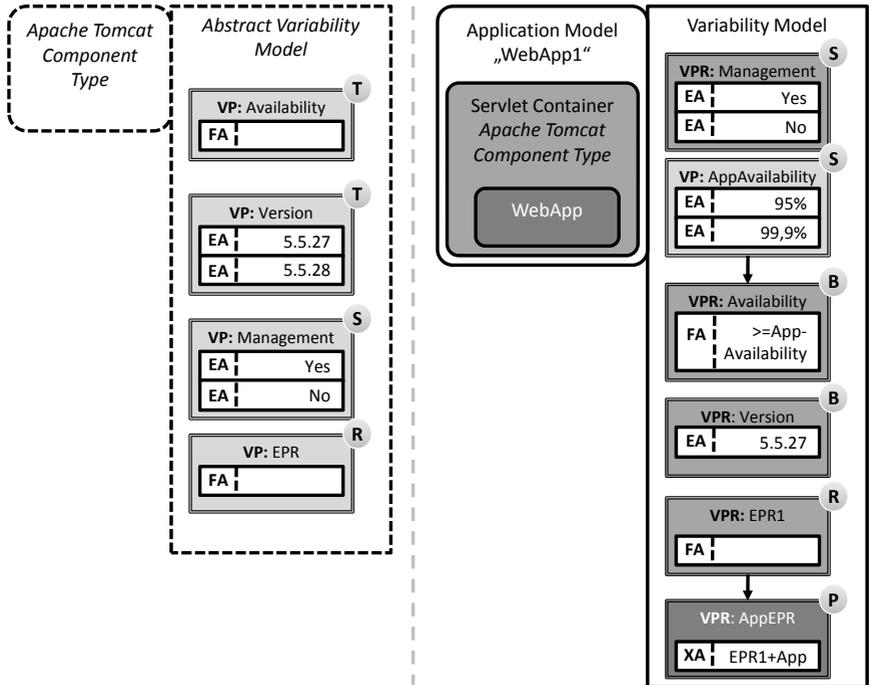


Figure 3.20: Modeling requirements on provider supplied components

The right part of Figure 3.20 shows an example application template *WebApp1* that contains a provider supplied component of component type *Apache Tomcat*. The left part of the figure shows the definition of the *Apache Tomcat* component type and its associated abstract variability model. This abstract variability model contains two variability points that must be customized during the template customization, i.e. before a provider can offer this template to customers. The *Availability* variability point and the *Version* variability point. Those must be set to the respective value (any value between 0 and 100% in

the availability case and either 5.5.27 or 5.5.28 in the version case). Then the component can be offered to customers.

In the variability model for the application model these two variability points *Availability* and *Version* are imported as variability points that must be bound during the component binding phase. These two variability points are also refined by corresponding variability point refinements (VPR *Availability* and VPR *Version*). The variability point refinement for the *Availability* variability point contains a condition that constrains the allowed values for the free alternative depending on the value of a variability point (the *AppAvailability* variability point) that is bound during the solution engineering phase of the application template. The allowed values are those that are higher than the value selected at the *AppAvailability* variability point. Thus if a customer has selected that he wants 95% availability for his application, the free alternative only accepts values greater or equal than those 95%. In case a provider now offers a component of the *Apache Tomcat* component type and has customized the value to a lower value than 95%, this is not a suitable component. The refinement of the *Version* variability point works similar. Only those components at a provider that have the value of 5.5.27 (i.e. are of version 5.5.27) are allowed.

The example thus shows how capabilities and requirements and their allowed values can be specified for a component type by variability points that must be bound during template customization. The example in Figure 3.20 also shows how requirements can be made more strict in application templates by refining these variability points that must be bound at binding time and that correspond to the capabilities of offered components.

Figure 3.21 shows a graphical shortcut to illustrate the requirements for the *Apache Tomcat* component type as shown in Figure 3.20. This shortcut is used in this thesis to illustrate the requirements explicitly without having to draw the variability points. In the shortcut it is always assumed that all variability points in the template customization and solution engineering phase have already been bound, so that the allowed values for the requirement variability points are already fixed. In addition, this shortcut notation also assumes that the requirements are independent, i.e. no dependency exists between the

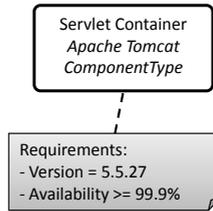


Figure 3.21: Shortcut notation for requirements on provider supplied components

requirement variability points. Thus this notation is simply used for brevity when requirements are illustrated in this thesis.

3.5 Providers and their Capabilities

Having described the Cafe application metamodel and the variability metamodel that can be used to describe provider-supplied components, the provider metamodel is now described that allows specifying the capabilities of a provider. Similar to the Cafe application metamodel that describes the *required* components to run an application, a provider can describe its available components as well as the components it is *capable* of provisioning. Thus the most important entities that a provider offers are:

- *Already provisioned deployments*. These are components that have already been provisioned, for example, to be used in other applications or for other customers.
- *Provisioning services*. These are services that can provision new components. Provisioning services are annotated with *provisioning capabilities* that describe what type of components a provisioning service can set up.

For example, the top part of Figure 3.22 shows the required components for the sample application described in Section 3.3. The bottom part of Figure 3.22 shows the capabilities of a provider. In this example the provider offers an already provisioned application server component and a SMS component in the

single instance pattern, as well as an already provisioned CRM component in the single configurable instance pattern. There is no already provisioned BPEL engine, but the provider offers a provisioning service (provisioning service A) that can provision an application server with a BPEL engine in the multiple instances pattern.

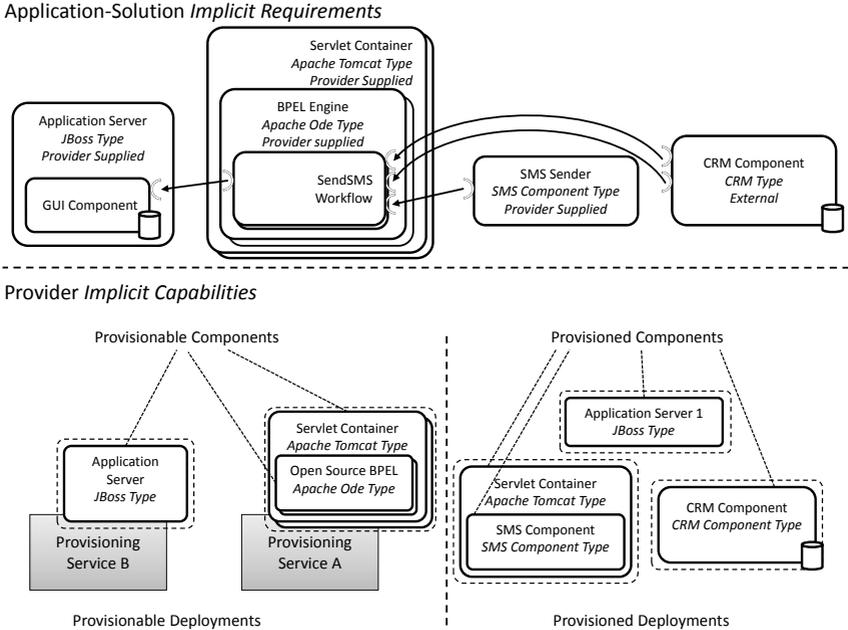


Figure 3.22: Application requirements and provider capabilities

3.5.1 A Metamodel for Providers

Having introduced a metamodel to specify the requirements an application poses on the infrastructure it should be provisioned in, a metamodel is now introduced that allows a provider to describe its capabilities. To be able to match requirements against capabilities and thus select the most suitable components for an application, the capability metamodel for providers makes use of several concepts of the application metamodel, such as the concept of

a component and its type as well as deployment relations and multi-tenancy patterns. Figure 3.23 shows the main elements of the provider metamodel. These elements are explained in detail below.

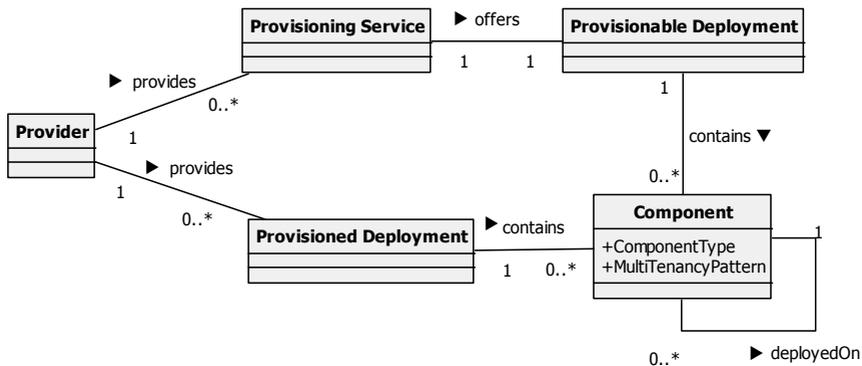


Figure 3.23: Entities of the provider metamodel

At first the concept of a *provider* is introduced to distinguish components that are offered by different providers.

Definition 56 (Providers). *The set $Prov = \{\dots, p_i, \dots\}$ defines the set of all providers.*

To describe the already provisioned environment of a provider, the concept of a *deployment* is introduced.

Definition 57 (Deployments). *The set $DP = \{\dots, dep_i, \dots\}$ contains the set of deployments at all providers. A deployment $dp \in DP$ is defined as a tuple*

$$dp = (C_{provisioned}, D_{provisioned}, type_{provisioned}, P_{provisioned})$$

where

- $C_{provisioned}$ is a set of components
- $D_{provisioned} \subseteq C_{provisioned} \times C_{provisioned}$ is a set of deployment relations
- $type_{provisioned} : C_{provisioned} \rightarrow CT$ is a map that assigns a component type to each already provisioned component

- $p_{\text{provisioned}} : C_{\text{provisioned}} \rightarrow P$ is a map that assigns a multi-tenancy pattern to each already provisioned component

The map $\text{provDp} : \text{Prov} \rightarrow \rho(DP)$ returns all deployments available at a provider.

Thus a deployment represents a set of provisioned components that are connected via deployment relations as well as their component types and multi-tenancy patterns. The set of deployments represent all components deployed at a provider.

Definition 58 (Provisioned components at a provider). *A provider $p \in \text{Prov}$ offers a set of provisioned components $C_{\text{provisioned}}(p) = \{c \in C \mid \exists d \in \text{provDp}(p) \wedge c \in \pi_1(d)\}$.*

Already provisioned components can be used to make an application available for a new customer. However, not all components required from a provider will always be already provisioned. Thus a provider must provision those components that are not available when an application should be provisioned for a new customer. To provision a new component, a provider must call a *provisioning service* [ML08b]. A provisioning service encapsulates the actions that must be performed to provision a new component or a set of components. The concept of provisioning services, their implementation possibilities and their interfaces are described in detail in Section 5.1. For now it is sufficient to know that a provisioning service advertises its *provisionable components*, i.e. the components it can provision.

Definition 59 (Provisioning Services). *The set $PS = \{\dots, pvs_i, \dots\}$ defines the set of provisioning services available at different providers. The map $ps : \text{Prov} \rightarrow \rho(PS)$ assigns a set of provisioning services to a provider. These are the provisioning services offered by this provider.*

The map $\text{provDep} : PS \rightarrow DP$ assigns a provisionable deployment to a provisioning service. A provisionable deployment represents the deployment that a provisioning service is capable of provisioning.

Note that each provisioning service has exactly one provisionable deployment. In case a provisioning engine can provision different deployments this is modeled

as several provisioning services. Thus a provisioning engine can offer multiple provisioning services.

For a given provider $p \in \text{Prov}$, the set

$$C_{\text{provisionable}}(p) = \{c \in C \mid \exists pvs \in ps(p) : c \in \pi_1(\text{provDep}(pvs))\}$$

contains all provisionable components that can be provisioned by a provider.

The map $\text{derivedFrom}_p : C_{\text{provisioned}}(p) \rightarrow C_{\text{provisionable}}(p)$ returns the provisionable component from which a provisioned component has been derived.

Capabilities of Already Provisioned Components and Provisionable Components. As described in Section 3.4.7, provider supplied components in an application template form implicit and explicit requirements on the provider. Thus already provisioned components and provisionable components must advertise their capabilities so that they can be matched against the requirements. Similar to implicit requirements, already provisioned and provisionable deployments fulfill a set of *implicit capabilities*. These implicit capabilities are the same as the implicit requirements, namely the component type, the multi-tenancy pattern and the deployment relationships of the already provisioned components or the provisionable deployments. The already bound variability points of a component constitute the *explicit capabilities* of each component in a deployment. These capabilities are formally defined in Section 3.4.7 where the binding of variability points is defined.

The implicit and explicit requirements of a provider supplied component in an application model can be used to find suitable already provisioned components or a provisioning service by evaluating their implicit and explicit capabilities. Performing component binding based on requirements and capabilities is done during the provisioning and thus described in detail in Section 5.3.3.

3.5.2 Component Binding

The set $CB = \{\dots, cb_I, \dots\}$ defines a set of *component bindings*.

To make an application model executable at a provider $p \in Prov$, each provider supplied component in an application model $a \in AM$ must be *bound* to an already provisioned component $cp \in C_{provisioned}(p)$ or a provisionable component $cps \in C_{provisionable}(p)$.

Definition 60 (Component binding). *Given the set of components $C_a = \pi_1(a)$ of an application model $a \in AM$, a component binding can be defined as a map:*

$$cb : C_a \rightarrow C_{provisioned}(p) \cup C_{provisionable}(p) \cup \{\perp\}$$

where \perp defines that no already provisioned component or provisionable component has been mapped to that component.

The semantics of a component binding specified by an application vendor or a customer is the following: In case a provisionable component is mapped to a component in the application model of a template, that component must always be provisioned by the associated provisioning service. Thus a component binding specified for an application template or during the template customization or solution engineering is a *static binding* to a provisioning service. In case the component can be reused because it is of one of the single instance multi-tenancy patterns, the semantics is that this component can also be realized by a component that has been created using the associated provisioning service. In case an already provisioned component is associated with a component this component must always be realized by the mapped component.

As said above, provider supplied components in an application model can be automatically bound to an already bound component or a provisioning service via a provisionable component during the automatic component binding in the provisioning phase (see Section 5.3.3). However, in some situations this automatic behavior is not desired. For example, if a specific already running component should be bound to a provider supplied component, or one specific provisioning service out of a set of suitable provisioning services should be statically bound to a provider supplied component. In these cases the component binding can be performed at any of the phases in the Cafe development life-cycle. For example, an application vendor could perform the

component binding for some provider supplied components of an application model during the packaging of an application template, or the provider could do it during the template customization by binding a corresponding variability point. Note that the corresponding variability model for these two cases is different.

Multiple different component bindings can exist for an application model that associate different already provisioned components or provisioning services to the provider supplied components.

A component binding is serialized as a *component binding descriptor*. Listing 3.12 shows the pseudo schema of such a component binding descriptor. It consists of a set of binding elements that contain an action attribute which states whether the provisioning engine must provision the associated component, subscribe to it or deploy on it. The applicationComponent element references the component in the application model of the associated template to which the bound component is associated. The EPR element references the EPR of the provisioning flow that represents the component (see Section 5.2.2). The targetComponentId is only necessary to identify components that are already provisioned, as the EPR of the component flow does not uniquely identify the already running instance of the component flow which represents the already provisioned component.

Listing 3.12: Component binding descriptor pseudo schema

```
<componentBindingDescriptor
2  targetNamespace="URI" name="NCName"
  xmlns="http://www.iaas.uni-stuttgart.de/cafe/schemas/
    componentBindings">
4  <binding action="provision|subscribe|deploy">*
    <applicationComponent>QNAME</applicationComponent>
6    <EPR>wsa:EndpointReference</EPR>
    <targetComponentId>String</targetComponentId>+
8  </binding>
</componentBindingDescriptor>
```

3.6 Car: Cafe Application Template Archive

One requirement for Cafe as stated in the research issues in Section 1.2 is a standardized package format. This package format is important, as it enables the exchange of Cafe application templates between application vendors and providers. As described above, the application format must contain a description of the components that make up the application template as well as the required infrastructure to run an instance of the application. Another important requirement for the package format is the possibility to express the variable parts of the application template. Detailed requirements for such an application format are described in the following.

Car files should be easy to create and use. Therefore, Car follows the simple approach taken in the Java world with `.jar`, `.war` and `.ear` packages [Sun08]. These packages are zip archives that contain a special folder (the `META-INF` folder) in which metadata is stored.

A Cafe application template archive file (Car file for short) therefore is a zip file with the file extension `.car`. Using a simple zip file has the advantage that tooling support and build automation tools (such as Maven[MO05]) exist that support zip files out of the box, thus developers do not require special tooling to build Car files. The Car file root contains a `META-INF` folder that contains the application descriptor as well as the variability descriptors and component binding descriptors. The other files that are used to implement the application can be placed in arbitrary folder structures under the root folder. Figure 3.24 shows a sample archive structure for the sample application template shown in Figure 3.5. The Cafe modeling tool introduced in the implementation chapter (Chapter 6) allows to model application models, variability models and component bindings in a graphical way and export them as Car files.

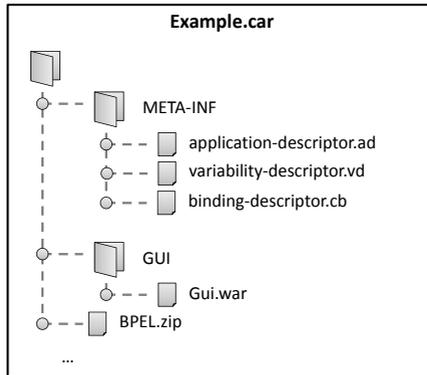


Figure 3.24: Example of a Cafe archive (Car file)

3.7 Detailed Template Engineering Process

As shown in Figure 3.3 the first step in the development process of a Cafe application is the *template engineering* phase. During this phase the application vendor develops the basic Cafe application template, including the variability model. This phase can be combined with any established software development methodology such as the Rational Unified Process [Kru99], the spiral model [Boe88], the waterfall model [Roy87] or more agile development methods such as extreme programming [Bec99] or Scrum [Sch95]. Evaluating particular software development methodologies for their suitability for template engineering for Cafe is out of the scope of this thesis. However, the application vendor must gather requirements from possible customers as well as from Cafe application providers to design meaningful templates that can be easily customized. Additionally, templates must also be highly flexible, so that the customer base becomes large enough enabling the Cafe provider to exploit economies of scale. In addition, the providers must be able to fulfill the implicit and explicit requirements contained in a template. Techniques from requirements engineering for software product line engineering [WL99, vdL02, PBvdL05] can be reused to capture the requirements from different customers. In particular, these techniques distinguish between requirements engineering for the

platform (which is done by the application vendor in Cafe) and requirements engineering for the application (which is done by the customer). The necessary variability requirements are hard to capture beforehand and may change in the course of the life-cycle of the Cafe applications and with the subscription of new customers. Therefore, a development methodology is needed, such as the spiral model or extreme programming that can deal with changes in variability requirements arising during the life-cycle of an Cafe application.

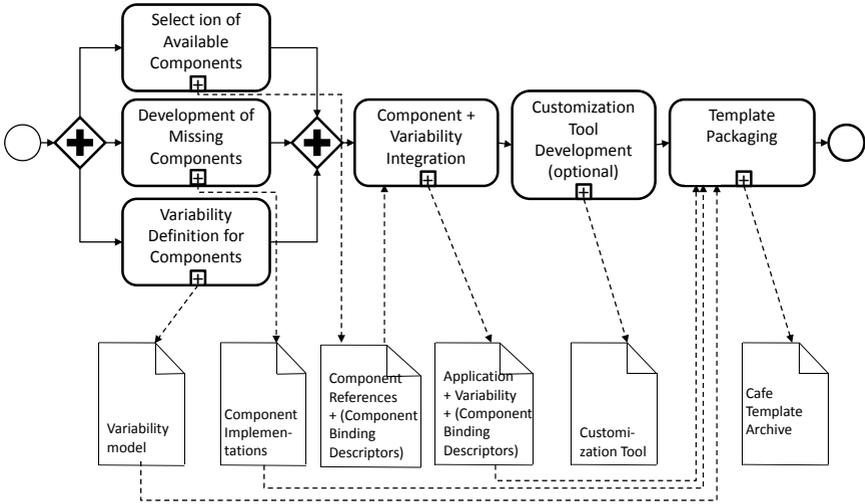


Figure 3.25: Steps and artifacts in template engineering

The template engineering phase corresponds to the *domain engineering* phase of software product line engineering [PBvdL05, MMLP09]. The application template that is the output of this phase of the development process is the basic application augmented with the variability. Figure 3.25 shows the detailed subprocesses of the template engineering phase. The subprocesses are described in detail below.

3.7.1 Selection of Available Components

Cafe applications can rely on component types that are offered by providers. In contrast to component-based engineering [BW96, HKSH04] the Cafe model thus does not assume that the components are included in the application package upon delivery of the application to a Cafe provider. Similar to the service-oriented architecture paradigm, applications can also contain components that run at third parties and are only referenced by the application. The output of this phase are references to components deployed at third parties (such as Web services) as well as code for components which require inclusion of the code in a new application (such as Java libraries). In the example in Figure 3.5, the SMS Component and CRM Components as well as the infrastructure components are such available components at a provider. Component types as defined in Definition 3 play an important role in the selection of available components. Since components in an application template that are of type *provider supplied* must specify the component type, components of this type must be available at the provider where the application template is later offered. In addition to that, provider supplied components can be bound to already provisioned components or provisioning services at the provider via component bindings (see Section 3.5.2). The output of this phase is thus optionally a set of component binding descriptors that describe the component bindings.

To ensure that an application template can be offered by a provider, the application vendor must make sure that only components of component types that are available at the provider have an implementation type of *provider supplied* in the application model of the template. To ensure the availability of component types, providers can advertise a list of available component types.

3.7.2 Development of Missing Components

Interleaved with the selection of available components is the phase in which components are developed that are internal and specific for one application template. Again, any established software engineering process can be used for these internal components. The output of this subprocess is the code for

the newly developed internal components. The *GUI* and *Workflow* components are such missing components in the example application shown in Figure 3.5. These components are specifically developed for the example application and their code is included in the resulting Cafe application package.

3.7.3 Variability Definition for Components

During this subprocess the variability for newly designed components is defined. This subprocess can be executed in parallel to the development of missing components, as components must be developed from scratch having variability in mind. In case a newly developed component is of a special component type the variability must refine the variability model of that component type. In case the newly developed component is an internal component, its variability can be either added to the variability model of the containing template, or the variability can be added to a new variability model that is then imported into the variability model of the template in the component and variability integration phase.

3.7.4 Component and Variability Integration

The component integration subprocess is the phase in which an instance of the Cafe application metamodel in form of a Cafe application-descriptor is created that describes the components selected and implemented during the previous phases. This subprocess requires an understanding how the different components work and which component is deployed on which other component. The result of this subprocess is the application descriptor that contains all components and their relationships.

In this phase also the cross-component variability of the Cafe application template is defined. The role that defines the variability for the Cafe application template is the *variability designer*. The variability designer specifies the overall variability of the application by combining the imported abstract variability models of provider supplied or external components with the newly designed variability for the internal components. The abstract variability models of provider supplied or external components can be refined according to the

rules specified in Section 3.4.4. In particular, variability points from different variability models can now be combined through dependencies, enabling conditions and expression alternatives so that component dependencies induced by the variability model are created. This is important, as the selected value at one variability point that belongs to one component may provide the value for another variability point of another component. In case an application template is of a specific component type, the resulting variability model of the application template must be a refinement of the abstract variability model of the component type. The rules for the refinement of abstract variability models are given in Section 3.4.4. Figure 3.15 shows a scenario where the variability model of an application template refines variability models of provider supplied components in that template and combine them with extended variability points defined in the abstract variability model of the component type of the application template.

Definition 61 (Application Template). *Given the definitions of a Cafe application model $am \in AM$ in Definition 1 and a Cafe variability model $vm \in VM$ in Definition 18 and a component binding $cb \in CB$ the set of application templates AT is defined as a subset of the cartesian product of the set of application models, the set of variability models and the powerset of component bindings:*

$$AT \subseteq AM \times VM \times \rho(CB)$$

Thus an application template is defined as an application model $am \in AM$, variability model $vm_{am} \in VM$, set of component bindings $CB_{am} \subseteq CB$ tuple (am, vm_{am}, CB_{am}) . Note that an application template $at \in AT$ is only valid if its variability model and component bindings fit to the application model.

3.7.5 Customization Tool Development

Having defined the application template and its variability, a tool must be developed that can be used by customers to configure the application prior to its deployment. In Chapter 4 algorithms are presented how such a tool can be automatically generated from the variability model at the provider.

However, in some cases the generic approach may not be suitable (for example if graphical modeling tools are needed to bind the variability). In these cases special-purpose customization tools can be built or the generated customization tools can be modified.

3.7.6 Template Packaging

Once all components and their variability have been defined, all code artifacts and descriptors are packaged into a Cafe application template archive (see Section 3.6). The output of the packaging step is thus the Car file that can then be uploaded to any Cafe provider.

3.8 Summary and Conclusion

Given the analysis of existing component models in Chapter 2, in this chapter an *application metamodel* for *Cafe applications* has been introduced. It was shown how this metamodel can be used to define *application models*. Besides the components that make up an application template, variability in the template has been introduced. Therefore, the *Cafe variability metamodel* is defined allowing the definition of variability as *variability points* that can dependent on each other. It was then shown how an application model and a variability model can be combined into a *application template* that describes the structure of an application as well as its variability. The Cafe archive (Car) package format was introduced as an implementation independent format to package Cafe application templates that can then be uploaded to a Cafe provider. The *template engineering process* was introduced as a part of the *Cafe development process*.

Once an application template is developed and packaged, it can be customized and offered by an Cafe application provider. Customers can then select one of the offered templates and further customize them. In the next chapter the customization of such application templates is described in detail, taking the rules and constraints formulated in the variability model of the template into account. Having customized the application template into a

customer-specific application solution, this solution can then be provisioned which is described in Chapter 5.

CUSTOMIZATION OF CAFE APPLICATION TEMPLATES

In this chapter the *customization* of Cafe application templates is described. Customization means the *binding* of variability by selecting a concrete alternative for each variability point in a variability model. As motivated in Section 3.4 variability in Cafe application templates can be bound during different phases of the Cafe development process.

The first phase in which variability must be bound is the template customization phase. In this phase the provider binds variability points that customize the offered template. The second phase in which customization is needed is the solution engineering phase. In this phase the customer binds customer-specific variability points. The resulting customized template which is called a *customer-specific solution* must then be provisioned for the customer. Still the solution contains several open variability points that must be bound during provisioning. Thus customization is also done during the provisioning phase by the provisioning infrastructure.

The ultimate goal of customization in Cafe is thus to produce a running

application that can be used by a customer. To ensure that an application can be executed, the customization must be *correct* and *complete*. Correct means that only enabled alternatives have been chosen and allowed values have been entered. Complete means that all variability points that must be bound have been bound. Only if a customization is correct and complete, it can be combined with the template into a running application. To ensure that a customization of a template is correct and complete, tooling supporting the customer during the customization must ensure that the binding of the variability model follows the operational semantics for the variability metamodel. The requirements that customization tools must fulfill are introduced in Section 4.1. The operational semantics for variability models is introduced in Section 4.2. It is shown how this operational semantics can be used to describe the actions that must and can be performed during a correct and complete customization.

In Section 4.3 the generation of customization workflows that follow the operational semantics of the variability model are introduced. It is then shown how these *customization flows*, together with standard workflow middleware can be used within customization tools that guide the customizers, such as providers, customers and the provisioning infrastructure through the binding of the variability of a template or solution while ensuring correct and complete customizations. The different purposes of the customization flows during the different phases is highlighted. The purpose of customization flows is not only the guidance of customizers but also the *validation* of already performed customizations. Thus, it is shown how customizations can be validated using customization flows.

Having described the operational semantics for variability models and the generation of customization flows from variability models, the implication of customization on the Cafe development process is described. In Section 4.4 the template customization phase is described and how customization flows can be used during this phase. In Section 4.5 the solution engineering phase is described in which a customer binds the variability of an application template.

4.1 Requirements for a Customization Tool

Cafe variability models contain variability points that can be bound at different phases during the set up of a Cafe application. Depending on these phases and the associated roles of a variability point, different persons or applications, such as the provisioning infrastructure, will perform the customization. In the following the different requirements for customization tools for Cafe application templates are described.

4.1.1 Deal with Abstraction from Implementation Details

The first requirement that a customization tool for Cafe application templates must fulfill is to deal with the abstraction from implementation details that is imposed by the Cafe variability metamodel. This requirement stems from the fact that customers do not want to deal with the implementation details of the template during solution engineering. Since components in the Cafe application templates can be implemented in any programming language, the customization tool should follow the abstraction from these programming languages.

In Cafe, humans are typically involved in the customization of a template in two phases, in the template customization phase and the solution engineering phase. In the template customization phase a provider must customize a template that is later offered to potential customers. A customization tool is needed that providers can use to customize the template without having to deal with the implementation details. In the solution engineering phase customers can customize templates into an application solution. Again, these customers must be presented with simple choices without them needing to know about the implementation details of the template they customize.

Thus a configuration tool for Cafe application templates should present the customers with a set of choices without having them to understand the implications on the code in detail. This is especially important, as Cafe application templates can be composed out of a multitude of components implemented in different programming languages. Often customers do not have experience

with these programming languages. Therefore, the customization tool must abstract from the implementation details of the individual components of the application template. Examples for these abstractions range from the simple specification of an application title that needs to be inserted in all HTML files that make up the UI, to the ticking of a check box specifying that the application must be highly-available which triggers the provisioning on an application server cluster with failover capabilities.

During the template customization and solution engineering phases the human-driven binding of variability points is very important. During the provisioning, the automated binding of variability becomes more important. However, human binding is still needed in this phase too, e.g. by administrators. Even with automated binding it is important that the customizer (which is another program in this case) does not need to know implementation details but can choose between simple options.

4.1.2 Guarantee Complete and Correct Customizations

When customizing a Cafe application template it is very important that the customization is both *complete* and *correct*. Two general possibilities exist on how to ensure the completeness and correctness of a customization: Either completeness and correctness are checked after the customization has been done or the tool only allows a customer to perform complete and correct customizations.

Checking the completeness and especially correctness after a customer has customized the application has two fundamental disadvantages: Given the possible complexity of a variability model for an application template that includes complex variability dependencies as well as a large set of variability points the checking algorithm can be very complex and time consuming [SZG⁺08, MML08, LSW09]. Additionally, it is not very user-friendly to tell a customer after the time-consuming customization of a complex application, that an error has been made at one of the early variability points and that a complete re-customization is necessary.

Therefore, the approach taken by Cafe is to exploit the explicit variability

model for the generation of customization tooling that forces users to only perform complete and correct customizations. On a high-level this is done by only allowing a user to select those alternatives of a variability point that are enabled because of his prior decisions. This ensures correctness of the customization. Completeness is ensured as the customization tool knows which variability points have to be customized by the customer and validates if they have been customized before it allows the application to be deployed.

4.1.3 Customer Guidance

A third, very important requirement for a customization tool is the guidance of customers. As said before, variability models can be very complex with hundreds of variability points. The customer should therefore be guided through the customization in a way that the dependencies of variability points are followed. A customization tool must therefore ensure that the customer cannot customize a variability point that depends on another variability point unless that other variability point is already bound. This must be done regardless of whether the customer is a human being or another program. In the following sections an operational semantics is given to the variability metamodel. This operational semantics is used to describe how a customization tool can be generated from a variability model.

4.2 Executing Variability Models

As said above, variability models serve two purposes. The first purpose is the documentation of variability in a template. The second purpose is to use the variability model as a model to generate customization tools. Therefore, an operational semantics for the variability metamodel is needed. This operational semantics is introduced in this section. In general two possibilities exist to execute a variability model: The first one is to develop a variability model execution engine that can execute variability models. The second possibility is to map the variability model to an established programming or scripting language. In this thesis a mapping to established workflow concepts is introduced, since a

variability model can be seen as a description of the workflow a customer has to perform to customize an application. In such a workflow, the variability points correspond to *tasks* the customizer has to complete. These tasks are connected via dependencies that express the *control flow* between the tasks. A variability model and its variability descriptor serialization can be therefore seen as a domain specific language (DSL) [MS05] for the generation of customization flows. Using an established workflow language to execute the variability models has several advantages: First of all, workflow languages with an execution semantic such as WS-BPEL are accompanied by workflow engines that allow to execute such workflows. These engines offer a set of nonfunctional properties such as transactionality, distribution, scalability, robustness, authorization and persistence for the workflows that are executed on them. To take advantage of these engines, a mapping from variability models to WS-BPEL workflows is introduced in detail in [ML08a]. While [ML08a] introduces a mapping from variability descriptors to WS-BPEL, the focus of the next section is to describe an operational semantics for the variability metamodel and then provide a basic mapping to general workflow constructs. Thus the approach presented here is more general than the one we presented in [ML08a].

4.2.1 Different Purposes for Executable Variability Models

Given the recursive aggregation model of the Cafe application metamodel and the definition of the Cafe variability metamodel, executable variability models serve additional purposes than to just guide a customizer through a customization of a template. An executable variability model for an application template can be used as described above as a customization tool for the application template. In case an application template A is used as a component in another application template AG , the aggregating template AG refines the abstract variability model of the component type of template A . During customization of the template AG the abstract variability model of template A is also bound. As the concrete variability model of A may refine the abstract variability model of its component type, some variability points must still be bound. However, they can depend on variability points present in the abstract variability model.

Thus the executable concrete variability model of template A can be executed again without prompting the user for the variability points that have already been bound during the binding of the abstract variability model of template A . Instead of prompting the user the customization tool can check if the already bound values are allowed (given the enabling conditions and alternatives for that variability point). Thus the executable variability model can also be used as a validation tool. Given the recursive nature of Cafe application templates this is especially important if one provider integrates components of other providers.

4.2.2 Operational Semantics for the Cafe Variability Metamodel

This section introduces an operational semantics for the Cafe variability metamodel. In general, the set of variability points inside a variability model can be seen as tasks someone (a human or a machine) needs to complete in order to customize an application template. These tasks are ordered by the dependencies. As defined in Section 3.4, variability points and their dependencies form an acyclic graph. Variability points that do not have a (transitive) dependency relation between each other can be bound in parallel, therefore the resulting tasks can be active in parallel, too.

In the following it is important to note the special treatment of refinement variability points. These are not treated as special variability points but are only considered when the variability point they refine is treated. Thus when dealing with a variability point, it must be always checked whether corresponding refinement variability points exist and the information of the original variability point and the refinement must be combined to derive how the variability point as a whole must be treated.

Steps in a Customization. To distinguish between discrete points in time during a customization, the notion of a *step* in the customization is introduced.

Definition 62 (A step in a customization). *Steps are discrete points in time during a customization and are represented by the natural numbers \mathbb{N} . At each step $s \in \mathbb{N}$ each variability point has a well-defined state. After a variability point*

is bound, the corresponding step is finished and the step value is incremented by one. Step $s = 0$ marks the beginning of the customization. The customization requires the binding of all variability points. Therefore, the amount of steps to be taken in a customization equals the cardinality of the set of variability points that must be bound. Given a set of variability points $VP_s \subseteq VP$ out of the variability points of a variability model that must be bound the step $s = |VP_s|$ denotes the end of the customization where all variability points have been bound. Therefore, the set of steps S is defined as follows:

$$S = \{n \in \mathbb{N} \mid 0 \leq n \leq |VP_s|\}$$

Variability Point States. To be able to distinguish which variability point has already been bound at a given step in a customization, a state model for variability points is introduced.

Definition 63 (States of a variability point). *The set States is defined as the set of states a variability point can have:*

$$States = \{Inactive, Active, Bound, Disabled\}$$

Possible state transitions are depicted in Figure 4.1. In the following the individual state transitions are described in more detail:

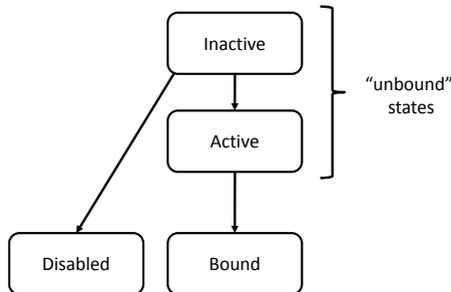


Figure 4.1: States of a variability point

Before the states are discussed in detail, some additional functions are

defined that make the following definitions easier.

Definition 64 (Variability point state at a given step). *The variability point state map*

$$state : VP \times S \rightarrow States$$

assigns a state at a given step $s \in S$ in the customization to a variability point.

Definition 65 (Set of variability points with a given state at a given step). *The set $VP_{st}(s) \subseteq VP$ is the set of variability points with a given state $st \in States$ at a given step s in the customization. It is formally defined as*

$$VP_{st}(s) = \{vp \in VP \mid state(vp, s) = st\}$$

Inactive variability points. Variability points in state inactive are variability points that cannot be bound by a customer because the necessary prerequisites to bind them have not been met, i.e. the variability points an inactive variability point depends on are not all in state bound or disabled.

Definition 66 (Inactive variability points). *At each step $s \in S$ in the customization at least one variability point in the set of variability points an inactive variability point depends on must be unbound (i.e. in state active or inactive).*

$$\forall vp \in VP_{inactive}(s) \exists vpd \in VP^-(vp) : \quad vpd \in VP_{inactive}(s) \vee \\ vpd \in VP_{active}(s)$$

Disabling variability points. A variability point vp must be disabled if it does only depend on variability points that are either disabled or bound and if none of the alternatives of vp is enabled. No alternative for vp is enabled if the first enabling condition that evaluates to true does not contain any alternatives that are enabled (see Definition 70). Another possibility is that no enabling condition evaluates to true and thus no alternative is enabled. As defined above, in case no enabling condition is specified for a variability point, always

all alternatives are enabled. Thus, as variability points must always have at least one alternative, in this case a variability point cannot be disabled.

At each step $s \in S$ of the customization, all variability points in state disabled must only depend on variability points in state bound or disabled and the first enabling condition that evaluates to true must not contain any alternatives that are enabled.

To be able to formalize the set of future disabled variability points at a given step in the customization, the evaluation of enabling conditions must be formalized first.

Definition 67 (Evaluation of conditions). *Since the conditions are not formally defined, a simple function is used that returns true or false for a given condition c at a given step s . How this is evaluated is left to the processor of the chosen condition language (i.e. the XPath processor if XPath is the expression language). Thus the function*

$$evaluate : Co \times S \rightarrow \{true, false\}$$

returns the result of a condition at step s . It is further defined as shown below:

$$evaluate(c, s) \mapsto \begin{cases} \text{true} & \text{if condition processor returns true} \\ \text{false} & \text{otherwise} \end{cases}$$

Definition 68 (Set of enabling conditions of a variability point that evaluate to true). $EC_{true}(vp, s)$ is defined as the set of enabling conditions of a variability point $vp \in VP$ that evaluate to true at step $s \in S$.

$$EC_{true}(vp, s) = \{ec \in ec(vp) \mid evaluate(cond(ec), s) = true\}$$

In case an enabling condition that evaluates to true has enabling condition refinements it must be checked if at least one of these enabling condition refinements evaluates to true to determine the set of enabled alternatives.

Definition 69 (Set of of enabling condition refinements of an enabling condition that evaluate to true). *The set of enabling condition refinements of an enabling condition $ec \in EC$ that evaluate to true at step $s \in S$ is defined as follows:*

$$EC_{true}^{ref}(ec, s) = \{ecr \in EC \mid ecr(ecr) = ec \wedge evaluate(cond(ecr), s) = true\}$$

Definition 70 (First enabling condition of a variability point that evaluates to true). $f_{true} : VP \times States \rightarrow EC$ returns the first enabling condition of a variability point $vp \in VP$ that evaluates to true at step $s \in S$. The following condition must hold true which states that for each variability point there exists no enabling condition that evaluates to true that is declared before the first enabling condition.

$$\forall vp \in VP : (\neg \exists ea \in EC_{true}(vp, s) : ea <_{ec(vp)}^{vp} f_{true}(vp, s))$$

Definition 71 (First enabling condition refinement for an enabling condition that evaluates to true). $f_{true}^{ref} : EC \times States \rightarrow EC$ returns the first enabling condition refinement of an enabling condition $ec \in EC$ that evaluates to true at step $s \in S$. The following condition must hold true: No other enabling condition refinement out of the set of enabling condition refinements exists, that evaluates to true and is declared before the first enabling condition refinement that evaluates to true.

$$\forall ec \in EC : (\neg \exists ecr \in EC_{true}^{ref}(ec, s) : ecr <_{ecr(ec)}^{ec} f_{true}^{ref}(ec, s))$$

The set of enabled alternatives for a variability point $vp \in VP$ at step $s \in S$ is then defined as follows: if the variability point is not active the set of enabled alternatives is empty. Otherwise the set of enabled alternatives is the set of enabled alternatives that is associated with the first enabling condition evaluating to true, or all alternatives if no enabling condition is assigned to the variability point at all. In case enabling condition refinements exist for the first enabling condition evaluating to true, the set of alternatives that is enabled are those of the first enabling condition refinement that evaluates to true associated with the first enabling condition that evaluates to true. In case no enabling condition refinement evaluates to true, the original enabling condition

still evaluates to true. Therefore, in this cases the enabled alternatives of the original enabling condition are enabled.

Definition 72 (Enabled alternatives of a variability point). *The function*

$$enAltS : (VP \cup VPR) \times S \rightarrow \rho(Alt)$$

is defined as follows:

$$enAltS(vp, s) \mapsto \begin{cases} \emptyset & vp \notin VP_{active}(s) \\ alternatives(vp) & ec(vp) = \emptyset \\ enAlts(f_{true}^{ref}(f_{true}(vp, s)), s) & EC_{true}^{ref}(f_{true}(vp, s), s) \neq \emptyset \\ enAlts(f_{true}(vp, s)) & otherwise \end{cases}$$

Definition 73 (Persistence of enabled alternatives). *Once a variability point becomes active and alternatives are enabled for that variability point, these alternatives remain enabled for all subsequent steps.*

$$enAltS(vp, t) = enAltS(vp, s) \Rightarrow t \geq s \wedge vp \in VP_{active}(s)$$

Having introduced these additional functions the set of future disabled variability points at a given step $s \in S$ of a customization can now be formalized as the set of variability points that are only dependent on variability points in state bound or disabled and have no enabled alternative at step s :

Definition 74 (Future disabled variability points). *The set*

$$VP_{futureDisabled}(s) = \{vp \in VP_{inactive}(s) \mid VP^{*-}(vp) \subseteq (VP_{bound}(s) \cup VP_{disabled}(s)) \wedge enAltS(vp, s) = \emptyset\}$$

is the set of variability points that will be disabled in the future and that can already be determined at step s .

Activating variability points. Active variability points are those variability points that can be bound customizer (customer or provider). I.e., a customizer

can select these variability points in a customization tool and select one of their alternatives. In case of a free alternative a value can also be entered.

Definition 75 (Potentially active variability points). *At each step $s \in S$ in the customization all variability points that are eligible for activation, i.e. that can transition from inactive to active must only depend on variability points in state bound or disabled and must have at least one enabled alternative.*

$$VP_{potActive}(s) = \{vp \in VP_{inactive}(s) \mid VP^{+-}(vp) \subseteq (VP_{bound}(s) \cup VP_{disabled}(s)) \wedge enAltS(vp, s) \neq \emptyset\}$$

Binding variability points. When binding a variability point the customer selects one alternative out of the set of enabled alternatives.

Definition 76 (Selected alternative of a variability point). *The function $selAlt : VP \rightarrow Alt \cup \{\perp\}$ returns the alternative that has been selected by a customer for a variability point or variability point refinement. In case no alternative has been selected \perp is returned. The selected alternative of a variability point must be from the set of enabled alternatives of that variability point which contains all refinement alternatives in case they are enabled. Thus, the following condition must hold true for all selected alternatives:*

$$\forall vp \in VP \cup VPR \forall s \in S : selAlt(vp) \in enAltS(vp, s) \vee selAlt(vp) = \perp$$

i.e. the selected alternative must be either in the set of enabled alternatives at this step or empty.

Definition 77 (Selected value at a variability point). *Once an alternative has been selected for a variability point vp the function $selVal : VP \rightarrow B \cup \{\perp\}$ returns the selected value of a variability point. The value that is returned depends on the*

variability type. The following definition assumes that $a = selAlt(vp) \neq \perp$

$$selVal(vp) \mapsto \begin{cases} explicitValue(a) & a \in Alt^{Explicit} \\ buildingBlocks(a) & a \in Alt^{Expression} \\ B_{I_a}(a) & a \in Alt^{Locator} \\ freeValue(a) & a \in Alt^{Free} \\ \perp & a \in Alt^{Empty} \end{cases}$$

Auto-binding of variability points. In case only one possible binding of an active variability point exists (i.e. only one alternative is enabled), the variability point can be bound automatically. Such variability points are called *automatic* variability points. Automatic variability points can be bound automatically by a customization tool when they become active. The automatic binding of variability points is called *auto-bind* from now on. Since free alternatives always require the specification of a value, variability points with an enabled free alternative do not qualify for auto-binding.

Definition 78 (Automatic variability points at a given step). *The set of automatic variability points at a given step is determined by the set of active variability points that only have one enabled alternative that is not a free alternative.*

$$VP^{Automatic}(s) = \{vp \in VP_{Active}(s) \mid |enAltS(vp, s)| = 1 \wedge (enAlts(vp, s) \cap Alt^{Free} = \emptyset)\}$$

Definition 79 (Persistence of selected alternatives). *Once a variability point transitions to state bound and an alternative is selected for this variability point, this alternative remains selected for all subsequent steps (unless the assignment is changed).*

$$selAlt(vp, t) = selAlt(vp, s) \Rightarrow t, s \in S \wedge t \geq s \\ \wedge vp \in VP_{bound}(s)$$

Definition 80 (Customization of a set of variability points). *Let VMC be a set of variability model customizations. Then a variability model customization $vmc(vm) \in VMC$ of a variability model $vm \in VM$ is a tuple consisting of a step $s \in S$ in which the customization of the variability model currently is, the variability point state map, the variability point selected alternative map and the variability point selected value map:*

$$vmc(vm) = (s, state, selAlt, selVal)$$

The function $cust : C \rightarrow VMC$ returns the customization of the variability model associated with a component, i.e. how the variability points of the variability model have been customized for that particular component.

Customization Document. A customization document is a serialized representation of a customization. Thus it contains the selected alternatives and their values for the variability points that are bound in a customization. The right part of Figure 4.2 shows the customization document representing a customization of a variability model depicted in the left part of the Figure.

A customization document contains an entry for each variability point and each variability point refinement. For a given variability point $vp \in VP$ that is not refined by any variability point refinement this entry is a triple consisting of the unique name of the variability point vp , the selected alternative at that variability point $selAlt(vp)$ and the selected or entered value $selVal(vp)$. Variability points A and B in Figure 4.2 are examples of such variability points.

In case a variability point has a variability point refinement (such as variability point C in Figure 4.2) it must be treated differently. Two cases exist: During the binding of the variability point a refinement alternative ar is selected. In this case the alternative a , refined by the refinement alternative ar (i.e. $a = AltR(ar)$), is saved as selected alternative of the variability point entry in the customization document. In addition, the refinement alternative ar is saved as the selected alternative of the variability point refinement entry in the customization document. In the case when a normal alternative (i.e. one that belongs to the refined variability point and not the refinement) is selected, then

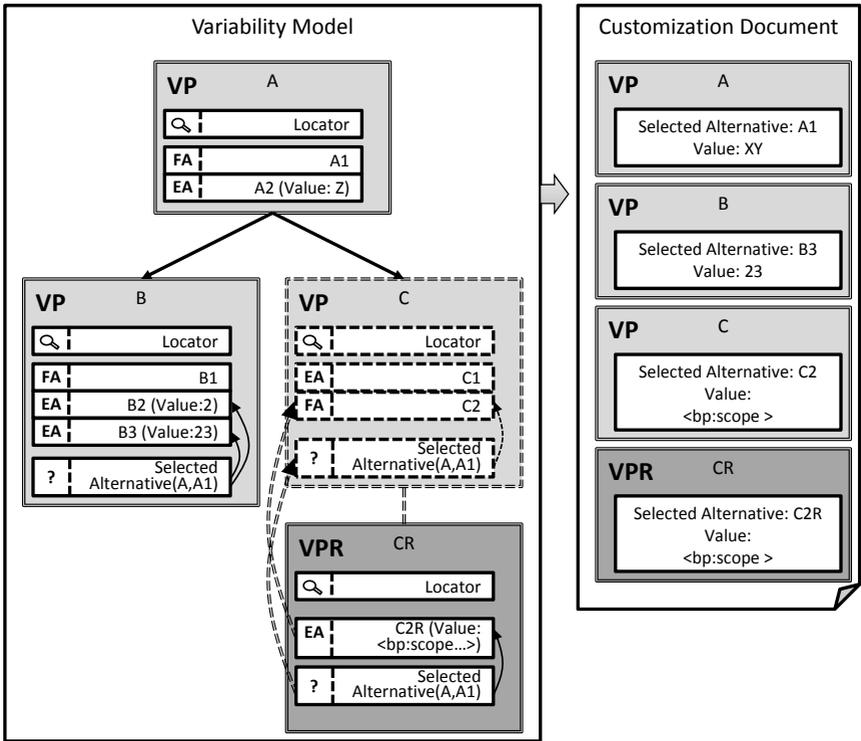


Figure 4.2: Customization document

this alternative is saved as selected alternative for both the variability point and the variability point refinement. In any case, the selected or entered value is saved as selected value of the variability point and the variability point refinement entry in the customization document. Thus the entry in the customization document for a variability point vp with a variability point refinement vpr is a three tuple of the unique name of the variability point vp , the alternative refined by the selected alternative of the variability point refinement $AltR(ar)$, and the selected value at the variability point $selVal(vp)$.

Similar to variability points, customizations for variability point refinements are saved in the customization document. For each variability point refinement

$vpr \in VPR$ the name of the refinement vpr , the selected alternative of the corresponding variability point $vp = VPrfine(vpr)$ (which can include alternative refinements) $selAlt(vpr)$ and the selected value of the corresponding variability point $selVal(vp)$ is saved along with the unique name of the variability point refinement.

Saving both, the variability point refinements and variability points in a customization document has the following reason: if the customization document is evaluated against the refined variability model, the variability point refinement values are used. In case it is evaluated against the abstract variability model, the variability point values are used since the abstract variability model does not know anything about the refinements.

Navigating through a variability model. Having described the different states of variability points and the transition between these states, the navigation algorithm that a customization tool must perform to guide a customer through the customization of a set of variability points VP_i out of a variability model, is shown in Algorithm 4.1 and is described below. The algorithm can either be used to guide a customer through the customization or to check whether a customization is correct and complete. The part 1 to part 4 comments in the pseudo code of the algorithm refer to the explanation below.

The algorithm starts at step 0 with a set of inactive variability points and runs in a loop until no variability point is in state inactive or active anymore, i.e. until all variability points have been bound or disabled.

Part 1: Disable eligible variability points. For each inactive variability point determine if it is eligible for disablement, if yes, disable this variability point by setting the state to disabled. Then increment the step count by one and continue and disable the next eligible variability point. In case the set of inactive variability points does not or no longer contain a variability point that is eligible for disablement continue at part 2.

Part 2: Determine the set of potentially active variability points. Determine for all inactive variability points whether they can transition to state active, i.e. whether they are in the set of potentially active variability points. Set the state active on all those variability points that can transition from state

Algorithm 4.1 Navigating through a variability model

```
1:  $s = 0$ 
2: while  $VP_i \cap (VP_{inactive} \cup VP_{active}) \neq \emptyset$  do
3:
4:   for all  $vp \in VP_{futureDisabled}(s)$  do
5:      $state(vp, s) = \text{disabled}$ 
6:      $s = s + 1$ 
7:   end for
8:
9:   for all  $vp \in VP_{potActive}(s)$  do
10:     $state(vp, s) = \text{active}$ 
11:   end for
12:
13:   for all  $vp \in VP_{automatic}(s)$  do
14:     for all  $a \in enAltS(vp, s)$  do
15:        $selAlt(vp, s) = a$ 
16:     end for
17:     for all  $v \in possVal(selAlt(vp, s))$  do
18:        $selVal(vp) = v$ 
19:     end for
20:      $state(vp, s) = \text{bound}$ 
21:      $s = s + 1$ 
22:   end for
23:
24:   display all  $vp \in VP_{active}(s)$  and  $vpr \in VPR$  where  $vp = VPrefine(vpr)$ 
25:   to the customizer
26:   user selects  $alt_s, val_s$  at variability point  $vp_s \in VP_{active}$ 
27:    $selAlt(vp_s, s) = alt_s$ 
28:    $selVal(vp_s) = val_s$ 
29:    $state(vp_s, s) = \text{bound}$ 
30:    $s = s + 1$ 
31: end while
```

inactive to active according to the rules above. Continue at part 3.

Part 3: Auto-bind eligible variability points. For each automatic variability point do the following: Set the only enabled alternative as the selected alternative of this variability point. Set the selected value as the value of the selected alternative. Set the state of the variability point to bound. Increment the step count by one then continue with the next automatic variability point. Once all automatic variability points have been bound, continue at part 4.

Part 4: Display a list of variability points in state active. Display all active variability points with the corresponding variability point refinements to the customizer who performs the customization. The customizer can then select one of the variability points he wants to bind. By selecting one alternative and a value in case of free alternatives, the customizer determines that the selected alternative of the variability point is set to the alternative selected and the selected value of the variability point is set to the value entered by the customizer. The selected alternative can also be a refinement alternative that is introduced by a variability point refinement. Increment the step count by one and continue at part 1 of the algorithm.

4.2.3 Complete and Correct Customizations

Having defined the operational semantics for a customization of a variability model, several properties of a customization can now be formally defined. As defined in Section 4.1.2, two properties are important for a customization. The customization must be *complete*, i.e. all required variability points must be bound or disabled. Furthermore, the customization must be *correct*, i.e. only alternatives can be selected that have been enabled.

Complete and Correct Customization. A complete customization of a set of variability points $VP_i \subseteq VP$ out of a variability model is formally defined as a customization of a variability model in which all those variability points have been bound or are disabled.

Definition 81 (Complete customization of a set of variability points). *For a customization $vmc(vm) = (s, state, selAlt, selVal) \in VMC$ of a variability*

model vm and a subset of its variability points $VP_i \subseteq \pi_1(vm)$ the function $cmCust : VMC \times \rho(VP) \rightarrow \{\text{true}, \text{false}\}$ is defined as follows:

$$cmCust(vmc(vm), VP_i) \mapsto \begin{cases} \text{true} & \forall vp \in VP_i : state(vp, s) \in \\ & \{bound, disabled\} \\ \text{false} & \text{otherwise} \end{cases}$$

A customization $vmc(vm)$ of a variability model vm is complete with regard to the variability points $VP_i \subseteq VP$ iff $cmCust(vmc(vm), VP_i) = \text{true}$

A correct customization of a set of variability points $VP_i \subseteq VP$ is formally defined as a customization of a variability model where the following holds true: All variability points are only bound to alternatives that are in the enabled alternative set for this variability point (i.e. all enabling conditions have been obeyed) and the selected value for an alternative is in the set of possible values for that alternative. If this is not the case, the variability point must be disabled. Otherwise the customization is not correct. However, this rule only applies for variability points that do not have a variability point refinement.

Special treatment of variability points with variability point refinements is needed. The special treatment is necessary because of the nature of abstract and concrete variability models in Cafe. An abstract variability model serves as a definition for all variability points and their possible alternatives for a component type. However, concrete components might refine the variability model and thus might restrict the allowed alternatives or the allowed values that might be entered at a free alternative under certain conditions. In Section 3.4.4 and especially in Section 3.4.5 strict rules for the refinement of variability points, their enabling conditions and alternatives are given. In order to enhance the definition of correct customizations, the two use-cases for variability refinements must be recaptured. In the first use-case a concrete variability model refines an abstract variability model. However, during the binding of the variability the concrete variability model is not known only the abstract model. Thus the variability point refinements, enabling condition refinements and alternative refinements of the concrete variability model are

not known when performing the binding of the variability. As a result, while the customization may be correct for the abstract variability model, it might not be correct for the refined concrete variability model. An example for this situation is later explained during the component binding in Section 5.3.3. In this case a customization of a variability model for the variability points of a component type that must be bound at template customization time is offered by already provisioned components. This customization of the variability model is then compared against the refined variability model for those variability points contained in a template to see whether the binding of the variability points in the already provisioned component is already a correct customization of the refined variability model of the template.

The second use-case for variability refinements is to refine an abstract variability model of a component type in an aggregating application template. In this case the refinements are only known in the variability model of the aggregating template and not the abstract variability model provided by the component provider. In such a situation, the customization of the variability model contains both, the selected alternative and the selected refinement alternative. However, the selected alternative can be used for validation and the variability point can be treated as one that has not been bound using a refinement.

The following is similar to the actions performed in Algorithm 4.1 with the following changes: To check the correctness of the customization against the refinement for each variability point, it must be checked if the variability point has alternative and enabling condition refinements for its variability point refinements $vpr = VPrefine(vp)$. If this holds true, it must be checked if the selected alternative a for a variability point vp in the customization of the variability model is an enabled alternative. If yes, the binding of that variability point is correct. If not, it must be checked if one or more refinement alternatives ra are enabled at the current step s that are a refinement of the selected alternative $a = selAlt(vp)$. Then it must be checked if any of these enabled refinement alternatives allows the value selected for the variability

point.

$$\exists ra \in enAltS(vp, s) : AltR(ra) = a \wedge selVal(vp) \in possVal(ra)$$

If this is the case, the binding of variability point vp is correct, otherwise it is not and the entire customization is not correct.

Definition 82 (Correct customization of a set of variability points). For a customization $vmc(vm) = (s, state, selAlt, vpValue) \in VMC$ of a variability model vm and a subset of its variability points $VP_i \subseteq \pi_1(vm)$, the function $crCust : VMC \times \rho(VP) \rightarrow \{\text{true}, \text{false}\}$ is defined as follows:

$$crCust(vmc(vm)VP_i) \mapsto \begin{cases} \text{true} & \forall vp \in VP_i : ((selAlt(vp, s) \in enAltS(vp, s)) \\ & \wedge selVal(vp) \in possVal(selAlt(vp, s))) \\ & \vee (\exists ra \in enAltS(s) : \\ & AltR(ra) = selAlt(vp, s) \\ & \wedge selVal(vp) \in possVal(ra))) \\ & \vee state(vp, s) \in \{\text{disabled}\} \\ \text{false} & \text{otherwise} \end{cases}$$

A customization $vmc(vm)$ of a variability model vm is called correct with regard to the variability points $VP_i \subseteq VP$ iff $crCust(vmc(vm), VP_i) = \text{true}$

Definition 83 (Complete and correct customization). Given a subset of variability points $VP_i \subseteq \pi_1(vm)$ of a variability model $vm \in VM$ and a customization $vmc(vm)$, the function $crmCust : VMC \times \rho(VP) \rightarrow \{\text{true}, \text{false}\}$ returns whether an instance of a variability model is a complete and correct customization of the set of variability points. Thus it is defined as:

$$crmCust(vmc(vm)VP_i) \mapsto \begin{cases} \text{true} & cmCust(vmc(vm)VP_i) = \text{true} \\ & \wedge crCust(vmc(vm)VP_i) = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$$

A customization $vmc(vm)$ of a variability model vm is called complete and correct with regard to the variability points $VP_i \subseteq VP$ iff $crmCust(vmc(vm), VP_i) = true$

4.3 Generating Customization Flows from Variability Models

As motivated in Section 4.2 it is now shown that variability models can be transformed into workflow models. It is shown that these models preserve the operational semantics of the variability metamodel and thus guarantee that only correct and complete customizations can be produced using the generated workflow models. In related work, Mendonca et al. [MCO07] formulated the idea of mapping feature trees of a software product line to process models modeled in BPMN [OMG09] for documentation purposes. The approach discussed below goes one step further than this idea and thus allows the direct generation of executable process models out of a variability model. Such a process model is then called a *customization flow*.

Customization flows also serve another purpose. Since they are generated from a variability model they can also be used to validate whether a customization document is a correct and complete customization of a set of variability points. The generation of customization flows is an important aspect of the development of Cafe applications since Cafe application vendors no longer need to develop customization tools and validation tools from scratch on top of their applications but can generate the complete customization and validation logic. In [ML08a] the generation of BPEL4People workflows from variability models is described in detail. In [Naa10] it is shown how variability models can be mapped to case handling concepts [VdAWG05] to allow additional flexibility. In this thesis, the focus lies on the mapping to general workflow constructs to show the general feasibility of the approach and to abstract from concrete implementation technologies. Therefore, variability models are mapped to process model graphs (PM-Graphs) as defined in [LA94, LR00]. In the following subsections it is shown how a variability model can be mapped to a PM-Graph thus transforming a variability model into a customization flow model. It is then shown how correctness and completeness of a customization

corresponding to an instance of the customization flow are ensured by the customization flow model and the executing engine.

4.3.1 Introduction to PM-Graphs

To describe the generation of a customization flow from the variability model of an application template, Process Model-Graphs (PM-Graphs for short) [LR00] are used as a formal model. Leymann and Roller introduce PM-Graphs as a formal model for workflows in [LR00]. The full definition is not repeated here in detail, but the main concepts of PM-Graphs that are necessary for the mapping from variability models to these PM-Graphs are introduced. The elements of a PM-Graph and related elements are listed in Definition 84 and described in more detail below where needed for the mapping from variability models to PM-graphs.

Definition 84 (PM-Graph). In [LR00] a process model graph (PM-Graph) is defined as a tuple

$$G = (V, \iota, O, N, \Psi, C, \Omega, \epsilon, E, \phi, \Delta, \vec{\Delta})$$

where:

- V is a finite set of data elements.
- $\iota : N \cup C \cup \{G\} \rightarrow \rho(V)$ is called an input container map.
- $o : N \cup \{G\} \rightarrow \rho(V)$ is called an output container map.
- N is a finite set of activities.
- $\Psi : N \rightarrow \mathcal{E}$ is called an activity implementation map.
- C is a finite set of conditions.
- $\Omega : N \rightarrow Q$ is the staff assignment map.
- $\epsilon : N \rightarrow C$ assigns each activity an exit condition.
- $E \subseteq N \times N \times C$ is a set of control connectors.
- $\phi : N \rightarrow \bigcup_{A \in N} \varphi_A$ assigns join conditions.
- $\Delta : N \times (N \cup C) \rightarrow \bigcup_{A \in N, B \in N \cup C} \rho(o(A) \times \iota(B))$ is the data connector map

- $\vec{\Delta} : N \rightarrow \bigcup_{A \in N} (\rho(\iota(P) \times \iota(A)) \cup \rho(o(A) \times o(P)))$ is the process data connector map.

4.3.2 Mapping Variability Models to PM-Graphs

As said above, a set of variability points in a variability model $vm \in VM$ are mapped to one process model, i.e. one PM-Graph $g \in PMG$, where PMG is the set of all PM-Graphs. As a result, the generation algorithm can be defined as a map $cfGen : \rho(VP) \rightarrow PMG$.

Definition of Activities. Each variability point $vp \in VP$ can be seen as a task where a human or the provisioning infrastructure needs to make a decision (namely select one of the enabled alternatives).

Therefore, for each variability point of the variability model a corresponding activity must be in the PM-Graph that represents the selection of one alternative by a human or a machine. These activities are called *variability activities* from now on. However, one variability point can have multiple enabling conditions. Depending on these enabling conditions, different alternatives can be enabled for that variability point. Essentially, this means that each variability point can be represented by different tasks that represent different allowed choices depending on which enabling condition evaluates to true first. Thus for each enabling condition $ec \in ec(vp)$ of a variability point $vp \in VP$ a variability activity $va_{ec} \in N$ is generated where N is the set of all activities of the customization flow model. In case no enabling condition is specified for a variability point this is treated as one enabling condition that enables all alternatives and has a condition that always evaluates to true. Figure 4.3 shows three generated variability activities for a variability point with three enabling conditions. Note that BPMN [OMG09] is used as a visualization of the generated PM-Graph as it is used as a process notation throughout the entire thesis. However, this means that additional gateways are introduced in the notation that do not appear in the PM-Graph itself. As shown in Figure 4.3 the variability activities for a variability point vp are preceded in the control flow by a *variability point start activity* $vs_{vp} \in N$ and succeeded in the control flow by a *variability point end activity* $ve_{vp} \in N$. The variability point

start and end activities are dummy activities that serve only synchronization purposes, therefore they do not have any input and output containers and an empty activity implementation. Thus the activity implementation map of the variability point start and end activities points to an empty activity implementation $\Psi(vs_{vp}) = \perp$ and $\Psi(ve_{vp}) = \perp$.

Using the variability point start and variability point end activities, the control connectors between variability points can be specified between the variability point start and end activities. This makes the resulting process model more understandable, as it prevents that a multitude of replicated control connectors need to be specified between the variability activities of different variability points. The control connectors shown in Figure 4.3 are specified below. Other constructs like input containers and output containers are described in detail later in this chapter.

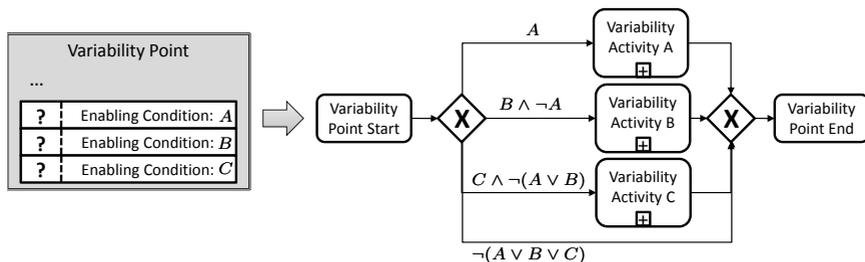


Figure 4.3: Mapping a variability point to activities and control connectors

Additional Activities for Variability Point Refinements. In case variability point refinements exist for a variability point, these variability point refinements must be considered when generating the activities and control connectors that represent the variability point. Variability point refinements can contain refinement enabling conditions that refine an existing condition further. Variability points with such refinements must be treated differently than normal variability points. The semantics of the refinement enabling conditions is that both, the refined and the refining enabling condition must evaluate to true (see Definition 53). Thus, after the variability point start activity, an *enabling condition*

refinement activity ecr_{ec} is inserted for each enabling condition $ec \in ec(vp)$ that has one or more enabling condition refinements, i.e. where

$$\exists ecr \in ec(vpr) : ec = ecR(ecr) \wedge VPrefine(vpr) = vp.$$

The enabling condition refinement activity ecr_{ec} is a dummy activity with an empty activity implementation $\Psi(ecr_{ec}) = \perp$.

This enabling condition refinement activity ecr_{ec} is connected to the variability point start activity via the control connector that would normally connect the variability point start activity with the variability activities of that enabling condition as shown in Figure 4.4. How these control connectors are specified is shown in detail below. Similar to normal enabling conditions, a variability activity $va_{ecr} \in N$ is created for each enabling condition refinement $ecr \in ec(vpr)$ which is an refinement of ec , i.e. $ecR(ecr) = ec$. This variability activity is then connected to the enabling condition refinement activity as shown in Figure 4.4. The variability activity of the original enabling condition is also connected to the enabling condition refinement activity.

The dark activities in Figure 4.4 are the additional activities needed for the enabling condition refinement of enabling condition A.

Definition of Control Connectors for Enabling Conditions. The variability activities $va_{ec_1} \dots va_{ec_k}$ representing all enabling conditions of one variability point, are connected via control connectors to the variability start activity vs_{vp} (see Figure 4.3), i.e. $\forall enc \in ec(vp) : \exists es \in E : es = (vs_{vp}, va_{enc}, c_{enc})$ where $c_{enc} \in C$ represents the transition condition which evaluates to true when the condition in the corresponding enabling condition evaluates to true. Each variability activity va_{ecr} of an enabling condition refinement ecr is connected to its corresponding enabling condition refinement activity ecr_{enc} where $enc = ecR(ecr)$, via a control connector $(ecr_{enc}, va_{ecr}, c_{ecr}) \in E$ where $c_{ecr} \in C$ represents the transition condition which evaluates to true if the condition declared in the enabling condition refinement ecr evaluates to true.

The variability activities of a variability point vp and its variability point refinement vpr , where $VPrefine(vpr) = vp$, are connected via a control con-

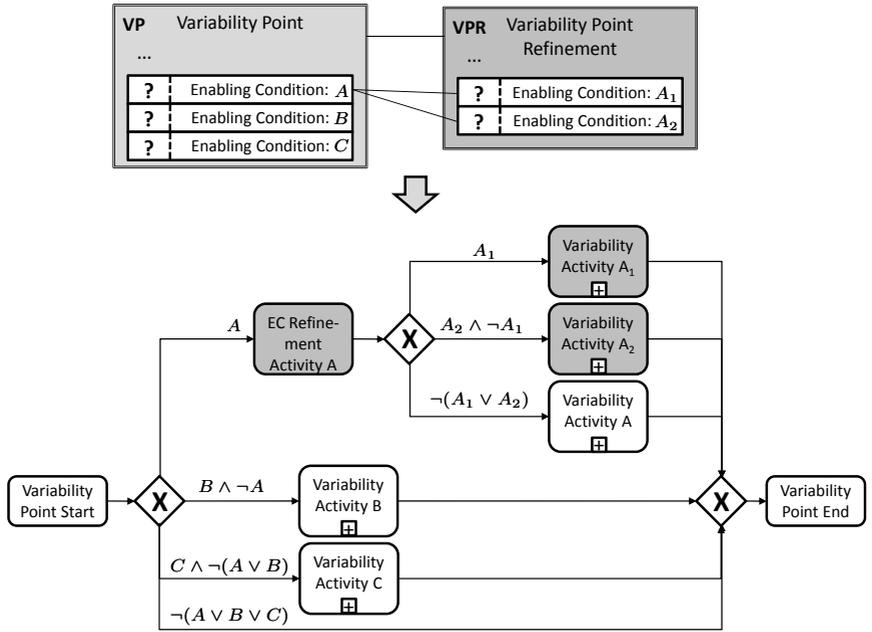


Figure 4.4: Additional activities and control connectors created for enabling condition refinements

vector to the variability point end activity, i.e. $\forall enc \in ec(vp) \cup ec(vpr) : \exists ee \in E : ee = (va_{enc}, ve_{vp}, true)$, where $true$ represents a transition condition that always evaluates to true. The join condition $jc = \phi(ve_{vp})$ of the variability point end activity ve_{vp} is defined as an *or-join*, i.e. $jc = c_1 \vee \dots \vee c_k$ where $c_1, \dots, c_k \in C$ represent the boolean conditions of the transition conditions of the control connectors leading to ve_{vp} .

Given this definition of the join condition, the join condition at a variability point end activity always evaluates to true if at least one of the transition conditions leading to the variability activities of a variability point evaluates to true. However, in case none of the transition conditions leading to the variability activities evaluates to true (i.e. if no enabling condition evaluates to true) the variability point end activity and subsequent activities would be

subject to dead path elimination of the workflow engine [LR00]. However, this is different from the semantics for the execution of variability models presented above. A variability point is called disabled in case no alternative becomes enabled because (i) the condition of an enabling condition that does not enable any alternative evaluates to true, or (ii) because no condition of any enabling condition enables to true. In this case subsequent variability points still need to be bound and should not be disabled. Therefore, to prevent dead-path elimination of these subsequent variability points, a control connector from the variability point start activity to the variability point end activity of a variability point is introduced that has a transition condition that always evaluates to true if all enabling conditions evaluate to false, i.e. $\forall vp \in VP : \exists ese \in E : ese = (vs_{vp}, ve_{vp}, dc)$; dc is defined as the negated conjunction of all enabling conditions:

$$dc = \neg \left(\bigvee_{i=1}^{|ec(vp)|} ec_i(vp) \right)$$

and $ec_i(vp)$ is defined as the i -th enabling condition of variability point vp (see Definition 49).

The same approach is taken for the transition condition of the control connector that connects an enabling condition refinement activity ecr_{ec} with the variability activity of the refined original enabling condition va_{ec} . Since this transition condition should only evaluate to true if all refinement enabling conditions evaluated to false, it is also annotated with the negated conjunction of all conditions in the refinement enabling conditions as shown in Figure 4.4 for the transition condition between the enabling condition refinement activity for enabling condition A and the variability activities for A .

Given the or-join semantics of the join condition of the variability point end activity, this activity will now always be activated, since one incoming transition condition will always evaluate to true.

As said above, each control connector pointing from the variability point start activity to the variability activities or enabling condition refinement activities

contains a transition condition c_{ec} that should evaluate to true under the same conditions as the condition in the respective enabling condition ec would evaluate to true. The same is true for the control connectors that connect enabling condition refinement activities with the respective variability activities as shown in Section 4.3.2.

Since transition conditions in a PM-Graph have a different semantics than traditional if-then-else statements and the semantics of the enabling conditions is that of traditional if-then-else statements, the enabling conditions cannot be simply copied to the transition condition. Multiple transition conditions on control connectors leaving a single activity can evaluate to true resulting in parallel execution of the activities connected via the control connectors. In the operational semantics of the variability metamodel only the first enabling condition that evaluates to true is considered, thus only one variability activity is allowed to be executed. To prevent that multiple variability activities are executed in parallel, the transition conditions leading to these activities must be defined in such a way that always only one evaluates to true. Note that in Figure 4.4 and Figure 4.3 an exclusive gateway is used in the BPMN notation that has exactly the semantics that only one transition condition evaluates to true. Nevertheless to keep the generation algorithm general enough for workflow models without such a semantics the transition conditions need to be treated as if such an exclusive gateway would not be available. In [ML08a] we use an `if` activity in BPEL to achieve the semantics of the exclusive gateway. Given the ordering of enabling conditions the exclusive gateway behavior can be ensured in the following way:

- For all enabling conditions, copy the condition to the transition condition and ensure that the transition condition does not evaluate to true in case one of the previous transition conditions evaluates to true. This can be ensured by adding a conjunction of the disjunctions of the negated enabling conditions of all previous transition conditions.

Thus the transition condition c_{ec} from the variability start activity to the variability activities of the j -th enabling condition $ec_j(vp)$ of a variability point

or variability point refinement vp is defined as follows:

$$c_{ec_j} = cond(ec_j) \wedge \neg(\bigvee_{i=1}^{i < j} cond(ec_i))$$

where $ec_i(vp)$ is defined as the i -th enabling condition of variability point vp (see Definition 49).

Variability Activities. Figure 4.5 shows the generated variability activity for a human enabling condition $ec \in ec(vp) \setminus EC_{auto}(vp)$ of a variability point $vp \in VP$ and for an auto enabling condition $eca \in EC_{auto}(vp)$. Note that both process models shown in the Figure are identical except for the *prompt user* and *auto-bind* activities. A variability activity is a structured activity that contains the following atomic activities. Note that PM-Graphs do not allow nesting of activities. However, in this case we can see each sub-process in the BPMN diagrams shown as a separate PM-Graph so that nesting is not required.

For each enabling condition an enabling condition start activity $start_{ec} \in N$ is generated. The enabling condition start activity is a dummy activity that is only needed for the readability of the process model. Thus it has empty input and output containers and an empty activity implementation: $\Psi(start_{ec}) = \perp$.

When the control flow reaches the variability activities of a human or auto enabling condition it is first checked whether the variability point has already been bound. This could, for example, be the case since the customization flow has been called with a customization document that already contains bound variability points because the component has already been customized as part of another application template. In case the variability point has already been bound it is checked in the *check selected alternative and value* activity whether the selected alternative is one of the allowed activities for the given enabling condition ec , i.e. whether $selAlt(vp) \in enAlts(ec)$. Additionally, if the selected alternative is a free alternative, it is checked whether the selected value is in the range allowed by the free alternative.

Thus a $check_{ec} \in N$ activity is generated for each enabling condition in a variability point. This $check_{ec}$ activity is connected to the enabling condition start

activity $start_{ec}$ via a control connector $esc \in E : esc = (start_{ec}, check_{ec}, selAlt(vp) \neq \perp)$. The transition condition of this control connector evaluates to true if for this variability point an alternative has already been selected.

If the alternative is allowed and if the value is allowed in case of free alternatives, the customization document is updated with the value of the selected alternative. In case the alternative is not part of the enabled alternatives and/or the value is not in the allowed range, a fault is thrown that stops the customization, the activities that throw the fault are not shown in Figure 4.5 for clarity reasons. Note that fault handling is not part of the PM-Graph metamodel, thus more precisely a fault situation should have been modeled by a separate variable `isCorrect` that is set to `false` in case a binding of a variability point is not correct. Since this is not central for the thesis as we do not execute PM-Graphs but BPEL in the prototype which supports fault-handling we omit the details here.

In case the variability point has not been bound before the control flow reaches the variability activities, either the user needs to be prompted or the variability point can be auto-bound. In case the enabling condition is not an automatic condition i.e. $ec \notin EC_{auto}(vp)$ the prompt user activity $prompt_{ec} \in N$ is generated which prompts a user to select one of the enabled activities. The prompt user activity is connected to the enabling condition start activity via a control connector $(start_{ec}, prompt_{ec}, selAlt(vp) = \perp) \in E$.

In case the enabling condition is an automatic condition, i.e. $ec \in EC_{auto}(vp)$ the auto-bind activity $auto_{ec} \in N$ is generated which selects the only active alternative. The auto-bind activity is connected to the enabling condition start activity via a control connector $(start_{ec}, auto_{ec}, selAlt(vp) = \perp) \in E$.

Afterwards the selection is added to the customization document. Details on how data is handled in variability activities are given below.

Control Connectors for Dependencies. Each dependency in the variability model is mapped to a control connector connecting the variability point end activity created for the variability point that is the source of the dependency with the variability point start activity created for the variability point that is the target of the dependency (see Figure 4.6). The little clouds shown in

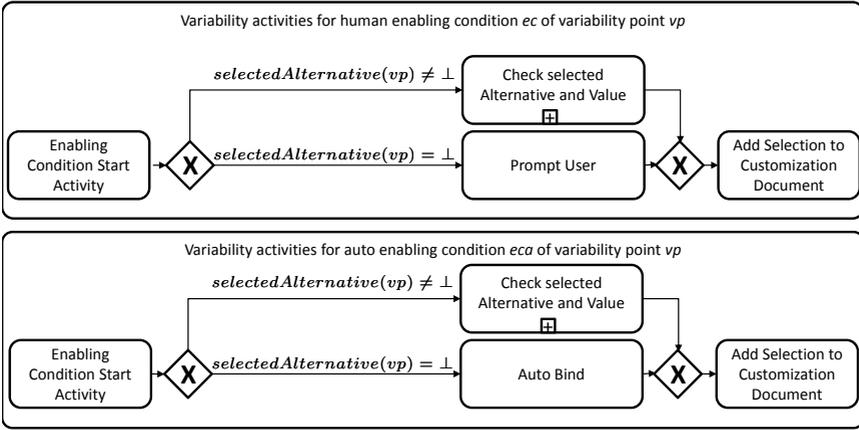


Figure 4.5: Variability activities for human and auto enabling conditions

Figure 4.6 represent the set of activities between the variability point start and end activities of one variability point, as those are not relevant for this figure. As shown in Figure 4.6, multiple control connectors leaving a variability point end activity are treated as parallel (and) splits, thus all subsequent variability point start activities become active at the same time. Similar multiple incoming control connectors at a variability point start activity are treated as and-joins, thus all previous variability point end activities must have completed successfully for a variability point activity to start. The transition condition on these control connectors always evaluates to true, as dependencies are always followed. Thus formally for all dependencies $(vp_1, vp_2) \in Dp$ in the variability model, one control connector $de \in E$ exists connecting the corresponding variability point end and start activities. This control connector is defined as: $de = (ve_{vp_1}, vs_{vp_2}, true)$.

Definition of the Data Elements. To define the input and output containers [LR00] for activities and transition conditions, the data elements used in the customization flow must be defined. At first, the data elements needed to evaluate whether a transition condition of a control connector leading to a variability activity (representing an enabling condition) evaluates to true must

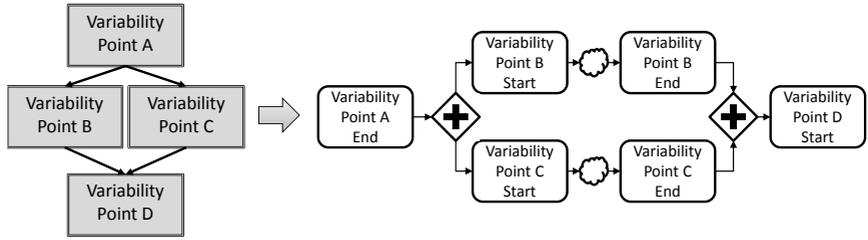


Figure 4.6: Mapping dependencies to control connectors

be defined. Since the input for these conditions are building blocks and/or the selected alternatives of previously bound variability points, the type of these data elements is an array of building blocks B or selected alternatives Alt . Given the definition of data elements $v \in V$ in PM-Graphs as pairs $\langle m; s \rangle$ where $m \in M$ is an element of the set of all names and $s \in S$ is an element of the set of all structures [LR00], the array of building blocks or alternatives can be defined as follows: First, the basic data structure is defined: $BoAType \in V = \langle M; B \cup Alt \rangle$ where building blocks and alternatives are treated as structures for simplicity reasons, i.e. $B \cup Alt \subseteq S$.

The array over the $BoAType$ is then defined as $BoATypeArray \in V$ where one array element again is either a building block or an alternative, i.e. $BoATypeArray(k) \in B \cup Alt$ and $k \in \mathbb{N}$.

According to [LR00] the transition condition $c_{ec} \in C$ for a control connector $e = (vs_{vp}, va_{ec}, c_{ec}) \in E$ connecting a variability start activity or an enabling condition refinement activity of a variability point vp to a variability activity or an enabling condition refinement activity representing an enabling condition ec is defined formally as a Boolean function over its input container $\iota(c_{ec})$. Given the set of transition conditions in control connectors pointing to variability activities $C_e \subseteq C$ the input container $\iota(C_e)$ of such a transition condition can be defined as:

$$\iota(C_e) \subseteq BoATypeArray$$

The second type of input containers to be specified are the input contain-

ers for variability activities. A variability activity receives a set of enabled alternatives of the corresponding enabling condition ec as an input. The data type $enabledAlts = \langle name, a \rangle$ is defined as a name ($name \in N$), alternative ($a \in enAlts(ec)$) pair. The data type of the input container for a variability activity is then defined as array over the $enabledAlts$ data type, i.e. $enabledAltsArray \in V$ where $enabledAltsArray \in enAlts(ec)$ Thus the input container $\iota(VA)$ of the set of variability activities $VA \subseteq N$ is defined as:

$$\iota(VA) \subseteq enabledAltsArray$$

The output of a variability activity is a single alternative that is the selected alternative for that variability point. The output container data type $selectedAlt$ is defined as a name alternative pair: $selectedAlt = \langle selected, a \rangle$ where $selected \in M$ and $a \in enAlts(ec)$. The output container $o(VA)$ of a variability is then defined as

$$o(VA) \subseteq selectedAlt$$

This selected alternative also contains the value that has been entered for that alternative.

Figure 4.7 shows the mapping of a variability point with a free alternative (*Free Alternative 1*) and an explicit alternative (*Free Alternative 2*), as well as two enabling conditions. *Enabling Condition A* enables only the explicit alternative whereas *Enabling Condition B* enables both alternatives. The activities representing *Enabling Condition A* contain an automatic variability activity that selects the only enabled explicit alternative of *Enabling Condition A*. Thus in case this enabling condition evaluates to true the *Explicit Alternative 2* is automatically taken as the selected alternative and its name and value are added to the customization document. In case *Enabling Condition B* evaluates to true, both alternatives are enabled and thus are added to the input data for the *prompt user* variability activity. The customer then selects one of the activities. In case the customer has selected the *Free Activity 1* he additionally has to enter a value. After the *prompt user* activity is completed the selected

alternative and its value is added to the customization document.

A check activity $check_{ec}$ for an enabling condition ec has the same input container as the corresponding variability activity va_{ec} . In addition, the entire variability document for the customization flow is part of the input for the check activity. Thus $i(check_{ec}) = \{i(va_{ec})\} \cup \{customizationDoc\}$ is the input container. The output container then contains either true or false. Thus $o(check_{ec}) = \{true, false\}$. The activity implementation of the check activity checks whether the alternative selected for the enabling condition ec of variability point vp is allowed, i.e. is part of the input container of the corresponding variability activity, i.e. it checks if $selAlt(vp) \in i(va_{ec})$. It also checks if the selected value is allowed at the selected alternative: $selVal(vp) \in allowedVal(selAlt(vp))$.

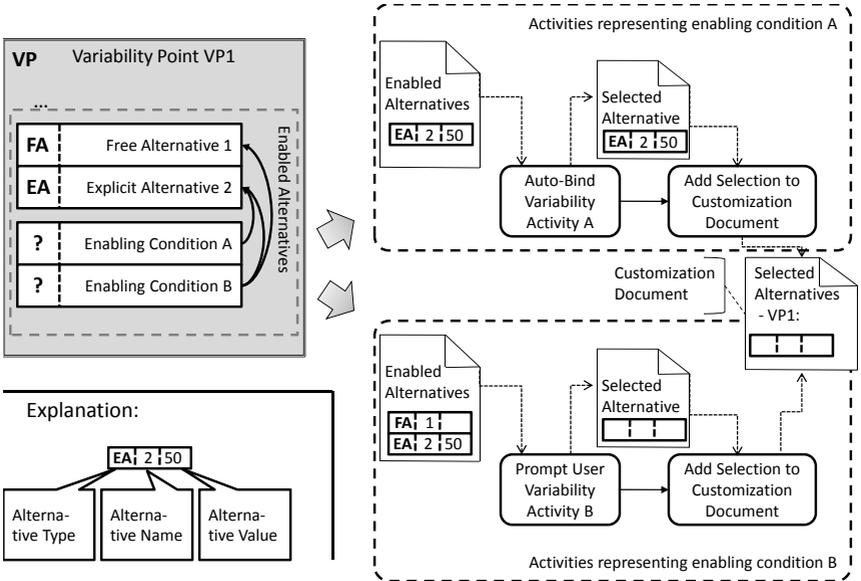


Figure 4.7: Mapping of alternatives to input and output data of variability activities

Definition of Staff Assignments. As said above, variability activities for hu-

man enabling conditions can be seen as tasks in a workflow. The variability metamodel allows to assign roles to variability points. These roles can then be mapped to roles in the workflow. According to [LR00] a staff query $q : \mathbb{N} \rightarrow \rho(\mathcal{A})$ from the set of all staff queries Q returns at a given point in time i a set of agents $q(i) \in \rho(\mathcal{A})$. We define the staff query for the variability activities for a variability point $vp \in VP$ as : $q_{vp}(i) = \{a \in \mathcal{A} \mid roles(vp) \in roles(a, i)\}$, i.e. the query for a variability activity returns those agents that have a role at time i that corresponds to the role assigned to the variability point. For example, a variability point with an assigned role of *technology specialist* can only be bound by agents with that role. The staff assignment that assigns a staff query to an activity is then defined as follows for all *prompt user* activities representing a human enabling condition $ec \in ec(vp) \setminus EC_{auto}(vp)$ for a given variability point vp :

$$\Omega(va_{ec}) = q_{vp}$$

Those activities representing an automatic enabling condition (auto-bind activities) $ec \in EC_{auto}(vp)$ for a given variability point or variability point refinement vp are associated with an activity implementation $e^{auto} \in \mathcal{E}$ of the set of activity implementations \mathcal{E} via the activity implementation map $\psi(va_{ec}) = e^{auto} \in \Psi$.

Given an input container of an auto-bind activity va_{ec} representing an automatic enabling condition $ec \in EC_{auto}(vp)$ is an array of alternatives with exactly one member $a \in enAlts(ec)$ $\iota(va_{ec}) = \langle \langle enabledAlternative; a \rangle \rangle$ and an output container $o(va_{ec}) = \langle \langle selectedAlternative; b \rangle \rangle$ the following holds true: $a = b$. This means that for an automatic activity the only alternative in the input container is always automatically selected and contained in the output container.

Returning the Customization Document. Once all variability points have been bound in a customization flow, the customization flow must return the customization document it produced to the starter of the customization flow. Therefore, a *return customization document* activity rta is added to the set of

activities N . This activity is connected to all variability point end activities of *end variability points*, i.e. variability points that do not have any outgoing dependency, via control connectors.

Definition 85 (End variability points). *The set of end variability points VP_{end} is defined as the set of variability points that do not have any outgoing dependency.*

$$VP_{end} = \{vp \in VP \mid \neg \exists d \in Dp : pi_1(d) = vp\}$$

Since the variability point end activities vpa_{vp_e} of each end variability point vp_e is connected to the return customization document rta activity there exists a control connector from each variability point end activity to the return customization document activity with an empty transition condition $(vpa_{vp_e}, va_{rta}, \perp) \in E^{rta} \subseteq E$.

The return customization document activity rta has an and-join condition, i.e. it only executes if all variability point end activities of end variability points have executed successfully and the set of transition conditions of all control connectors E^{rta} evaluate to true. Thus the join condition of the return customization document activity $\phi(rta)$ is defined as $\phi(rta) = \bigwedge_{i=0}^{|E^{rta}|} \pi_3(E_i^{rta})$ where E_i^{rta} denotes the i -th element in the set of control connectors leading to the return customization document activity.

Given the definitions above, the full algorithm to generate a PM-Graph from a subset of variability points $VP_g \subseteq VP_{vm}$ in a variability model vm is summarized in pseudo code in Algorithm 4.2.

4.3.3 Guarantee Complete and Correct Customizations

Given the generation of the customization flow as presented in the last subsection the following properties of all possible customizations that can be performed using this flow model can be proven:

- All possible customizations are complete, i.e. for all paths through the process model all variability points are touched.

Algorithm 4.2 Process model generation algorithm

```
1: for all  $vp \in VP_g$  do
2:   create VP start activity  $vs_{vp}$ 
3:   create VP end activity  $ve_{vp}$ 
4:   for all  $enc \in ec(vp)$  do
5:     if  $|ec(vp)| = 1 \wedge ec(vp) \cap Alt^{Free} = \emptyset$  then
6:        $\triangleright$  VP can be bound automatically for that enabling condition
7:       create variability activity  $va_{enc}$  with auto-binding
8:     else
9:       create variability activity  $va_{enc}$  with human involvement
10:      for all  $alt \in enAlts(enc)$  do
11:        add alternative to the input container
12:      end for
13:    end if
14:     $c_{enc} = \text{createTransitionCondition}(enc)$  (see Section 4.3.2)
15:    if  $\exists ecr \in ec(vpr) : enc = ecR(ecr) \wedge VPrefine(vpr) = vp$  then
16:       $\triangleright$  if an enabling condition refinement exists:
17:      insert EC refinement activity  $ecr_{enc}$ 
18:      create control connector  $e = (vs_{vp}, ecr_{ec}, c_{enc}) \in E$ 
19:      for all  $ecr \in ec(vpr)$  do
20:         $\triangleright$  create variability activity (human or auto)  $va_{ecr}$ 
21:         $c_{ecr} = \text{createTransitionCondition}(ecr)$  (see Section 4.3.2)
22:        create control connector  $e = (ecr_{ec}, va_{ecr}, c_{ecr}) \in E$ 
23:      end for
24:       $\triangleright$  create the control connector from the EC refinement activity
25:       $\triangleright$  to the variability activity of the original enabling condition
26:      create  $e = (ecr_{ec}, va_{ecr}, \text{createTransitionCondition}(ecr)) \in E$ 
27:      create control connector  $e \in E = (va_{ecr}, ve_{vp}, \text{true})$ 
28:    else
29:      create control connector  $e = (vs_{vp}, va_{enc}, c_{enc}) \in E$ 
30:    end if
31:    create control connector  $e = (va_{enc}, ve_{vp}, \text{true}) \in E$ 
32:  end for
33: end for
34: for all  $d \in Dp$  do
35:   create control connector  $e \in E = (ve_{\pi_1(d)}, ve_{\pi_2(d)}, \perp)$ 
36: end for
```

- All possible customizations are correct, i.e. for all paths through the process model only alternatives can be selected for variability points if

the corresponding enabling condition evaluated to true.

Sketch of proof: To prove the completeness of all possible customizations it must be proven at first that there is no path through the workflow where not all variability point start activities are touched. Then it must be proven in the first step that for all variability point start activities at least one alternative is bound or the variability point is disabled.

Touching of all variability point start activities. Given the generation algorithm a variability point start activity vs_{vp} and variability point end activity ve_{vp} is generated for each variability point vp . Thus, according to [LR00] if no control connectors were added, each variability point start activity would be ready for execution at the beginning of the workflow as activities not connected via control connectors are activated in parallel.

However, as control connectors are added between variability point end activities and variability point start activities of variability points where dependencies exist in the variability model these must be examined more closely. However, as the control connectors linking variability point end activities to variability point start activities have transition conditions that always evaluate to true, these control connectors always enable the execution of the variability point start activities and thus the binding of the corresponding variability point. Thus the only situation where a variability point start activity could be affected by dead path elimination would be if all variability point end activities it is connected to are in state *dead*. Therefore, it must now be proven that variability point end activities are never affected by dead path elimination and thus can never be in state *dead*. This is ensured, as at least one transition condition of a control connector from each variability point start activity to each variability point end activity evaluates to true by design and each variability point end activity has an “at-least-one” (or) join condition. Thus variability point end activities are never in state *dead* and subsequent variability point start activities are never in state *dead* so that each variability point is touched once.

□

The sketch of proof above only shows that all variability points are always either bound or disabled, therefore it ensures completeness. In the following sketch of proof, the correctness of the customization is shown which means that for each variability point only alternatives are selected that have been activated according to the enabling conditions:

Ensuring correctness of a customization in the customization flow. As said above a variability point can be seen as a task a customer must complete. Completing a task means that the customer must select one of the offered alternatives and in case of a free alternative must enter a value that is valid with regard to the restriction imposed on the free alternative. Therefore, it must now first be proven that only alternatives can be selected that are enabled given the respective enabling condition. This is ensured, as for each variability point for each enabling condition a separate variability activity exists that has an input container that only contains the enabled alternatives of that enabling condition. This variability activity only becomes active, if the corresponding transition condition evaluates to true. This transition condition only evaluates to true by design if the corresponding enabling condition would evaluate to true. Thus only alternatives can be selected that are enabled. In case no variability activity becomes active no alternative can be selected and the variability point is considered disabled which corresponds to the semantics of disabled variability points in the operational semantics of variability models.

The second aspect that must be proven is the aspect that for all free alternatives the entered value must correspond to the respective restriction. This is ensured by the respective tool that allows the entering of the value.

However, in case the customization tool cannot be trusted, an additional check could be added as an exit condition of each variability activity that checks whether the selected alternative is one of the enabled alternatives and in case of free alternatives whether the entered value is in the set of allowed values.

□

Given the two sketches of proofs above, a provider can be sure that after an instance of a customization flow completes without error, the customization

produced or validated by this instance of a customization flow is a correct and complete customization.

4.3.4 The Use of Customization Flows in Different Phases of the Cafe Development Process

Variability is bound during different phases of the Cafe development process. Given the five phases as described in Section 3.4.4, it is now shown by whom and how variability points are bound during the different phases. The different semantics of binding a variability point during one of these phases is highlighted too. It is also shown how customization flows are generated for the different phases.

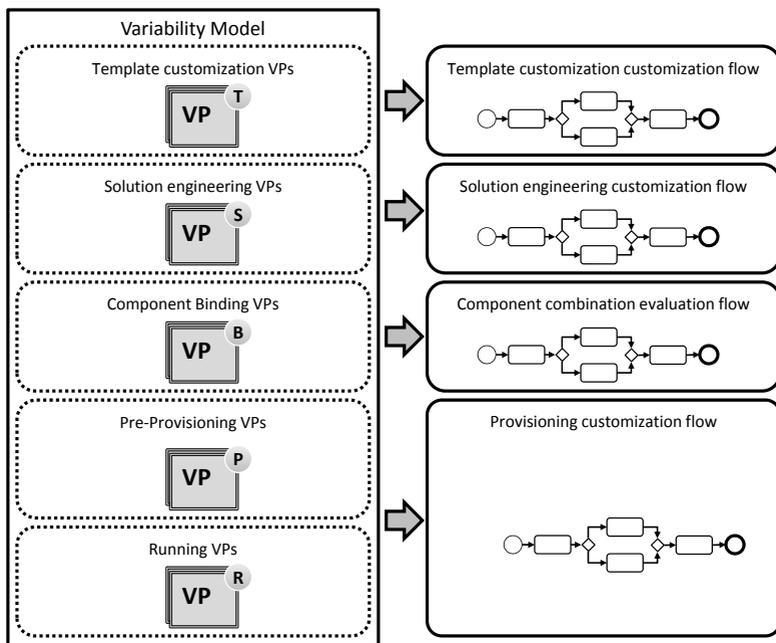


Figure 4.8: Variability model and generated customization flows

Figure 4.8 shows that multiple customization flows are generated for a variability model of an application model, relating to the different phases in

which the variability must be bound. The different customization flows and their usage are described in detail below:

Binding of Variability During the Template Customization Phase. Binding of variability during the template customization phase is done for all components in the template at once, i.e. the variability points for different components that must be bound during the template customization phase can have dependencies among each other.

The set of variability points that must be bound for a variability model $vm \in VM$ during the template customization phase is formally defined as:

$$VP_t(vm) = \{vp \in \pi_1(vm) \mid bindT(vp) = \text{template customization}\}$$

These variability points are from now on called the *template customization related variability points*.

Given the algorithm in Section 4.3.2, a customization flow for the provider-specific adaption of an application template can be generated. This customization flow is called the *template customization customization flow* from now on. The template customization customization flow is typically supplied with the application template by the application vendor. Providers can then use this customization flow to adapt the template to their needs before customers can customize the other variability points of the application template. As shown in Section 4.2.2, the output of a customization flow is a customization document. The customization document produced during template customization is thus called the *template customization customization document* or *tc-doc* for short.

In particular, providers can use the template customization customization flow to produce different versions of a template that distinguish themselves through their different tc-docs.

However, in case an application template is of a specific component type and this component type is used in another application template in a provider supplied component, the tc-doc of that template becomes important during the component binding. Component binding is explained in detail in Section 5.3.3. In short during component binding suitable already provisioned components

and provisioning services are searched for a provider supplied component in an application model of a template. Depending on the tc-doc of such an already provisioned component or provisionable component of a provisioning service, the already provisioned component or provisioning service may be suitable or not. It is suitable if the tc-doc of the already provisioned or provisionable component represents a correct and complete customization of the variability points of the variability model of the application model containing the provider supplied component that must be bound during the component binding phase.

Note that the customization document that is the result of the customization flow of one phase can be accessed by all customization flows for subsequent phases, for example, the solution engineering customization flow can access the customization document of the template customization flow. This is necessary as variability points in the later phases can have enabling conditions that depend on the selections in one of the earlier phases. The documents

Binding of Variability During the Solution Engineering Phase. Once a customer has selected a template $a \in AM$, the variability points of the associated variability model $vm = \pi_2(a)$ that must be bound during the solution engineering phase must be bound by that customer. The set

$$VP_s(vm) = \{vp \in \pi_1(vm) \mid bindT(vp) = \text{solution engineering}\}$$

formally defines the set of variability points that must be bound during the solution engineering phase. These are from now on called *solution engineering variability points*. All components in an application template are in the solution engineering phase at the same time, thus variability points affecting different components can have dependencies among each other if they both must be bound during the solution engineering phase.

Following the algorithm described in 4.3.2 a customization flow can be generated from the solution engineering variability points. This customization flow is from now on called the *solution engineering customization flow*. It serves multiple purposes: in case the application template is used as a standalone application template, i.e. it is not contained in any other application template,

the customization flow is used to prompt a human customer for customization choices for the solution engineering variability points. The output of this customization flow is a customization document that contains all customizations done during the solution engineering phase. This customization document is from now on called the *sc-doc* which is short for *solution engineering customization document*.

The second case where the customization flow is needed during the solution engineering phase is when an application template *A* is of a specific component type and this component type is used in another *aggregating* application template *AG*. In this case the variability model of the aggregating application template also imports the variability points of the aggregated variability model. Therefore, it is responsible to also prompt the user for the customization choices that affect the aggregated variability model. The provider of the application template *A* must then validate whether the customization document produced during the customization of the application template *AG* is a correct and complete customization so that *A* can actually be provisioned. This validation can be performed by the customization flow for *A*. Instead of prompting a customer for customization choices, the customization flow checks for each variability point whether the customization of this variability point is correct. This is done by evaluating the selected alternative and the selected value against the possible values and enabled alternatives at each variability point.

Binding of Variability During the Component Binding Phase. Variability points in the variability model of an application template that must be bound during the component binding phase correspond to requirements that must be matched by the capabilities of suitable components as described in Section 3.4.7. This matching is done by comparing the tc-docs of suitable components with the allowed values for the components in the application template. The set of variability points of a variability mode $vm \in VM$ that must be bound during the component binding phase is formally defined as:

$$VP_b(vm) = \{vp \in \pi_1(vm) \mid bindT(vp) = \text{component binding}\}$$

These variability points are called *component binding variability points* from now on.

The customization flow that deals with the customization of the component binding variability points is called *component binding evaluation flow* from now on. It serves the purpose of finding suitable component bindings having associated tc-docs that match the requirements expressed through the variability points that must be bound during the component binding phase.

Thus the component binding evaluation flow of a template is bound by the provisioning infrastructure that performs the component binding. The component binding customization flow is a separate workflow because it is used to validate if one or more component bindings contain suitable components that can be used to realize the provider supplied components in a template. Finding correct and complete component bindings and the component binding evaluation flow are described in detail in Section 5.3.2.

Binding of Variability During the Pre-Provisioning Phase. The variability points in a variability model $vm \in VM$ that must be bound during the pre-provisioning phase are formally defined as

$$VP_p(vm) = \{vp \in \pi_1(vm) \mid bindT(vp) = \text{pre-provisioning}\}$$

and are called *pre-provisioning* variability points from now on.

Pre-provisioning variability points are always specific for one component. Since components may have different states during the provisioning (i.e. one component is still in state pre-provisioning while another one is already in state running) one customization document is generated for each component that has pre-provisioning variability points. This customization document is called the *pre-provisioning customization document* (*pc-doc* for short) for a component. It is created in the aggregating template of a component and then sent to the respective component to configure it. Since the aggregating template might be supplied by a third party, the provider of the component must use a provisioning flow to check whether the selected alternatives and values are correct.

Binding of Variability During the Running Phase. The variability points in the variability model of a variability model $vm \in VM$ that must be bound during the running phase are formally defined as

$$VP_r(vm) = \{vp \in \pi_1(vm) \mid bindT(vp) = \text{running}\}$$

and are called *running* variability points from now on. Similar to the pre-provisioning variability points, one customization document must be created for each component that has associated running variability points. These are then sent to the respective components which must verify that the respective customization document is a correct and complete customization. This is again done with a customization flow on the component provider side.

The pre-provisioning and running variability points are represented by one customization flow, the *provisioning customization flow*. This customization flow deals with the customization of the individual components during the pre-provisioning and running phase. Since the components are treated separately during the pre-provisioning and running phase, i.e. one component can be already running while another one is not provisioned yet, additional activities, *start component customization* and *return component customization document* activities, must be inserted into the provisioning customization flow. These activities are responsible to start the customization of a component and to notify the starter of the customization of a component when all variability points of a component have been bound. Thus the return component customization document activities are similar to the return customization document activity as introduced in Section 4.3.2. The difference is that they send the customization document of a single component and not the complete customization document. For each group of variability points that belongs to one phase and one component, a start component customization and a return component customization document activity is generated. The variability points of one phase (the pre-provisioning or running phase) associated with one component are in a *component sphere* as shown in Figure 4.9. A component sphere $cs \in CS$ is a hyperedge connecting a set of variability points $VP_{c,p} \subseteq \pi_1(vm)$ of a variability model $vm \in VM$ belonging to a component c and a phase p . Thus a component

sphere $cs \in CS$ where CS is the set of all component spheres, is defined as:

$$cs = VP_{c,p} = \{vp \in \pi_1(avm(c) \cup VP(c)) \mid bindT(vp) = p\}$$

The component spheres form a *component sphere hypergraph* $CSHG = (VP, CS)$ consisting of the set of variability points and the hyperedges.

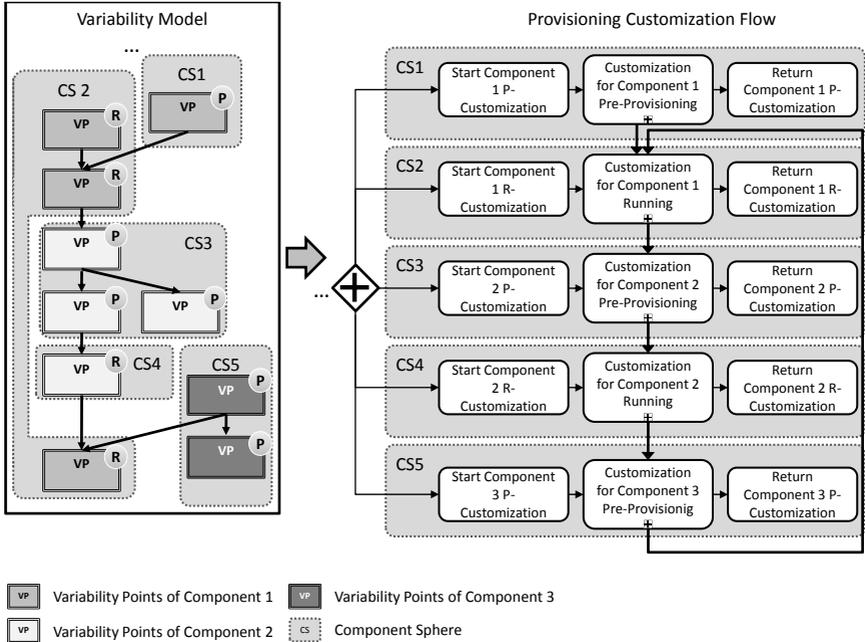


Figure 4.9: Component spheres and their influence on the generation of the provisioning customization flow

For each hyperedge for a given component c and phase p a start component customization activity scc_p^c and a return component customization activity rcc_p^c are added to the provisioning customization flow. These are responsible of receiving requests for customization and returning customization documents. The activities for the customization of the variability points of the component sphere are placed in between the start component customization and return component customization activities, depicted by the *Customization for subpro-*

cesses in Figure 4.9. All start component customization activities are activated in parallel at the beginning of the provisioning customization flow. Variability point dependencies between variability points of different component spheres are still present in the generated provisioning flow in the form of control connectors as described in Section 4.3.2. These are depicted in Figure 4.9 through the links between the customization for subprocesses.

4.4 Template Customization Phase

After having received an application template package from an application vendor, the provider can modify the application template. This is done during the template customization phase. The modified template, which is the output of the template customization phase, is then called a *provider-specific application template*. The main idea behind the template customization is to reduce the number of variability points and/or alternatives customers can or have to bind during the customization phase. Another motivation for template customization is the removal of alternatives that are not compliant with the provider’s infrastructure. For example, the SMS component in the example application (see Figure 3.5) could be affected by variability that would allow to substitute it with an automatic phone-call component that must be supplied by the provider. However, one provider does not want to allow customers to use such a component because he does not have a component of that type available and cannot provision components of that type. In this case this provider could bind a variability point that must be bound in the template customization phase in a way that the selection of the automatic phone-call component is no longer allowed during the template customization phase.

Thus the template customization phase is concerned with the binding of the subset of variability points $VP_t(vm) \subseteq \pi_1(vm)$ of a variability model $vm \in VM$ that must be bound during the template customization phase. Thus, the provider-specific application templates can be defined as follows:

Definition 86 (Provider-specific application template). *The set of provider-specific application templates is denoted by $PST = \{\dots, pst_i, \dots\}$.*

A provider-specific application template $pst \in PST$ can be defined as an application template $a \in AT$, customization $vmc \in VMC$, provider $p \in Prov$ tuple, i.e.

$$pst \subseteq AT \times VMC \times Prov$$

where the customization is a correct and complete customization of the template customization variability points of the variability model $vm = \pi_2(a)$ of the template a , i.e. $crmCust(vmc, vm, VP_t(vm)) = true$

Definition 87 (Template customization phase). The template customization phase can be defined as a function $tcp : AT \rightarrow PST$ from application templates to provider-specific application templates.

Given the definition of the provider-specific templates, the set of available provider-specific application templates $PST(p)$ at a provider $p \in Prov$ can be defined as $PST(p) = \{pst \in PST \mid \pi_3(pst) = p\}$

4.5 Solution Engineering Phase

In the solution engineering phase a customer adapts a provider-specific application template offered by a Cafe provider. As discussed in Section 2.1.4, in software product line engineering this phase is called application engineering and deals with the refinement of a platform into a concrete application. Similar to that notion this phase is called *solution engineering* in Cafe. In the Cafe solution engineering phase a provider-specific application template is transformed into a *customer-specific application solution* or *solution* in short. A solution is a provider-specific application template in which all variability points that can be bound during the solution engineering are bound. In particular, these are those variability points that need to be bound by the customer. Thus a solution is always customer-specific as it contains the customizations a particular customer has made. Variability points, such as concrete EPRs of components, that can only be bound during or after the provisioning are not bound in a solution.

The solution engineering phase has three main sub-phases. The first phase, the *template selection* phase deals with the selection of a template by a cus-

tomor from an application portal. Once a template has been selected, a new customization of the variability model of that template is created. In the second phase, the *template customization* phase the customer then *binds* the variability of the template. This is done by choosing between the offered alternatives for each variability point. In the third phase, the *solution creation* phase the customization document and the application template are automatically combined into a customer-specific application solution.

Figure 4.10 shows these three sub-phases and the template including the variability model. It also shows the customer-specific application solution including the customization of the variability model in which all variability points that can be bound during the template customization phase are bound.

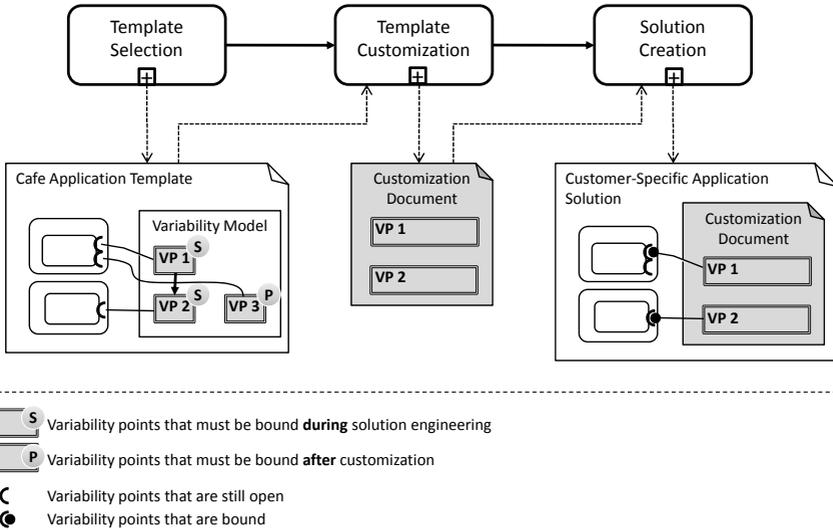


Figure 4.10: Solution engineering process and artifacts

In the *Template Selection Phase* the customer of a Cafe provider selects a provider-specific application template $pst \in PST(p)$ that he wants to customize from the application portal of provider p .

4.5.1 Customization Phase

When the customer has selected a provider-specific application template pst from which he wants to create a customer-specific application solution, a customer-specific customization of the variability model for that template is created. In the customization phase all solution-engineering variability points need to be bound. All other variability points are bound in one of the later phases.

The output of this phase is a customization $vmc \in VMC$ of all solution engineering variability points of the selected provider-specific template. Thus given a provider-specific template pst , the customization vmc performed during the solution engineering phase must be a correct and complete customization of the solution engineering variability points of the variability model $vm = \pi_2(\pi_1(pst))$ of the template from which the provider-specific template pst has been created $VP_s(vm)$. Thus the following must hold true: $crmCust(vmc, vm, VP_s(vm)) = true$

In the Cafe prototype the customization created during the solution engineering is serialized as a customization document as shown in Figure 4.2.

4.5.2 Solution Creation Phase

The provider-specific application template and the customization document containing the variability point bindings for the template customization and solution engineering phase constitute the ingredients to create the solution that is specific for one customer of a given application. In the solution creation phase the customization document and the template are combined into a customer-specific application solution. To do this, the values selected during the solution engineering and documented in the customization document are inserted at the right places at the right artifacts in the template as defined by the variability model through the locators. The same has to be done for the values selected during template customization that are documented in the customization document of the provider-specific application template. To do this, a special service is required. This service is called *solution creation service*. It first creates a customer-specific copy of the template. Then it interprets the

customization document by running through the list of bound variability points of the two customization documents that contain the bound variability points of the template customization and solution engineering phase. The variability points are treated in the order they have been bound in case variability points exist that overwrite each other because they have locators pointing to the same building block. For each variability point the solution creation service must retrieve the locator from the variability model of the application template and then modify the artifact at the place indicated by the locator.

Definition 88 (Customer-specific application solution). *A customer-specific application solution sol out of the set of customer-specific application solutions SOL , $sol = (am, vm, vmc_{sol}, CB_{sol})$, is defined as a tuple consisting of an application model $am \in AM$, a variability model $vm \in VM$, a customization $vmc_{sol} \in VMC$ and a set of component bindings $CB_{sol} \subseteq CB$. Note that the customization contains the selected alternatives and selected values for all variability points that have been bound in the previous phases, i.e., it is a combination of the customization already contained in the provider-specific template and the customization produced during solution engineering.*

Definition 89 (Solution creation). *Formally the solution creation sc is a map*

$$sc : PST \times VMC \rightarrow SOL$$

that transforms a provider-specific application template $pst \in PST$ and a customization vmc of the solution engineering variability points of that template into a customer-specific application solution $csa \in SOL$.

4.6 Summary and Conclusion

In this chapter the operational semantics for the variability metamodel introduced in Chapter 3 has been defined. Then, the generation of *customization flows* from the variability metamodel has been described. Different scenarios where customization flows are used in the context of Cafe have been presented. It has been shown how customization flows guarantee *correct* and

complete customizations, ensuring that the customer-specific application solutions generated from application templates through customization and thus are executable.

A *customization* of a variability model was defined that captures the customization choices a customer has performed. It is shown how customizations can be used to transform application templates into *provider-specific* application templates and those into *customer-specific solutions*. Having created a customer-specific application solution, this solution can now be provisioned. In the next chapter the provisioning of solutions is described in detail. It is shown how the individual components of a customer-specific application solution are deployed and thus a *customer-specific application deployment* is created.

PROVISIONING AND MANAGEMENT OF CAFE APPLICATION SOLUTIONS

This chapter describes the last step in the high-level Cafe development process as shown in Figure 3.3, the provisioning step. This step takes the customer-specific application solution as an input and produces a *valid* installation of the application for that customer. Valid means that all customizations made by a customer during the solution engineering phase, as well as the implicit and explicit requirements of the customer-specific application solution are met.

In this chapter concepts and an architecture for the provisioning and management of Cafe applications are introduced. During the discussion several properties of Cafe applications have to be taken into account:

1. As discussed in Section 3, Cafe applications are developed without knowing the concrete infrastructure they are later deployed on, therefore the infrastructure requirements must be mapped to concrete infrastructure

capabilities of providers.

2. Cafe application solutions can depend on components that must be provisioned by a provider as expressed by provider supplied components in the application model. Since different providers use different provisioning infrastructures, a common interface for these tools is needed to be able to describe necessary provisioning actions for an application solution without knowing the used infrastructure beforehand.
3. Cafe application solution provisioning must be atomic. Either all components are provisioned or none is provisioned.
4. Cafe application solution provisioning must take variability into account, i.e. the provisioning of a Cafe application heavily depends on the customizations contained in the customer-specific application solution and itself must bind the open variability points of the solution.
5. Cloud application solution provisioning must be SLA and cost aware. I.e. the customer-specific service and cost levels must be taken into account while provisioning a Cafe application.

To cope with these requirements, the following concepts are introduced in the next sections: in the first section of this chapter it is shown how concepts of service orientation can be used for the provisioning of Cafe application solutions. *Provisioning services* are introduced that encapsulate the specific provisioning tools used at different providers. These provisioning services can then be orchestrated by *provisioning flows*. It is shown how provisioning flows relate to the customer-specific application solution, especially the deployment and dependency relations contained in it. Then it is shown how customer-specific provisioning flows can be generated from the information contained in the application solution. By employing Web service technology, provisioning flows can orchestrate provisioning services spread over different providers and data centers. Additionally, provisioning flows are atomic, i.e. they ensure that an all-or-nothing semantics is preserved during provisioning.

In the second section of this chapter a unified interface for components at a provider is introduced. This *Cafe provisioning and management interface* (CPMI) abstracts from the concrete operations needed to provision and manage

a resource using a concrete provisioning infrastructure. The interface adopts a component-oriented view offering operations to create, query and terminate components via *component flows* that offer a standardized interface for a resource type and maps it to the interface of the concrete provisioning tools in use.

The third section of this chapter then describes how the CPMI and the service-oriented view on provisioning can be used to specify provider-independent provisioning flows that are used to provision a Cafe application solution for a new customer. It is also shown how during provisioning open variability points are bound and how component dependencies are resolved. Thus, the relation between provisioning flows and the customization flows introduced in Section 4.3 is clarified. It is shown how provisioning flows can be generated from the application model and the variability model of a customers-specific application solution.

The fifth part of this chapter deals with an optimization of the cost of running an application for a customer. Therefore, an optimization component is introduced that analyzes the components that need to be provisioned for a new customer. Depending on the SLA requirements and the cost of individual components, the optimization component binds concrete providers to a customer. The provisioning flow then ensures that the components are provisioned at the right provider.

5.1 Service-Oriented Provisioning and Management of Cafe Applications

In this section the concepts for the provisioning and management of Cafe applications are introduced. As mentioned before one requirement for Cafe applications is that they can be distributed across different providers. Additionally, a provider might use different specialized provisioning tools to provision different types of components. For example, provider A provisions an Apache Ode BPEL engine using the OpenQRM engine while at Amazon EC2 an EC2 specific provisioning engine is used to provision an image that contains the

Apache Ode BPEL engine. From the point of view of the application that must be provisioned, the actual provisioning engine that is used to provision the BPEL engine does not matter. The important thing is that after issuing a request the respective component is provisioned and can be used.

This is similar to a service requester in a service-oriented architecture that invokes a service. A service is a black-box with a published interface, but how it is implemented is not important to the service requester as long as the service fulfills the requested functional and nonfunctional properties.

5.1.1 Provisioning Services Offered by Different Provisioning Engines

In Cafe provisioning services are implemented by different provisioning and systems management tools. These tools range from cloud management tools hidden behind clearly defined interfaces, such as Amazon EC2s management interface [Ama09] or the interface for the open source Eucalyptus cloud management system [NWG⁺09] to dedicated provisioning and management systems, such as IBM's Tivoli System Automation Manager (TSAM) [IBM10], Sun's N1 Service Provisioning System (SPS) [Sun09] or the generic provisioning engine described in [CSL03]. Other provisioning services are provided by home-grown systems that are specifically developed to set up specific applications in one particular data center [KB04].

Currently, there is no accepted standard and agreement for standardized provisioning and management interfaces. Therefore, different provisioning engines and systems management tools offer different interfaces. These interfaces differ in various dimensions.

The first dimension is the dimension of granularity. Figure 5.1 shows three provisioning engines that expose interfaces with different granularity. One engine is an image distribution engine that allows provisioning virtual machine images such as an image containing a complete application stack consisting of an application server and a BPEL engine. When calling such an engine typically an image id must be supplied telling the engine which image it must provision [Ama09, RBL⁺09]. In Figure 5.1 the image with the id 1 denotes that the application server and BPEL engine image must be provisioned.

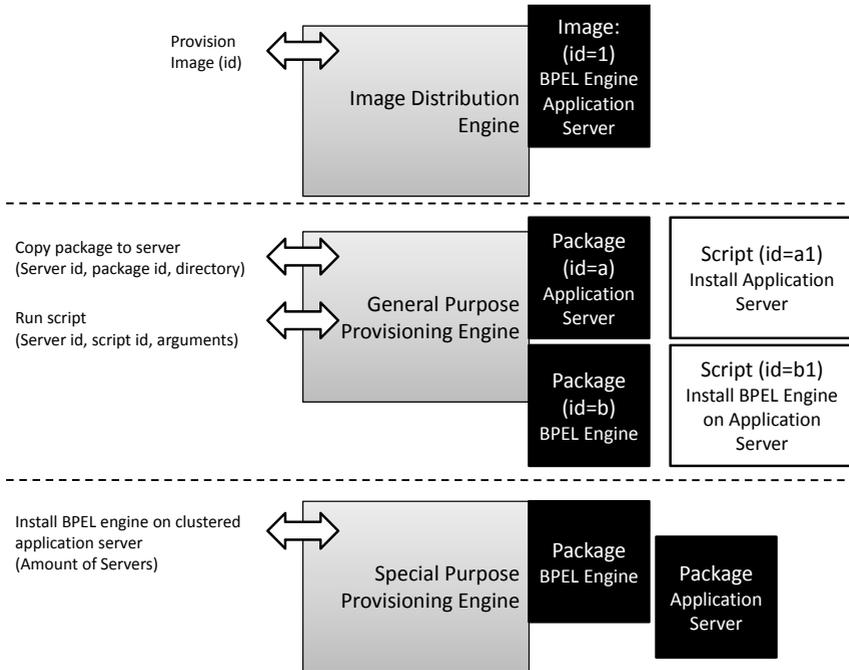


Figure 5.1: Different granularity of provisioning services

The second engine shown in Figure 5.1 is a general purpose provisioning engine that allows provisioning an application server on a standard Linux server, as well as offers operations to provision other components (such as a BPEL engine) on that application server. It exposes two operations typically found in such engines [Sch08, GGS07]. The first operation copies an application package (referenced by an id) to a server, the second operation then allows to run a script that performs the necessary steps on the server to install a package. This script is specific for each component the engine can provision. In this case scripts to install the application server and the BPEL engine need to be present in the system. When provisioning the BPEL engine at first the application server package must be copied to the server, then the corresponding installation script must be executed, then the BPEL engine package can be

copied and the script that installs the engine on the application server can be executed. Compared to the single call of the image distribution engine, this requires more calls but allows more fine grained control. For example, the general purpose provisioning engine could be used to provision two application servers and BPEL engines that form a high-availability cluster with failover capabilities by supplying the necessary arguments to the install scripts.

The third provisioning engine shown in Figure 5.1 is a special purpose provisioning engine that can provision an application server and BPEL engine in a cluster. When calling this engine the amount of servers must be supplied and the entire configuration of servers and additional load-balancers is done internally.

This example corresponds to the situation in established products, while the interfaces of Amazon EC2 or a VMWare server only allow setting up complete machine images, the interfaces of provisioning engines such as Sun's N1 SPS or IBM's Tivoli Provisioning Manager offer more fine-grained operations such as "deploy an application server on an already installed server". The problem of different granularity of services is a problem that is also known in the business domain. While one CRM system exposes a service that allows to retrieve a customer and all the associated data at once, another CRM system might require several service calls to perform the same. In the business domain composite services are used to aggregate services on a lower granularity-level into higher-level services. The technique often used for that is called "orchestration" [Pap03, Pel03]. The same technique can be employed for the Grid [EBC⁺05, Ley06, Slo06, DFH⁺07, SM07, MKL09] and for the provisioning domain [KB04, ML08b]. Similar to the business domain, orchestration techniques can be employed to orchestrate lower-level provisioning services into higher-level ones. In the next section *provisioning flows* are introduced. These provisioning flows orchestrate provisioning services into higher-level ones.

The second dimension in which the interfaces of provisioning engines differ, is the technology dimension. Some interfaces for provisioning engines, for example, those of Amazon EC2 and IBM Tivoli Provisioning Manager use Web service technology while others use Java interfaces (Sun N1), REST-based Interfaces (GoGrid, IBM TSAM) or offer command line interfaces (Sun N1,

OpenQRM, EC2). In order to expose a uniform interface, these interfaces need to be wrapped with a common technology.

The third dimension where interfaces of different provisioning engines differ are the offered operations and parameters for these operations.

In order to overcome the problems of different operations and different technologies a *provisioning unification layer* is introduced. The provisioning unification layer provides a standardized component-oriented interface to the outside. This interface is from now on called the *Cafe provisioning and management interface* (CPMI). By offering the CPMI the provisioning unification layer hides the idiosyncrasies of the underlying provisioning engines to the upper layers (cf. Figure 5.4). The provisioning unification layer contains mapping functionality from the CPMI to the interfaces of the provisioning engines. The CPMI is described in more detail in Section 5.2.

5.1.2 Provisioning and Management Flows

A *provisioning flow* is defined as an orchestration of one or more provisioning services. Provisioning flows are used to create higher-level provisioning services from lower-level provisioning services. Similar to orchestrations in the business domain, provisioning flows therefore offer a recursive aggregation model for provisioning services.

A provisioning flow is a special kind of *management flow*. A management flow is an orchestration of a set of systems management services that are used to perform a certain systems management task in one or several data centers. In this thesis, the focus lies on provisioning flows, however, in a real product, management flows that perform system management tasks such as adding or removing servers, installing updates etc. must also be considered.

Figure 5.2 shows how the interfaces of the general purpose provisioning engine and the special purpose provisioning engine shown in Figure 5.1 can be brought to the same level of granularity with a provisioning flow. The provisioning flow orchestrates the provisioning services provided by the general purpose engine into a higher-level provisioning service (namely “install BPEL engine”) which is on the same level as the provisioning flow of the special

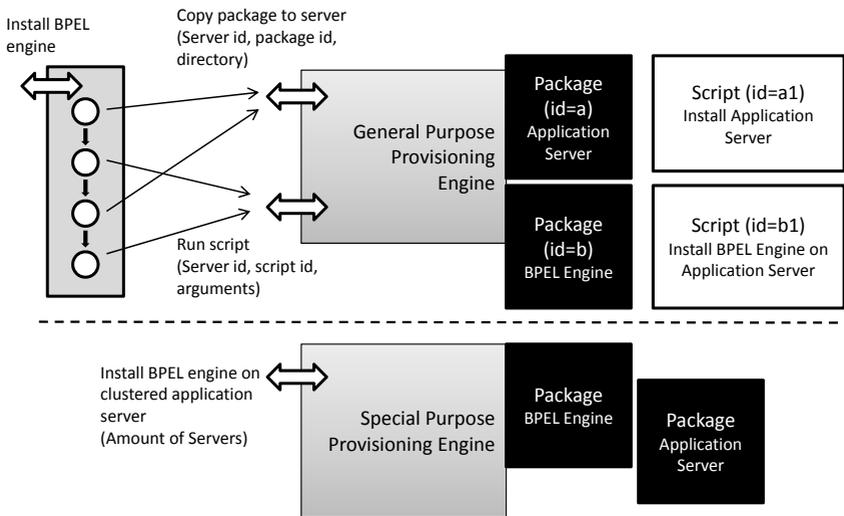


Figure 5.2: Using a provisioning flow to overcome granularity differences

purpose engine. Additionally, the provisioning flow could be extended to allow parameterization of the amount of servers which should be used. Furthermore, a step could be entered at the beginning that searches for available servers thus preventing that the user must know which servers he wants to use. Therefore, the provisioning flow could issue a call on a management database that knows which servers are available. Thus, in order to exploit the full potential of heterogeneous environments, provisioning flows must not only be able to orchestrate the provisioning services of a single provisioning engine, but also orchestrate the provisioning services of different engines and management tools. For example, a provisioning flow as the one shown in Figure 5.2 could install a BPEL engine on a server in the provider’s own data center using the general purpose provisioning engine and deploy a fail over environment in another provider’s data center using that providers provisioning interface while exposing a single operation “install BPEL engine with fail over capabilities” to the outside.

Provisioning and management flows can orchestrate provisioning and man-

agement services in one single data center as well as across different data centers and across different providers. Similar to workflows in the business domain, provisioning and management flows can be defined using *virtual services*. This means that the binding to concrete services can be deferred to deployment time or even runtime. This allows to specify provisioning and management flows without knowing the concrete endpoints of provisioning and management services. In the context of Cafe applications this means that provisioning and management flows can be attached to Cafe applications without knowing on which concrete infrastructure the applications are later provisioned. The binding of the provisioning flow to concrete provisioning services is then done at run-time by a *provisioning bus*. In short a provisioning bus ([MLW⁺09, Wie08]) is a special enterprise service bus that virtualizes provisioning and management services as well as resources. The late-binding of provisioning and management flows to provisioning and management services across different engines requires a standardized interface for provisioning and management services which is given by the CPMI.

To distinguish provisioning flows that operate directly on the provisioning engine operations such as the provisioning flow shown in Figure 5.2 from provisioning flows that operate on the CPMI, two terms are introduced:

- *Engine-specific provisioning flows* are provisioning flows that operate on the provisioning operations of one or several provisioning engines and thus are specific for these types of engines. These flows are typically used to orchestrate low-level operations of a provisioning engine into a higher-level operation. These provisioning flows are highly provisioning engine specific and thus out of scope for this thesis.
- *Engine-independent provisioning flows* are provisioning flows that operate on the generic engine-independent Cafe provisioning and management interface (CPMI). These engine-independent provisioning flows can be used to specify an orchestration of generic provisioning actions independent of the concrete type of engine later used. However, to be able to specify such engine-independent provisioning flows, the engine must offer an interface that corresponds to the CPMI, or the API must be

wrapped to be compliant to the CPMI. These flows are typically used to orchestrate different provisioning engines so that they provide different components that are needed to provision a customer-specific application solution.

5.2 Cafe Provisioning and Management Interface (CPMI)

The *Cafe provisioning and management interface* (CPMI) is an interface offering a set of provisioning and management services that can be orchestrated by provisioning flows into higher-level provisioning and management services. The CMPI abstracts from the concrete provisioning engines used. The CPMI offers a component-oriented view on provisioning and management. I.e., it offers provisioning and management services that are related to components rather than to provisioning and management tools. This is important to note, as the provisioning and management of one component (e.g. a physical server, or a BPEL engine including application server, middleware and physical machines) might be carried out by multiple provisioning and management tools. By offering a component-oriented view, the CPMI does not require users of the CPMI to be aware of the concrete provisioning infrastructure in place. This is especially important in the context of Cafe applications, where the provisioning flows for an application that use the CPMI are defined without prior knowledge of the provisioning infrastructure that is later in place. Furthermore, this enables Cafe providers to follow a best-of-breed strategy in choosing their provisioning and management tools, as the provisioning flows for Cafe applications can be defined on top of the CPMI and not on top of specific provisioning and management tools.

5.2.1 Mapping Provisioning and Management Services to Components via Component Flows

Most provisioning and management engines and tools offer a operation-oriented approach to the manipulation of resources and components [EMME⁺06, Sch08]. As a consequence, when using such a tool the client (a human or

an application) of such a tool must know which operation on which engine at which endpoint must be invoked. For example, a client must invoke the `run-instances` operation when the client wants to start a new machine image instance on Amazon EC2. Resource type identifiers serve as identification of the concrete type of resource to start. For example, in the case of Amazon EC2 an AMI id that identifies the type of machine image to start has to be supplied when invoking the `run-instances` operation. Other provisioning engines operate in a similar fashion with other operation names, states and parameters.

The CPMI thus describes a set of standardized states and operations of components so that the Cafe provisioning mechanism can be defined independently from the concrete provisioning engines used. The details of these states and operations are described in Section 5.2.2. These states and operations are similar to, but differ from the APIs and state models of existing provisioning engines whose interfaces also differ among each other. However, to be able to describe provisioning flows on top of the CPMI, it is necessary that the APIs and state models of existing provisioning engines are integrated with the CPMI. To be able to use the CPMI with existing engines, two general approaches exist:

1. Adapt all existing provisioning engines and systems management tools to use the CPMI, or
2. Wrap existing provisioning engines and systems management tools so that they expose the CPMI to the outside. This includes mapping states and operations in the CPMI to states and operations of the provisioning engine.

In this thesis the second approach of wrapping existing provisioning engines with CPMI-compliant wrappers is pursued for the following reasons: First, it is not feasible to adapt all existing provisioning engines to the CPMI. Second, one component might be managed by several purpose-built scripts or by different provisioning and management tools that can be integrated using a wrapper. Third, wrappers are needed anyway for technology reasons, as different provisioning engines use different API technologies. These can then also be wrapped and APIs can be invoked in a distributed fashion.

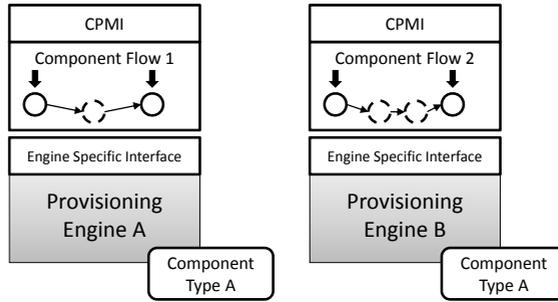


Figure 5.3: Different component flows for the same component type but different provisioning engines

In the following *component flows* are introduced that implement the CPMIs state and operation model for a given component type and provisioning engine type using workflow technology [ML08b].

At first the relation between component flows, provisioning engine types and component types must be clarified: One component flow exists per (component type, provisioning engine type) pair (see Figure 5.3). All component flows in general provide the CPMI to the outside thus they all have the same basic interface (see Section 5.2.2). Component flows for provider components and components capable of being deployed in single (configurable) instance mode additionally provide the respective operations as described in Section 5.2.3 and Section 5.2.4. Figure 5.3 shows an example of two provisioning engines A and B that both are able to provision components of component type A. They both have different interfaces, therefore two different component flows are needed. The component flows expose the same interface to the outside but are different (as indicated by the dashed activities). For example, provisioning engine B requires two steps to provision a component, whereas provisioning engine A requires only one step.

Component flows for the same component type additionally also provide the same set of properties as an answer to a `getProperties` request, namely the set of the bound variability points and selected values and alternatives of the component type in form of a customization document.

As a result two different component flows for the same component type using two different provisioning engine types provide the same interface to the outside. In the implementation chapter (Chapter 6) different technologies to implement component flows as well as actual component flows for different component types and provisioning engines are shown.

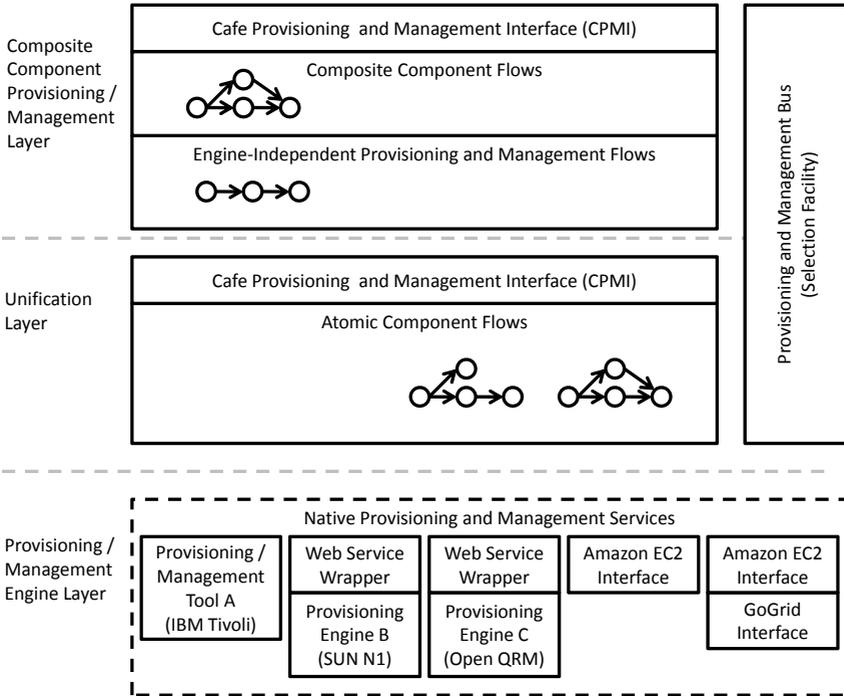


Figure 5.4: Provisioning and management infrastructure

Figure 5.4 summarizes the concepts that are presented in this section. Beginning from the bottom of the figure, different provisioning engines are first wrapped to be able to be called from component flows. Depending on the implementation technology for the component flows, different wrappers are needed. In this thesis, component flows are implemented in WS-BPEL, thus the interfaces of the provisioning engines are wrapped as Web Services to be able to be invoked from a WS-BPEL process. For example, the Sun N1 SPS engine

is made accessible via a Web service. Component flows are specified for the component types supported by the different engines. These *atomic component flows* then implement the CPMI. They are called atomic component flows as the components they represent cannot be further decomposed. On top of that composite components can be defined using engine-independent provisioning and management flows that access the atomic component flows for the components they compose. Again the composite components are wrapped with composite component flows that expose them as components with the generic CPMI to the outside.

Figure 5.4 also shows an orthogonal component, the *provisioning and management bus* or *selection facility* whose functionality will be explained in Section 5.3.

5.2.2 Generic Component States and Operations in the CPMI

To abstract from concrete provisioning engines a set of generic standard states and messages are introduced that build the foundation for the CPMI. Figure 5.5 shows these states and messages for basic components. Basic components are those components that can have one single customer (i.e. are provisioned using the multiple instances pattern) and do not allow other components to be installed on top of them, i.e. they are not provider components. In the following a process notation (BPMN) is used to describe the generic component states and messages. This notation is used for several reasons: first, it allows to not only describe the messages that can be received and sent by the component flows, but also the sequence and conditions under which these messages can be received and sent. Since the wrappers from the CPMI to the provisioning engines are defined as component flows anyway the definition of the CPMI in BPMN can be used to derive component flows from this description.

The messages shown in Figure 5.5 can be used by provisioning flows to set up basic components. The states and messages shown in Figure 5.5 are explained in detail below. The messages and states shown in Figure 5.5, Figure 5.7, Figure 5.8 and Figure 5.9 do not show error states and messages; however,

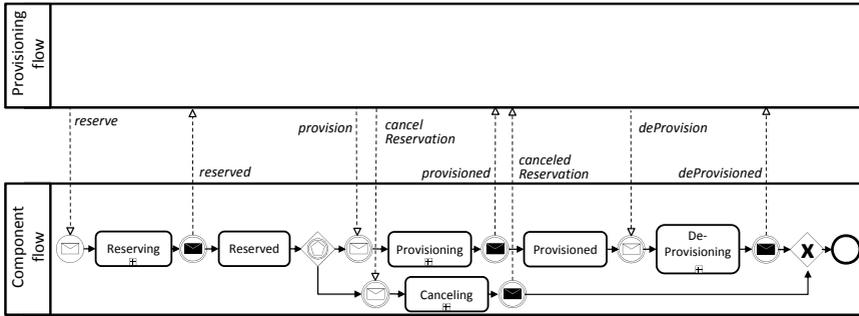


Figure 5.5: Basic component states and messages

these are explained in the text where needed. In case no error handling is described corresponding manual intervention by an administrator is needed.

Figure 5.6 shows the message flow between a provisioning flow and a basic component flow. The figure also shows the types of customization documents that are included in the messages. In the following the phrase “sending a message to a component” will be used to denote that in fact a message is sent to an “instance of the component flow that represents the component”. Note that the `getProperties` and `setProperties` operations shown in Figure 5.6 are not shown in Figure 5.5 in order to not clutter the Figure, however they are available during the whole life-time of a component flow instance.

Reserving a new Component. The first step in provisioning a component is to reserve it. This is important, as it helps to guarantee all-or-nothing semantics and reduces the need for compensation after having already provisioned some components. A provisioning flow must thus first reserve all components. Only if this is possible, the more complex provisioning steps can be executed. If the reservation of all needed components is not possible, provisioning of the complete solution is not possible and the provisioning flow must abort and cancel all already performed reservations.

To reserve a component, a `reserve(sc-doc)` message must be sent to the component flow representing a provisionable component. Provisionable

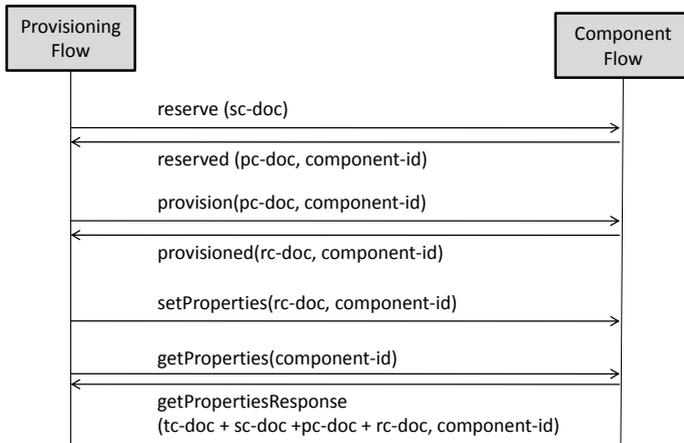


Figure 5.6: Messages and exchanged customization documents during successful provisioning of a component

components are identified in Cafe via an EPR that references their provisioning service. This message must contain or reference the solution engineering customization document (sc-doc) for that component. The component then transitions from state *Not Available* to state *Reserving* and a corresponding instance of its component flow is started. If the component can not be reserved for any reason, a `reservationNotPossible(reason)` message containing the reason why the reservation is not possible is returned to the sender.

Otherwise a `reserved(component-id, pc-doc)` message is returned and the component transitions to state *Reserved*. The component-id references the newly reserved component in all following interactions. Thus regarding the nature of component flows which are realized as workflows, the component-id can be seen as the *correlation token* that correlates all future requests with the instance of the component flow that has been created to represent the component. The returned pre-provisioning customization document (pc-doc) can contain already bound variability points that must be bound during the pre-provisioning customization phase. Normally these must be bound by the provisioning flow, however, the provisioning engine can already bind those

variability points of which it is aware. These variability points must then not be bound by the provisioning flow.

Canceling a reservation. Once a component is in state *Reserved*, it can be either provisioned (see next paragraph) or the reservation can be canceled in case the component is not needed. Canceling a component reservation is done by sending a `cancelReservation(component-id)` message to the respective component. The component is then in state *Canceling*. Once the canceling has completed, a `cancelCompleted(component-id)` message is sent to the client.

Provisioning a new component. A component in state *Reserved* can be provisioned by sending a `provision(component-id,pc-doc)` message. Provisioning means that the necessary steps are taken by the corresponding provisioning engine to set up and start the component. After having received a `provision` message, the code for the component must be prepared, i.e. the bindings of variability points contained in the pre-provisioning customization doc (`pc-doc`) as well as those submitted with the solution engineering document submitted during the reservation must be applied to the code that lies in the solution repository. How this is done is up to the provider of the component. In case the component is part of a (nested) Cafe application template an `applyCustomization(component-id,pc-doc)` message is sent to the repository which will then perform the necessary steps to configure the artifacts that implement the solution with the submitted `pc-doc`. The actions performed by the repository are the same as the actions performed on a template during the solution creation phase (see Section 4.5.2). Afterwards the repository returns an identifier that can be used to load the modified code from the repository. This identifier can then be used by the provisioning engine that implements the provisioning of the new component to download the code and deploy it. Since this is specific to the provisioning engine used, it is not explained in detail here. However, in the implementation chapter it is described how this is done, for example, for Amazon EC2 (see Section 6.2.11).

As soon as the provisioning engine has performed all necessary steps to set up the component a `provisioned(component-id,rc-doc)` message

is sent and the component transitions to state *Provisioned*. Once a component is provisioned all its properties are available via its running customization document (rc-doc). In particular, this includes properties such as the EPR of the component that can now be used to configure other components.

In case provisioning of the component is not possible a corresponding `provisioningNotPossible(component-id,reason)` message is sent.

De-provisioning a component. Once a component is in state *Provisioned* it can be de-provisioned by sending a `deProvision(component-id)` message. The component is now in state *DeProvisioning*. The corresponding provisioning engine must then take the necessary steps to remove the component. Having de-provisioned a `deProvisioned(component-id)` message is sent to the client. The component is now in state *Not Available* again and its component-id is destroyed. The instance of the component flow representing the component can now terminate.

Getting component properties. The properties of already provisioned components can be queried in all states sending a `getProperties(component-id)` message to the instance of the component flow that represents the component. In case a component has not been reserved yet, no corresponding instance of the component flow exists. To retrieve the already set properties (i.e., those set during the template customization phase) the corresponding provisioning service can be queried with a `getProperties` message. The `getPropertiesResponse` is the response for a `getProperties` message and returns all available customization documents of a component. Depending on the state of the component, different customization documents are already available.

Setting properties at runtime. Once a component is in state *Provisioned* running variability points of that component can be bound by sending a `setProperty(component-id,rc-doc)` message to the component. The component flow must then set these properties, i.e. by calling a configuration interface of the component and return a `PropertiesSet(component-id)` message. An example for properties that can be set when a component is

already running are for example administrator passwords. Others, such as EPRs are set during the provisioning and cannot be changed later.

5.2.3 Special Operations for Provider Components

Special operations are needed for provider component types (cf. Definition 17), i.e. component types that allow the deployment of other components on components of that component type. These special operations are needed to deploy the other components. Components that are of a provider component type are from now on called *provider components*.

A component that is to be deployed on a provider component is called a *deployment* from now on. A provider component can host multiple deployments. The subprocess shown in the upper part of the provider component swimlane in Figure 5.7 represents the states of a deployment. Note that the operations and the states for the actual component that is represented by the deployment are basic operations and states as shown in Figure 5.5. The states and operations for provider component types differ from those of basic component types in the states and operations shown in Figure 5.7.

Reserve a Deployment. Once a provider component is in state *provisioned* it must offer one separate `reserveDeployment(sc-doc, repositoryEPR)` operation for each implementation type that can be deployed on the provider component. The `repositoryEPR` parameter contains the location where the code for the component can be found. This can either be an attachment to the message or the endpoint reference of a repository such as the repository that stores the components of customer-specific solutions of Cafe application templates. Once the deployment has been reserved, the provider component responds this message with a `reservedDeployment(deployment-id)` message that acknowledges the processing of the message. The `deployment-id` contains a unique id to identify the deployment on the provider component. The deployment is now in state *Reserved*. In case the reservation fails, a `reserveDeploymentNotPossible` message is sent, the provisioning flow must then abort the provisioning of the whole application.

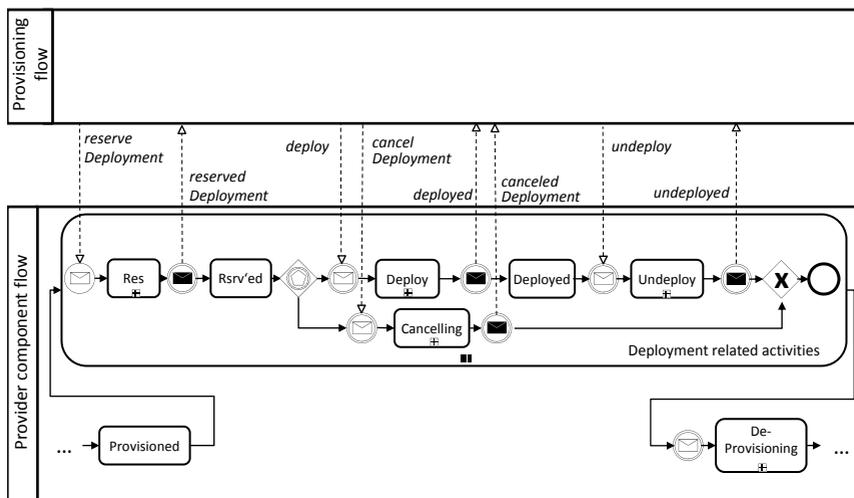


Figure 5.7: Additional states and operations for provider components

Cancel Deployment Reservations. In case a reservation must be canceled, this can be done by sending a `cancelDeployment(deployment-id)` message to the provider component which will then cancel the deployment with the id supplied. When the deployment is canceled a `canceledDeployment(id)` message is returned.

Deployment of Components on a Provider Component. Once a deployment has been successfully reserved, the component associated with the deployment can be deployed by sending a `deploy(deployment-id,pc-doc)` message. The deployment is then executed. In case of a component in a Cafe application solution that must be deployed, at first the customizations from the template customization and solution engineering phase for that component captured in the `tc-doc` and `sc-doc` must be applied. Thus a `applyCustomizations(tc-doc,sc-doc)` message is sent to the repository that stores the code for the component. Then the steps as described in the solution creation phase in Section 4.5.2 are performed. The repository answers with a `appliedCustomization(EPR)` message that contains

an EPR where the code for the customized component can be downloaded. Then the associated provisioning engine provisions the component on the provider component, i.e., by copying it from the repository to the provider component. However, how this is done depends on the provisioning engine used. Once the component has been deployed on the host component a `deployed(deployment-id,rc-doc)` message is returned containing the already bound variability points that must be bound during runtime in the rc-doc. In case the deployment fails a `deploymentNotPossible(deployment-id)` error message is sent.

Undeployment of Components from a Provider Component. Once a component is successfully deployed, it can be undeployed by a client by sending a `undeploy(deployment-id)` message which is answered by the provider component by an `undeployed(deployment-id)` message once the undeployment has finished. How the actual undeployment is done depends on the provisioning engine used.

Getting and setting properties of a deployment. Getting and setting properties of a deployment works as getting and setting properties of a component by sending a `getProperties(component-id,deployment-id)` or `setProperties(component-id,deployment-id,rc-doc)` message.

5.2.4 Special Operations for Single (Configurable) Instance Components

Components deployed in the single instance or single configurable instance pattern can be used by several customers at once. However, they must keep track which customers are using them. This is, for example, used to determine if such a component can be de-provisioned as they can only be de-provisioned when no customer needs them anymore. In addition to that, components deployed in the single configurable instance pattern can be configured individually for a subscription, thus they need interfaces where new customers can send their customization documents to configure the component for their needs. In the following the term *subscription* is used to describe the usage of a component by a customer. Thus, single (configurable) instance components must offer

operations to reserve subscriptions as well as to subscribe and unsubscribe customers. One customer can have multiple subscriptions for a component, for example, if that customer wants to use the component with multiple QoS levels or different functionality.

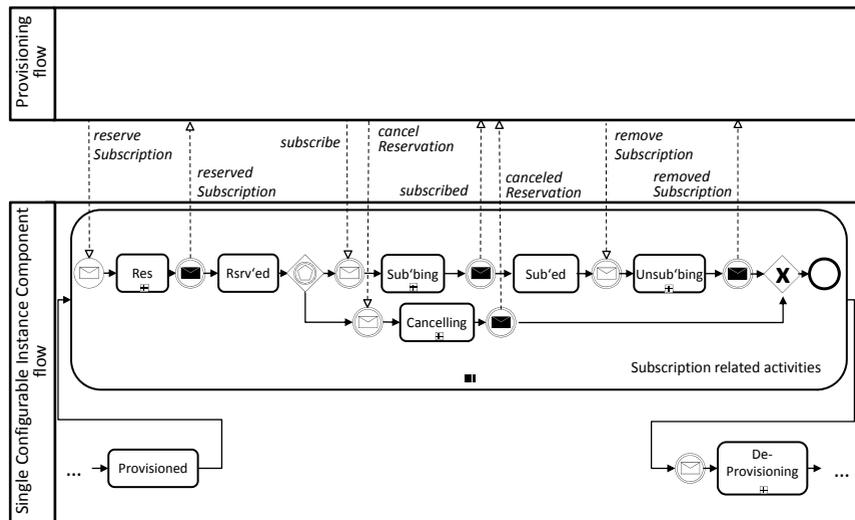


Figure 5.8: Additional states and operations for single configurable instance components related to subscriptions

Creating a Subscription Reservation for a Customer. Before subscribing, the subscription must be reserved. This is done for the same reason a reservation is required for basic components, thus to ensure that first all required components are reserved and then the actual provisioning takes place which prevents unnecessary provisioning. Once a client has sent a `reserveSubscription(sc-doc)` message, the component is in state *reserving*. The reservation must include the `sc-doc` that will later be used to customize the reserved component. This is already necessary at this stage since it may contain customization options that influence the allocation of resources at the provider.

If the reservation is possible, the provider responds to the reservation request by sending a `reservedSubscription(subscription-id,pc-doc)`

message including a `subscription-id` that uniquely identifies the subscription reservation on the component. The pre-provisioning customization document sent with the message contains the already bound variability points that must be bound during the pre-provisioning. The other pre-provisioning variability points must be bound by the provisioning customization flow. If the reservation is not possible a `reservationNotPossible` message is returned.

Canceling a Subscription Reservation. Next, the client can either cancel the reservation by sending a `cancelReservation(subscription-id)` message or subscribe to the component. Once a provider has received a cancellation message it responds with a `canceledReservation(subscription-id)` message to acknowledge the successful cancellation.

Subscription of a new Customer. To subscribe a customer, a client must send a `subscribe(subscription-id, tc-doc)` message to the component. Then depending on the provisioning engine used or the *configuration interface* of the component, the component is configured for the new customer. This is done by transforming the `tc-doc` and the previously submitted `sc-doc` into a format the provisioning engine or the configuration interface understands and invoking the provisioning engine or configuration interface. The configuration interface of a component is, for example, an `addCustomer` method that takes the configurations of that tenant as an input. In [Shi10] such a configuration interface has been developed for IBM's Sametime 3D Web interface which is multi-tenant aware (see also Section 7.6.2). Having added the new customer a `subscribed(subscription-id, rc-doc)` message is sent that contains the already available runtime properties in form of a set of bound variability points in the running customization document (`rc-doc`). In case the subscription fails a `subscriptionNotPossible(subscription-id)` message is returned.

Removal of a customer. Similar to the subscription messages the messages `removeSubscription(subscription-id)`, and the subsequent message `removedSubscription(subscription-id)` are exchanged when unsubscribing a customer from a single (configurable) instance component.

Getting and setting properties of a subscription. Getting and setting properties of a deployment works as getting and setting properties of a component by sending a `getProperties(component-id, subscription-id)` or `setProperties(component-id, subscription-id, rc-doc)` message.

Single Configurable Instance Provider Components. Figure 5.9 shows the states for components deployed in the single configurable instance pattern that additionally are also provider components. In this case the additional states and operations for provider components are only available after a customer has subscribed for the component. The additional states and messages are the same as with normal provider components, however, the subscription-id must be submitted when reserving a new deployment so that the deployment can be correlated with the correct subscription. In subsequent message exchanges the deployment-id identifies a deployment uniquely among all deployments and subscriptions of a component.

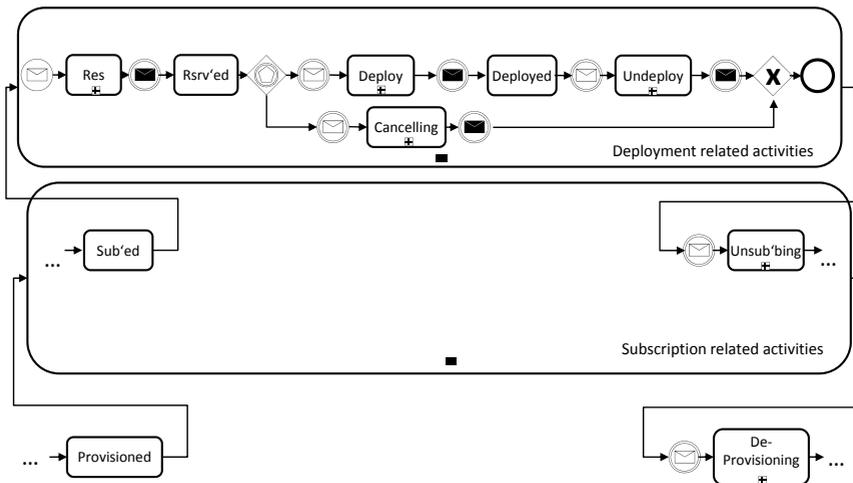


Figure 5.9: States for single configurable instance provider components

5.3 Application Provisioning

In this section the actual provisioning of a Cafe application is described. Figure 5.10 shows the subprocesses of the Cafe provisioning phase.

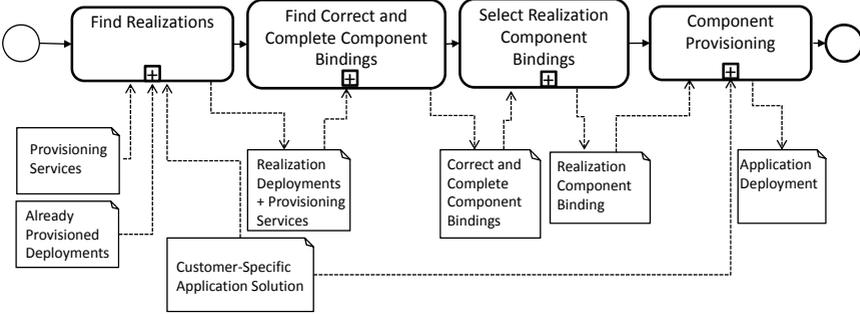


Figure 5.10: Overview of the provisioning subprocesses and required artifacts

As shown in Figure 5.10 the input for the provisioning of an application solution for a new customer is the customer-specific application solution that has been produced during the customization of a Cafe application template. Thus, provisioning can be seen as the action that makes a customer-specific application solution available for a new customer. As shown in Figure 5.10 the two additional main aspects that must be taken into account by the provisioning besides the customer-specific application solution, are the already provisioned components as well as the available provisioning services. Already provisioned components are important, as some components already provisioned for other customers may be reused for the new customer. The available provisioning services are important for the provisioning, as they allow to provision new components that may be needed for a new customer.

The output of the provisioning is then a customer-specific *application deployment*. This customer-specific application deployment contains all provisioned components that are necessary for a customer to use the application. These components are also configured such that the application can immediately be used. In particular, this means that the variability points that must be bound during the provisioning are bound.

5.3.1 Find Realizations

As a first step during provisioning, the already provisioned deployments and the provisioning services at a provider are examined if they are suitable to run individual components of the application model later on. This step is called the *find realizations subprocess*. Such a step can, for example, be based on the information in a *configuration and management database* (CMDB) [MSB⁺07] in which the currently provisioned environment of a provider is captured; in our prototypical implementation this information is held in the “already provisioned component database”. This subprocess consists of two distinct steps that can be executed in parallel. The *find realization provisioning services* step, and the *find realization deployments* step. A *realization provisioning service* is a provisioning service that can provision a set of components as specified in the application model of a customer-specific solution. A *realization deployment* is an already deployed set of components that are suitable to represent a set of components in an application model. Formal definitions and detailed descriptions of the steps are given below.

Depending on the multi-tenancy pattern and the implementation type of a component c in the application model $am = \pi_1(csa)$ of a customer-specific application solution $csa \in SOL$, different actions must be taken during the *find realizations* subprocess.

Three sets of components can be distinguished in an application model of a customer-specific application solution: Components that can be *provisioned*, components that can be *reused* and components that can be *deployed*.

Definition 90 (Components that can be provisioned). *The set of components $C_{prov}(am)$ in an application model $am = \pi_1(csa)$ of a customer-specific application solution csa that can be provisioned, are all components in that application model that do not have an implementation type of external. Thus $C_{prov}(am) = \{c \in \pi_1(am) \mid implT(impl(c)) \neq \text{external}\}$.*

The semantics behind the set of components that can be provisioned is the following: in case a suitable provisioning service is found for any of these components, this provisioning service can be used to make this component

available for the application solution, if none is found, the component cannot be provisioned by that provider or it must be reused because it is already provisioned. External components cannot be provisioned, therefore, no provisioning service is needed for those components.

Definition 91 (Reusable Components). *The set of components $C_{reuse}(am)$ in an application model $am = \pi_1(csa)$ of a solution $csa \in SOL$ is the set of components for which already provisioned components can be reused. This set is defined as the set of components that can be provisioned, having an assigned multi-tenancy pattern of single instance, or single configurable instance.*

$$C_{reuse}(am) = \{c \in C_{prov}(am) \mid p(c) \in \{\text{singleInst}, \text{singleConfInst}\}\}$$

Components with the multiple instance multi-tenancy pattern cannot be reused as they require a separate deployment of the component for each usage of the component.

Definition 92 (Components that can be deployed). *The set of components $C_{deploy}(am)$ in an application model $am = \pi_1(csa)$ of a solution $csa \in SOL$ is the set of components that can be deployed because the code for these components is contained or referenced in the application solution. Thus, this set is defined as the set of components that are not of implementation type external or provider supplied.*

$$C_{deploy}(am) = \{c \in C_{prov}(am) \mid implT(impl(c)) \notin \{\text{external}, \text{providerSupplied}\}\}$$

In case a component c is in the set of components that can be provisioned, i.e. $c \in C_{prov}(am)$, a suitable provisioning service needs to be determined that can provision the component. In case a component can be reused, i.e. if $c \in C_{reuse}(am)$ an already provisioned deployment needs to be found that contains a component that can be reused. Additionally, a suitable provisioning service needs to be determined since provisioning a new component might be cheaper, or because no suitable already provisioned component might exist.

Before describing the steps that are performed to find realization provisioning services and components for the components in the application model, a special case has to be examined. When finding realization provisioning services and realization components the semantics of provider supplied components must be considered. In case one or more provider supplied components $\{c_1, \dots, c_n\}$ have a deployment relationship on another provider supplied component c_p , they form a provisioning group. The semantics of a *provisioning group* is that all components in that set must be provisioned by one provisioning service that provisions all components in the provisioning group with the required deployment relations. To facilitate the definition of the following functions and the provisioning algorithm, the set of provisioning groups of an application model also includes all provider supplied components that do not have a deployment relationship on another provider supplied component. These form a provisioning group with a single component.

Definition 93 (Provisioning groups of an application model). *Given an application model $am \in AM$, the set of provisioning groups $PG(am)$ of the application model is defined as a set of a set of components C out of a set of components of the application model $C \subseteq \pi_1(am) \setminus \emptyset$ that are of implementation type provider supplied and that are connected via deployment relations.*

Thus given the set of components of an application model $C = \pi_1(am)$, the deployment relations of an application model $D = \pi_5(am)$, the implementation, implementation type map of the application model $implT = \pi_8(am)$ and the component implementation map $impl = \pi_7(am)$, the set of provisioning groups of an application model is defined as:

$$PG(am) = \{CS \subseteq C \mid \forall c_1, c_2 \in CS \exists d \in D : (\pi_1(d) = c_1 \vee \pi_2(d) = c_2) \wedge implT(impl(c_1)) = \text{providerSupplied}\} \wedge implT(impl(c_2)) = \text{providerSupplied}\}$$

The definition of the provisioning groups of an application model (see Definition 93) contains all provisioning groups that can be derived from the application model. In particular, it can contain provisioning groups that have

only a subset of a set of connected provider supplied components. However, for the provisioning the *maximal provisioning groups* are needed because all connected provider supplied components must be provisioned together.

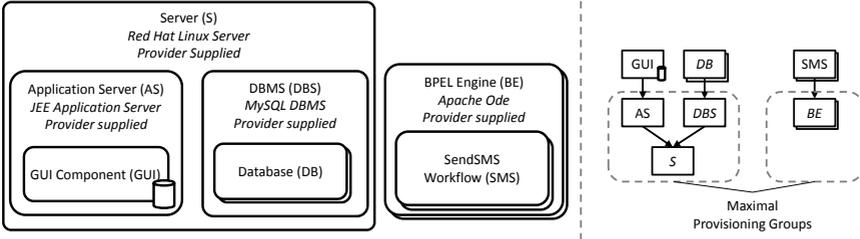


Figure 5.11: Condensed representation of an application model and maximal provisioning groups

Definition 94 (Maximal provisioning groups of an application model). *The set of maximal provisioning groups $PG_{max}(am)$ of an application model $am \in AM$ is defined as those provisioning groups for which no other provider supplied component exists in the application model that is connected directly to that maximal provisioning group.*

Given the set of components of an application model $C = \pi_1(am)$, the deployment relations of an application model $D = \pi_5(am)$, the implementation, implementation type map of the application model $implT = \pi_8(am)$ and the component, implementation map $impl = \pi_7(am)$ the set of maximal provisioning groups of an application model is defined as:

$$\begin{aligned}
 PG_{max}(am) = \{ & pg \in PG(am) \mid \neg \exists d \in D : \\
 & (implT(impl(\pi_1(d))) = providerSupplied \\
 & \wedge \pi_2(d) \in pg) \\
 & \vee (implT(impl(\pi_2(d))) = providerSupplied \\
 & \wedge \pi_1(d) \in pg) \}
 \end{aligned}$$

i.e. these are the connecting components of the graph of provider supplied components.

The definition of the maximal provisioning groups is important for the finding of provisioning services because a qualifying provider must supply provisioning services that can provision each of the maximal provisioning groups.

Figure 5.11 shows an example of an application model of a solution. The example follows the example shown in Figure 3.5 and consists of a *GUI (GUI)* component and a *Database (DB)* component whose implementations are supplied with the solution. Those two components have a deployment relationship on an *Application Server (AS)* component and a *DBMS (DBS)* component respectively, that are of implementation type *provider supplied*. The *Application Server* and *DBMS* components in turn have a deployment relationship on a *Server* component (S) that is also of implementation type *provider supplied*. The server is explicitly modeled, as the application developer wanted to make sure that the *Application Server* and the *DBMS* are deployed on the same server. If this is not relevant, the *Server* component would not need to be modeled. The application model also contains a *SendSMS Workflow (SMS)* component that has a deployment relationship on a *BPEL Engine (BE)* component that is provider supplied. The right part of Figure 5.11 shows a condensed representation of the application model that shows the components (nodes) and the deployment relations (edges). The edges denote that the source of the edge must be deployed on the target of the edge. Components written in italics are components that must be provider supplied. The icons for the multi-tenancy patterns are the same as in the visual representation of the application model. The right part of Figure 5.11 additionally shows the two maximal provisioning groups of the application model, namely a group consisting of the *Application server (AS)*, the *DBMS (DBS)* and the *Server (S)* that must be provisioned together, as well as a second group that only consists of the *BPEL Engine (BE)*. This condensed view is used in the following images of this chapter for brevity reasons.

Find Realization Provisioning Services. In the find realization provisioning services step, provisioning services that can provision the maximal provisioning groups $PG_{max}(am)$ of an application model $am \in AM$ are searched. Thus, for each maximal provisioning group $pg \in PG_{max}(am)$ a *possible provisioning*

service is searched.

Given the definition of provisioning services in Section 3.5 and a maximal provisioning group pg a possible provisioning service $ps \in PS(p)$ at a provider $p \in Prov$ is defined as a provisioning service that offers a provisionable deployment $pd = provDep(ps)$ compatible with the maximal provisioning group. Compatible means that the provisionable deployment contains a component for all components of the maximal provisioning group with the right deployment relations, the right multi-tenancy patterns and the right component types.

Definition 95 (Compatible deployments for a group of components). *Given a group of components $g \subseteq \pi_1(am)$ out of a variability model $am \in AM$, and a set of deployments $Dep \subseteq DP$ at a provider p . Then the set of deployment relations between the components of the group is $D_g \subseteq \pi_5(am) = \{d \in \pi_5(am) \mid \pi_1(d) \in g \wedge \pi_2(d) \in g\}$ the component component type map is $ct_g = \pi_6(am)$ and the component, multi-tenancy pattern map is $pt_g = \pi_11(am)$. Given these definitions, the compatible deployments are then defined as:*

$$\begin{aligned}
 DP_{compatible}(g, am, p, Dep) = & \{d \in Dep \mid \forall c_1, \dots, c_k \in g \\
 & \exists b_1, \dots, b_k \in \pi_1(d) : \\
 & ct_g(c_n) = \pi_4(d)(b_n) \wedge pt_g(c_n) = \pi_3(d)(b_n) \wedge \\
 & \forall d_1, \dots, d_l \in D_g \exists e_1, \dots, e_l \in \pi_2(d) : \\
 & \pi_1(d_m) = c_x \wedge \pi_2(d_m) = c_y \wedge \\
 & \pi_1(e_m) = b_x \wedge \pi_2(e_m) = b_y\}
 \end{aligned}$$

where $0 < n \leq k$, $0 < x \leq k$, $0 < y \leq l$ and $0 < m \leq l$. I.e. for each component c_n in the group g a corresponding component b_n exists in the deployment that has the same component type and the same multi-tenancy pattern. Additionally for each deployment relation d_m connecting two components in group g a corresponding deployment relation e_m exists in the deployment that connects the corresponding components in the deployment.

Definition 96 (Possible provisioning services). *Given the definition of the compatible deployments the provisioning services $PS_{possible}(pg, am, p)$ of a provider $p \in Prov$ that can provision a maximal provisioning group pg of an application*

model am are defined as provisioning services with a compatible provisionable deployment:

$$PS_{possible}(pg, am, p) = \{ps \in PS(p) \mid \exists pd \in provDep(ps) : pd \in DP_{compatible}(pg, p, am, provDep(ps))\}$$

The result of the *finding realization provisioning services* step is then an annotation of all maximal provisioning groups in the customer-specific application model with zero or more provisioning services that can provision the respective required maximal provisioning group.

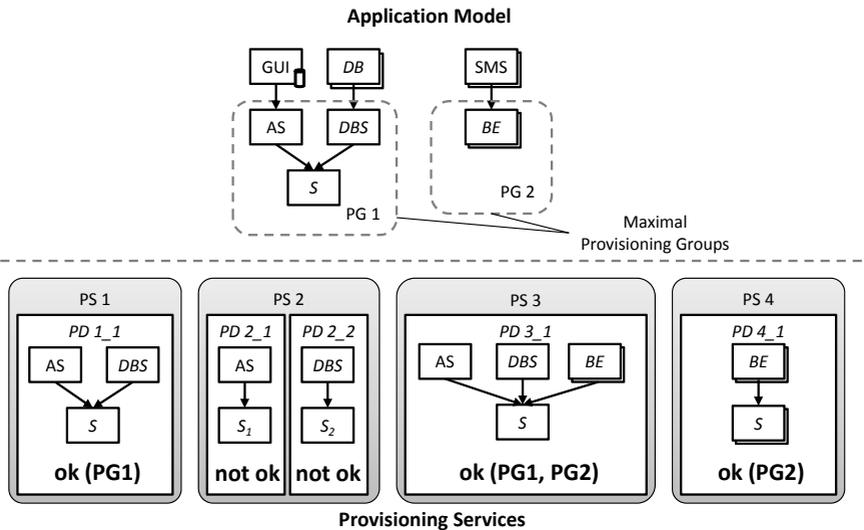


Figure 5.12: Possible provisioning services for an application model

Figure 5.12 shows the application model introduced in Figure 5.11 and a set of provisioning services ($PS 1$, $PS 2$, $PS 3$ and $PS 4$). Each of these provisioning services offers one or more provisionable deployments. For example, $PS 2$ offers deployments $PD 2_1$ and $PD 2_2$. The figure also shows which provisioning services are possible provisioning services for one of the provisioning groups because they contain compatible provisionable deployments. As shown $PD 1_1$

of *PS 1* is a compatible provisionable deployment for provisioning group *PG 1* but not for *PG 2*, as it does not contain a component of type *BPEL Engine (BE)*. *PS 2* is not a possible provisioning service, as the *AS* and *DBS* components are provisioned on two different servers S_1 and S_2 instead of on the same server. *PS 3* contains a deployment *PD 3_1* that offers all necessary components and thus is a compatible deployment for both *PG 1* and *PG 2*. The fact that there is no deployment relation in the application model that connects the *BPEL Engine (BE)* with any other of the provider supplied components, indicates that there is no restriction on how these must be connected or separated. *PD 4_1* of *PS 4* is a provisionable deployment that is compatible with *PG 2* but not with *PG 1*, as it does not contain any of the necessary components of *PG 1*.

Find Realization Deployments. Having found provisioning services for the provisioning groups it is now the task of the *find realization deployments* step to find *realization deployments* for components that can be reused. Realization deployments are already provisioned deployments that fulfill all implicit requirements annotated to components in an application model that can be reused.

When finding realization deployments two general strategies exist: the first strategy is to just find already provisioned components of the right component type and right multi-tenancy pattern for each component in the application that can be reused. This strategy would make the search for realization components rather simple, as already provisioned components could be indexed by their component type and multi-tenancy pattern and thus be easily found.

However, again as in the *find realization provisioning services* step, the maximal provisioning groups play an important role. As said above, maximal provisioning groups contain a set of components that must be provisioned together and whose deployment relations are important. A different strategy for the finding of realization components is required respecting the maximal provisioning groups. However, trying to find only the maximal provisioning groups is not sufficient when searching for realization components, as other components from the set of components that can be reused might also be already provisioned.

Thus, instead of only searching for realization components for maximal provisioning groups, realization components for maximal reuse groups must be searched.

Definition 97 (Maximal reuse groups). *The set $MG(am) = \{\dots, mg_i, \dots\}$ defines all maximal reuse groups in an application model.*

A maximal reuse group $mg(pg) \in MG(am)$ is associated to a maximal provisioning group $pg \in PG(am)$ of an application model $am \in AM$. The set of components in the maximal reuse group is defined as a set of components out of the set of components of the application model $C = \pi_1(am)$ that are reusable and are either components of the associated maximal provisioning group or components that have a deployment relation on any of the components of the maximal provisioning group.

$$mg(pg) = \{c \in C_{reuse}(am) \mid c \in pg \vee \exists d \in D : \pi_1(d) = c \wedge \pi_2(d) \in pg\}$$

Where D is defined as the set of deployment relations of the application model $D = \pi_5(am)$

Definition 98 (Complete Realization Deployments). *Given a maximal reuse group $mg \in MG(am)$ in an application model $am \in AM$ and a set of deployments $DP(p)$ at a provider p , the set of complete realization deployments $DP_{real}(mg, p)$ for a maximal reuse group is defined as the compatible deployments of that provider.*

$$DP_{completeReal}(mg, am, p) = DP_{compatible}(mg, am, p, DP(p))$$

The set of complete realization deployments is thus the set of deployments at a provider that fulfills all implicit requirements (components, deployment relations, multi-tenancy patterns and component types) of all the components in a maximal reuse group. However, given the definition of maximal reuse groups, these maximal reuse groups can also contain components that are not supplied by the provider and thus can be provisioned by deploying the files

contained in the implementation for such a component. When searching for possible realizations also those deployments can be considered that are only compatible to a subset of the components in a maximal reuse group which must always include all those components that are of implementation type *provider supplied*. These deployments are called *partial realization deployments* of a maximal reuse group.

Definition 99 (Partial Realization Deployments). *Given a maximal reuse group $mg(pg) \in MG(am)$, containing a provisioning group pg then all subsets of that maximal reuse group that can be realized by partial realization deployments are defined as the set of subsets $MGS(mg(pg)) \subseteq mg(pg)$ containing all components that are in pg , i.e. $\forall mgs \in MGS(mg(pg)) : pg \subseteq mgs$.*

Thus, given a maximal reuse group $mg \in MG(am)$ in an application model $am \in AM$ and a set of deployments $DP(p)$ at a provider p , the set of partial realization deployments is defined as:

$$DP_{\text{partialReal}}(mg, am, p) = \{pd \in DP \mid \exists mgs \in MGS(mg) : pd \in DP_{\text{compatible}}(mgs, am, p, DP(p))\}$$

Note that the set of partial realization deployments also contains the complete realization deployments.

Figure 5.13 shows the application model introduced in Figure 5.11 and a maximal reuse group *RG 1* of this model. The Figure also shows a set of provisioned deployments at a provider. *D 1_1* is a partial realization deployment of *RG 1*, as it contains realizations for all provider supplied components of *RG 1*. *D 2_1* and *D 2_2* are not realization deployments of *RG 1*, as they do not contain all necessary components. *D 3_1* is a complete realization deployment of *RG 1* as it contains realizations for all components in *RG 1*. *D 4_1* is not a partial realization deployment of *RG 1*, since it does not contain realizations for all provider supplied components in *RG 1*.

The result of the *finding possible realization deployments* step is a (possibly empty) set of already provisioned deployments for each maximal reuse group of the application model.

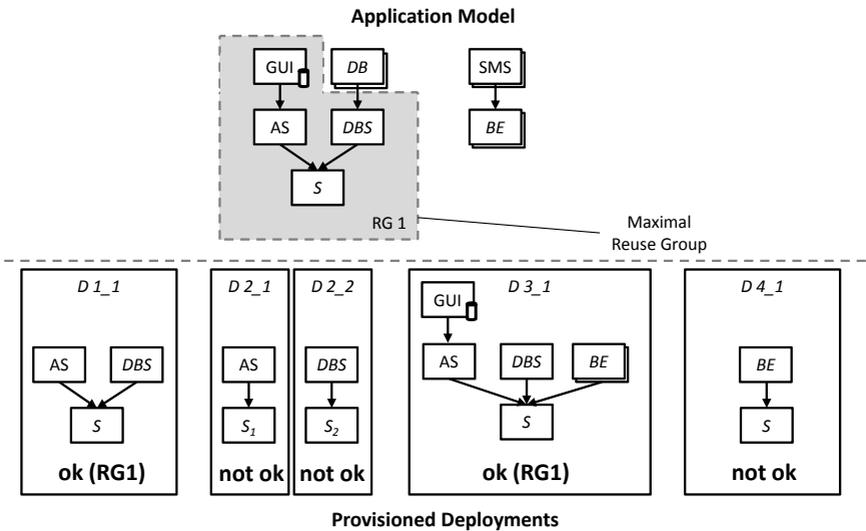


Figure 5.13: Possible (partial) realization deployments of an application model

5.3.2 Find Correct and Complete Component Bindings

The *find correct complete component bindings* subprocess is concerned with finding combinations out of provisionable components from realization provisioning services and already provisioned components out of realization deployments that together can realize all maximal provisioning groups. A component binding that can realize all maximal provisioning groups for an application solution is a component binding that allows to provision the whole application solution as it defines for each component that must be provisioned or reused which provisioning service or already provisioned component is used. All components not in a maximal provisioning group, can also be deployed because the necessary code is shipped or known where it can be retrieved, thus no component binding for those is necessary.

The component bindings that can realize all maximal provisioning groups are called *complete component bindings*. Complete component bindings are thus a special case of the component bindings as defined in Definition 3.5.2.

Several such complete component bindings can exist for an application model.

When finding complete component bindings for a customer-specific application solution $sol \in SOL$, two cases must be taken into account: In the first case no component binding has been specified during the previous phases, i.e. through template engineering, template customization or solution engineering. In this case the set of component bindings in the solution is empty, i.e. $\pi_4(sol) = \emptyset$. The provisioning infrastructure must then find a mapping of each provider-supplied component in the solution to a provisioned or provisionable component. In the second case one or multiple component bindings have been specified during the previous phases. In this case the set of component bindings in the solution is not empty and those existing component bindings must be *completed*, i.e. the provisioning infrastructure must find mappings for each component where no mapping has been specified in the component bindings.

Those two cases can be generalized in a way that the first case is treated as a special case of the second case. This can be done by simply treating the first case as a case where the set of component bindings is not empty, but one binding not containing any mappings from components to already provisioned components or provisioning services, exist. Thus the *completion* operation of a component binding can be defined as follows:

Definition 100 (Completion of a component binding). *Given an application model am containing a set of components $C = \pi_1(am)$ and a component binding $b \in CB$ as well as a provider p , then the set of complete component bindings $CB_{complete}(am, b, p)$ that can be produced for that application model, is defined as the set of component bindings $cb \in CB_{complete}(am, b, p)$, that respect the mappings expressed through the component binding b and map all components that are provider supplied and not mapped in b to an already provisioned component $c \in C_{provisioned}(p)$ or a provisionable component $cp \in C_{provisionable}(p)$. The following rules must also apply for all complete component bindings (for readability issues the rules are stated separately here and not in a set definition): Recall that a component binding is defined as a map from components in an application model to already provisioned components or provisionable components at a provider (See Definition 60).*

Rule 1: All mappings to already provisioned components in b must also be present in the completed mappings:

$$\forall c \in C : b(c) \in C_{\text{provisioned}}(p) \Rightarrow cb(c) = b(c)$$

Rule 2: All mappings to provisionable components must be present in the completed mappings or an already provisioned component derived from the provisionable component must be assigned:

$$\forall c \in C : b(c) \in C_{\text{provisionable}}(p) \Rightarrow cb(c) = b(c) \vee cb(c) \in C_{\text{provisioned}}(p) \\ \wedge \text{derivedFrom}_p(cb(c)) = b(c)$$

Rule 3: All components belonging to a maximal provisioning group must be bound to a provisioning service or provisionable component.

$$\forall c \in C : c \in pg \wedge pg \in PG_{\text{max}}(am) \Rightarrow cb(c) \neq \perp$$

Rule 4: All components belonging to the same maximal provisioning group pg where one component is bound to a provisionable component must be bound to provisionable components of the same provisionable deployment pd of a provisioning service ps .

$$\forall c, d \in C : c, d \in pg \wedge pg \in PG_{\text{max}}(am) \wedge cb(c) \in C_{\text{provisionable}}(p) \\ (\Rightarrow \exists ps \in PS(p) \exists pd \in \text{provDep}(ps) : cb(c) \in \pi_1(pd) \\ \wedge cb(d) \in \pi_1(pd))$$

Rule 5: All components belonging to a maximal reuse group $mg(pg)$ where one component is bound to an already provisioned component must be bound to provisioned components of the same deployment pd or must not be bound if they are not of implementation type provider supplied, i.e. do not belong to a maximal provisioning group.

$$\begin{aligned} \forall c, d \in C : \quad & c, d \in mg(pg) \wedge mg(pg) \in MG(am) \wedge cb(c) \in C_{provisioned}(p) \\ & (\Rightarrow \exists dep \in DP(p) : cb(c) \in \pi_1(dep) \wedge cb(d) \in \pi_1(dep) \vee \\ & (c \notin (pg) \wedge cb(c) = \perp)) \end{aligned}$$

The set $CB_{complete}(am, cp, p)$ then contains all complete component bindings for an application model am and a given partial component binding b at a provider p .

A complete component binding fulfills the implicit requirements of an application model of a customer-specific application solution. However, as customer-specific application solutions can also impose explicit requirements in terms of variability that must be bound at binding time, it is not only necessary to find a complete component binding but also a *correct and complete component binding* which is a complete component binding that also fulfills the explicit requirements.

A complete component binding of a customer-specific application solution is correct if the template customization documents of the bound components are a correct and complete customization of the set of component binding variability points of the variability model of the solution. Thus, for each component c in the application model of the solution the following must hold true: The *template customization customization document* (tc-doc) of the associated bound provisionable or already provisioned component pc is a correct and complete customization of the variability points associated with c .

In order to find all correct and complete component bindings for a customer-specific application solution all complete component bindings for the application model of the solution must be examined. The set of all complete component bindings is the n -ary cartesian product of all combinations of realization provisioning services and realization deployments found during the *find realization provisioning services* and *find realization deployments* steps, where n is the number of maximal provisioning groups in the application model.

To evaluate whether a component binding is correct and complete, a *component binding evaluation flow* is used. A component binding evaluation flow

(evaluation flow for short) works similarly to a customization flow. A customization flow if used in evaluation mode, checks whether a given customization document is a correct and complete customization of the variability model that was used to generate the customization flow. An evaluation flow checks whether one or multiple members of the set of customization documents (representing the explicit capabilities of the components in a component binding) are correct and complete customizations of the variability model that was used to generate the evaluation flow. To do so, the evaluation flow must check at each variability point if the given customization documents contain a selected value and selected alternative that is allowed to be bound given the previously bound variability points in that customization document.

The following optimization can be made to reduce the number of component bindings that must be evaluated during the search for complete and correct component bindings: Not every complete component binding is evaluated separately to check if it is correct, but *equivalence classes* are built that can be jointly evaluated.

Given the operational semantics of the Cafe variability metamodel described in Section 4.2, explicit requirements are described as component binding variability points. Explicit capabilities of a provisionable component or an already provisioned component are described as *template customization customization documents*. Thus an equivalence class can be defined as follows: Upon evaluation of a component binding variability point out of the explicit requirements of a solution, an equivalence class of component bindings contains all those component bindings that have the same selected alternatives and selected values at all previously evaluated variability points. In particular, this equivalence class contains all component bindings that have the same associated provisioning service or the same already provisioned deployment at all components targeted by the previously evaluated variability points.

Thus, at a given variability point vp_i , the set of equivalence classes for that variability point is created first by evaluating the equivalence classes for the previous variability point vp_{i-1} . Each equivalence class of the previous variability point vp_{i-1} is split into n new equivalence classes, where n is the number of different selected alternative, selected value combinations for the

variability point vp_{i-1} present in the equivalence class. Then for each new equivalence class it is checked if it is a correct customization of the variability point vp_i . If it is, the equivalence class is carried over to the next variability point vp_{i+1} , if not, it is dropped from the list of equivalence classes.

The evaluation flow thus works similar to a customization flow with the following differences: Once a variability point becomes active and is evaluated, it is checked for each enabling condition whether it evaluates to true for an equivalence class of component bindings.

Definition 101 (Component bindings equivalence relation). *The set $EqCl(s)$ contains all equivalence classes of component bindings that exist at a given step $s \in S$ during an evaluation where S is the set of steps to be performed during the evaluation (Def. 62). A component binding equivalence class $eq \in EqCl(s)$ is defined as a subset of all complete component bindings for an application model $am \in AM$, i.e., $eq \subseteq CB(am, p)$. For all component bindings in an equivalence class the following holds true: For each variability point that is already bound, the selected alternatives and selected values of the associated customizations have the same values. The associated customizations are the combined customization documents of all components included in the component binding. The function $vmcCB : CB \rightarrow VMC$ returns the combined customization of a component binding. As defined in Def. 80, $\pi_2(vmc)$ is the variability point, selected alternative map of a customization and $\pi_3(vmc)$ is the variability point, selected value map of a customization vmc .*

Let $c, d \in CB(am, p)$; The relation \approx is defined as follows:

$$c \approx d :\Leftrightarrow \forall vp \in VP_{bound}(s) : \pi_2(vmcCB(c))(vp) = \pi_2(vmcCB(d))(vp) \wedge \pi_3(vmcCB(c))(vp) = \pi_3(vmcCB(d))(vp)$$

i.e. for each previously bound variability point the same alternative and value have been selected for any given pair of component bindings in an equivalence class.

The relation \approx is reflexive. Let $c \in CB(am, p)$ then $c \approx c$. This is the case as only one alternative and value is selected for each variability point in a component

binding and thus the following holds:

$$\forall vp \in VP_{bound}(s) : \pi_2(vmcCB(c))(vp) = \pi_2(vmcCB(c))(vp) \wedge \\ \pi_3(vmcCB(c))(vp) = \pi_3(vmcCB(c))(vp)$$

The relation \approx is symmetric. Let $c, d \in CB(am, p)$ and let $c \approx d$, then $d \approx c$ must hold and thus the following must hold:

$$\forall vp \in VP_{bound}(s) : \pi_2(vmcCB(d))(vp) = \pi_2(vmcCB(c))(vp) \wedge \\ \pi_3(vmcCB(d))(vp) = \pi_3(vmcCB(c))(vp)$$

which holds as per the definition the same alternative and values have been selected at each variability point..

The relation \approx is transitive. Let $c, d, e \in CB(am, p)$ and $c \approx d$ and $d \approx e$ then $c \approx e$ and the following must hold:

$$\forall vp \in VP_{bound}(s) : \pi_2(vmcCB(c))(vp) = \pi_2(vmcCB(e))(vp) \wedge \\ \pi_3(vmcCB(c))(vp) = \pi_3(vmcCB(e))(vp)$$

as per the definition of the relation \approx : $\pi_2(vmcCB(c)) = \pi_2(vmcCB(d))$ and $\pi_2(vmcCB(d)) = \pi_2(vmcCB(e))$ and thus $\pi_2(vmcCB(c)) = \pi_2(vmcCB(e))$. In addition $\pi_3(vmcCB(c))(vp) = \pi_3(vmcCB(d))(vp)$ and $\pi_3(vmcCB(d))(vp) = \pi_3(vmcCB(e))(vp)$ and thus $\pi_3(vmcCB(c))(vp) = \pi_3(vmcCB(e))(vp)$

Thus \approx is an equivalence relation as it is reflexive, symmetric and transitive.

The set $EqCl(s)$ is the set of \approx equivalence classes for all variability points bound before a given step s .

Algorithm 5.1 shows the algorithm performed by the evaluation flow. Before the algorithm is started the states of all variability points is set to inactive. The algorithm is similar to that for the customization flows as shown in Algorithm 4.1. Part 1 and Part 2, i.e., the activation and deactivation of variability points is the same. However, Part 3 and Part 4 of the algorithm shown in Algorithm 4.1 are replaced by the Part 3 as shown in Algorithm 5.1.

In Part 3 for each active variability point vp all equivalence classes are built. This is done in the following way: In Part 3a all equivalence classes that existed

at the previous step $eq \in EqCl(s - 1)$ are investigated. For each component binding $cb \in eq$ it is evaluated whether an equivalence class eqn already exists at the given step that contains a component binding cbn with the same selected values and alternatives at that variability point. If yes, the component binding is added to that equivalence class. If not, a new equivalence class is built.

In Part 3b of algorithm 5.1 it is then evaluated for all equivalence classes if they represent component bindings that contain a selected alternative and selected value that are valid given the associated enabling condition at the given step, if yes, this equivalence class is considered in the next step, if not it is dropped.

The algorithm terminates once all variability points are in state disabled or bound. The remaining equivalence classes at the last step s contain all possible provisioning service component bindings that are possible realization combinations. The result of the evaluation flow is thus a set of component bindings $CB_{correctComplete}(am, p)$ that represents all combinations of provisionable components of different provisioning services and already provisioned deployments at provider $p \in Prov$ that can produce a correct and complete component binding for the application model $am \in AM$.

Algorithm 5.1 describes the algorithm how the evaluation flow works. The concrete description of the generation of customization flows from the variability model of an application template is given in Section 4.3. The generation of an evaluation flow is similar, as an evaluation flow is a special customization flow. Like for customization flows for each variability point a set of activities is generated. These activities are connected via control connectors that represent the dependencies between the variability points as described in 4.3. The evaluation flow differs in the activities performed after the variability point start activity $vs_{v,p}$ of a variability point has been executed. In the normal customization flow that must only deal with one instance of the variability model only one enabling condition becomes active, namely the one that first evaluates to true given the customization of the previous variability points in that variability model. In the evaluation flow dealing with multiple instances of the variability model, namely one for each equivalence class, multiple enabling conditions can become active, one for each equivalence class as shown in Algorithm 5.1.

Algorithm 5.1 Navigation logic of an evaluation flow

```
1:  $s = 0$ 
2:  $eq = CB(am, p)$ 
3:  $EqCl(-1) = \{eq\}$ 
4: while  $s < |VP_i|$  do
    5:
    6: for all  $vp \in VP_{active}$  do
    7:    $EqCl(s) = \emptyset$ 
    8:
    9:   for all  $eq \in EqCl(s - 1)$  do
    10:      $EqCl(s)_{temp} = \emptyset$ 
    11:     for all  $cb \in eq$  do
    12:       if  $\exists eqn \in EqCl_{temp}(s) \exists cbn \in eqn : selAlt(vp, vmcCB(cb)) =$ 
    13:          $selAlt(vp, vmcCB(cbn)) \wedge selVal(vp, vmcCB(cb)) =$ 
    14:          $= selVal(vp, vmcCB(cbn))$  then
    15:            $eqn = eqn \cup \{cb\}$ 
    16:         else
    17:            $newEq = \{cb\}$ 
    18:            $EqCl_{temp}(s) = EqCl_{temp}(s) \cup \{newEq\}$ 
    19:         end if
    20:       end for
    21:        $EqCl(s) = EqCl(s) \cup EqCl(s)_{temp}$ 
    22:     end for
    23:
    24:     for all  $eq \in EqCl(s)$  do
    25:       if  $\neg(selAlt(vp, vmcCB(eq)) \in enAltS(vp, s, vmcCB(eq)) \vee$ 
    26:          $selVal(vp, vmcCB(eq)) \in$ 
    27:          $possVal(selAlt(vp, vmcCB(eq)), vmcCB(eq)))$  then
    28:          $EqCl(s) = EqCl(s) \setminus eq$ 
    29:       end if
    30:     end for
    31:      $s = s + 1$ 
    32:   end for
    33: end while
34:
35:  $CB_{correctComplete}(am, p) = \{cb \in CB(am, p) \mid \exists eq \in EqCl(s) : cb \in eq\}$ 
```

▷ Part 1 and Part 2 as in Algorithm 4.1

▷ Part 3

▷ Part 3a

▷ Part 3b

Therefore, the evaluation of the individual enabling conditions and the subsequent activities are nested in a loop that iterates over each equivalence class for that variability point as shown in Figure 5.14. Before looping over all equivalence classes, the equivalence classes must be built for the given variability point as shown in Figure 5.14. This is done by the newly introduced built equivalence classes activity. Inside the loop activity the same activities as for the customization flow are inserted. These activities, as described in detail in Section 4.3.2, then check whether the customizations for that equivalence class are correct or perform necessary customizations. If the customization is not correct, the equivalence class is removed from the list of equivalence classes.

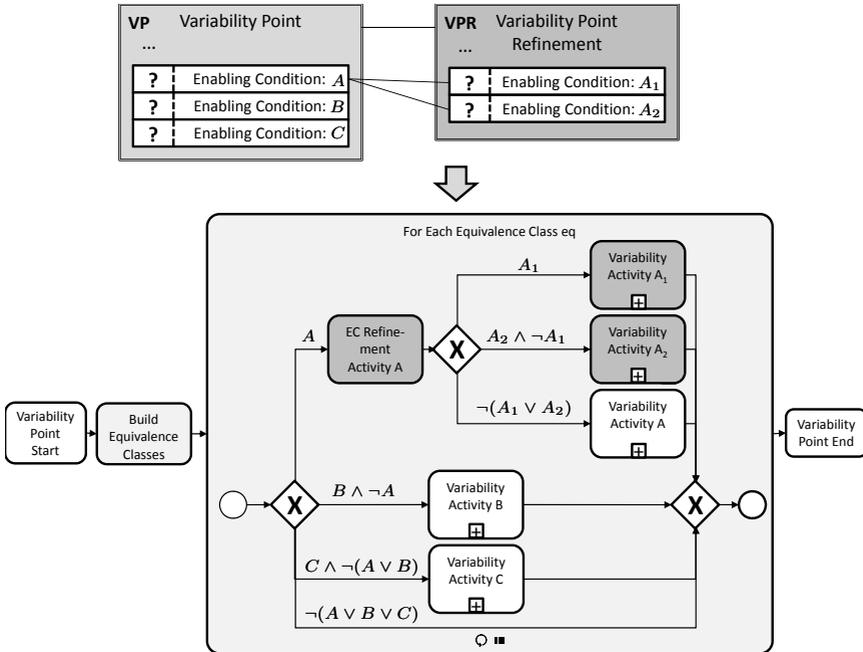


Figure 5.14: Activities for the evaluation of multiple equivalence classes in the evaluation flow

The output of the evaluation flow is then a set of equivalence classes each

containing a set of correct and complete components bindings.

5.3.3 Select the Realization Component Binding

Having produced a set of correct and complete component bindings in the previous subprocess, the *select realization component binding* subprocess (cf. Figure 5.10) is now concerned with the selection of one particular component binding out of the set of correct and complete component bindings that will be used for provisioning. This selected binding is called *realization component binding*. The mappings of the realization component binding are used to bind the components in the application model of a solution to the provisionable components and already provisioned components that are used to realize them. A straightforward solution for the selection would be to take the cheapest component binding. In Section 5.4 the optimization of component bindings is treated in more detail. To describe how the actual provisioning of a customer-specific application solution works, it is assumed that one correct and complete component binding has been picked for the application solution. In the Cafe prototype one such component binding is chosen at random.

Independent of the strategy of choosing a concrete realization component binding out of the set of correct and complete component bindings, the result is always the same: For each component in the application solution that must be provisioned or can be reused a concrete realization component or realization provisioning service is chosen depending on the mapping defined in the realization component binding.

Definition 102 (Realization component binding for a customer-specific application solution). *The function $rcb : SOL \rightarrow CB$ assigns a component binding to a customer-specific application solution that is used to realize the components in the application model of the solution. The assigned component binding must be in the set of correct and complete component bindings for the solution or no realization component binding may be assigned. Thus given a provider $p \forall s \in SOL : rcb(s) \in CB_{correctComplete}(\pi_1(s), p) \cup \{\perp\}$ where \perp denotes that no realization component binding has been assigned, and $\pi_1(s)$ is the application model of the solution.*

As a shortcut the function $real : C \rightarrow C_{provisioned}(p) \cup C_{provisionable}(p) \cup \perp$ returns for a given component c in the application model of the solution s the bound component. Thus $\forall c \in \pi_1(\pi_1(s)) : real(c) = rcb(s)(c)$

The already provisioned component or provisionable component and provisioning service associated to a component via the component binding is then used in the subsequent steps as the realization for that component.

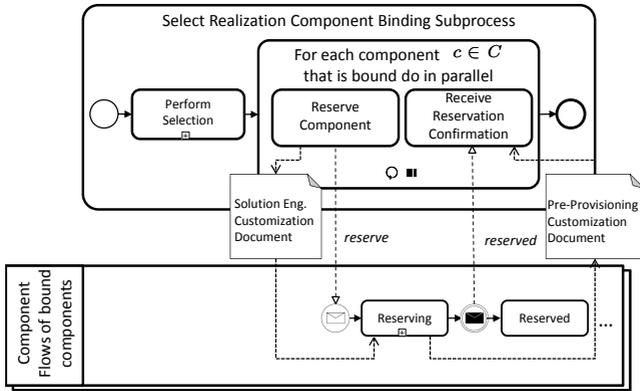


Figure 5.15: Select realization component binding subprocess, component flow interaction

Figure 5.15 shows the *select realization component binding* subprocess and its interaction with the component flows of the components that have been bound. This subprocess is part of the overall provisioning process as shown in Figure 5.10. Having performed the selection of the realization component binding in the *perform selection* step, a **reserve** message is sent to every component flow representing the respective bound component. Along with the **reserve** message the solution engineering customization document is passed so that the component flow can configure the component. Having configured the component, the component flow returns a **reserved** message including the pre-provisioning customization document that contains the already bound variability points for the pre-provisioning phase. The provisioning flow must then ensure that all variability points in the pre-provisioning customization document that have not been bound yet, are bound.

The *select realization component binding* subprocess can additionally be performed in a loop for all correct and complete component bindings, in case the reservation cannot be completed for all components of a component binding. In this case the next correct and complete component binding can be tried.

Figure 5.15 only shows the case where a component is bound to a provisioning service and thus the `reserve` message must be sent to reserve the new component. In case it is bound to an already provisioned component a `reserve subscription` message must be sent or a `reserve deployment` message, in case the component must be deployed.

5.3.4 Component Provisioning

Having performed the selection of the realization component binding and the reservation of all bound components, the next step in the provisioning of a customer-specific application-solution is to execute the necessary steps to set up the bound components. However, as the components have dependencies among each other, the ordering in which the execution of the provisioning of the individual components takes place is important. The order is determined by two types of dependencies as shown in Figure 5.16. The first type of dependencies are the deployment dependencies, i.e. components can only be set up if the components on which they are deployed (their required components) are provisioned such as components *A* and *B* in Figure 5.16. The second type of dependency to be considered are dependencies induced by the variability model relevant for provisioning. These are those dependencies where one variability point for a first component must be bound at pre-provisioning time but depends on a variability point of a second component that is bound at running time, thus the first component must be provisioned after the second component as shown for components *B* and *C* in Figure 5.16. Components that do not have dependencies among each other (such as component *A* and *C*) can be provisioned in parallel.

Definition 103 (Provisioning Order). *Given two components c and d the provisioning order is defined as follows:*

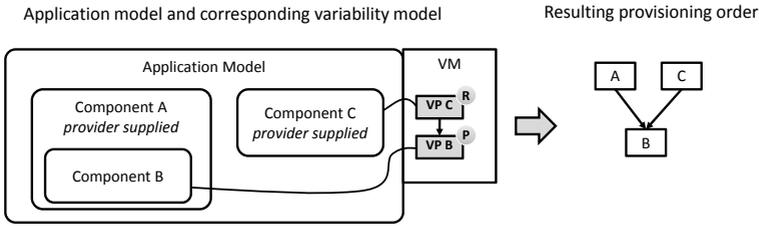


Figure 5.16: Ordering relevant dependencies and resulting provisioning order example

- $c >_{po} d$ indicates that c is provisioned after d
- $c =_{po} d$ indicates that c can be provisioned in parallel to d

Finding the Right Order for Provisioning. While determining the right order the following rules must be taken into account.

Ordering Rule 1 (Components can only be provisioned if their required components are available). *This rule is fairly easy to enforce, as it simply states that the provisioning must start at the sinks of the deployment graph of the application model of the solution and then move upwards following the deployment relations. Thus, given deployment graph $G = (C, D)$ out of the application model of a customer-specific application solution a component c must always be provisioned after its transitively required components $C^+(c)$ (Definition 13).*

$$\forall c_1, c_2 \in C : \begin{aligned} & c_2 \in C^+(c_1) \\ & \Rightarrow c_1 >_{po} c_2 \end{aligned}$$

However, the deployment relations are not the only dependencies between components that must be taken into account when finding the right order for provisioning. In addition to the deployment relations, the dependencies induced by the combination of variability points of different components must be taken into account. For dependencies induced by variability points see Section 3.4.6 and especially Definition 55. In particular, the dependencies to be considered are those where one component must be provisioned so

that the value of a variability point that is bound at runtime can be used to bind a variability point of another component that must be bound during the pre-provisioning phase.

Ordering Rule 2 (Components can only be provisioned if all provisioning-related variability points have been bound). *Some components must be configured before they are provisioned. These are all components having an associated variability point that requires binding before their provisioning and that has not been bound during the customer-related customization. Given a component graph of a customer-specific solution $G = (C, D)$, the set of components C_{CCP} that require pre-provisioning configuration depending on the binding of a variability point at another component can be defined as the set of components that have a non-empty set of components they depend on during the pre-provisioning phase.*

$$C_{CCP} = \{c \in C \mid C_{prov}^+(c) \neq \emptyset\}$$

Thus for all of those components the provisioning order must be greater than that of the components they depend on.

$$\forall c_1 \in C_{CCP}, \forall c_2 \in C : c_2 \in C_{prov}^+(c_1) \Rightarrow c_1 >_{po} c_2$$

The set C_{prov}^+ is defined as the set of components targeted by a variability point that must be evaluated before the provisioning takes place. Variability point dependencies on variability points that can be bound after the provisioning or have already been bound during customization do not have any influence on the order of provisioning of the components. These variability points have either been already bound or can be bound after the component is deployed. How this is done is shown in Section 3.4.6.

Ordering Rule 3 (Components that do not have any dependencies can be provisioned in parallel). *To speed up the provisioning, the provisioning of components*

that do not have dependencies can be done in parallel.

$$\begin{aligned} \forall c_1, c_2 \in C : \quad & c_2 \notin C^+(c_1) \wedge c_1 \notin C^+(c_2) \wedge \\ & c_2 \notin C_{prov}^+(c_1) \wedge c_1 \notin C_{prov}^+(c_2) \\ \Rightarrow & c_1 =_{po} c_2 \end{aligned}$$

Considering the example shown in Figure 5.11, the server as well as the application server, containing DBMS and DB, can be provisioned in parallel to the workflow component and the underlying ODE BPEL engine, while the GUI component must be provisioned after the workflow component has been provisioned.

To ensure these three ordering rules the provisioning algorithm must take the deployment graph as well as the provisioning-time dependency graph into account. This combined graph is called the *combined provisioning dependency graph*.

Definition 104 (Combined provisioning dependency graph). *The combined provisioning order graph*

$$G_{cpd} = (C_{prov}, Pd)$$

is a directed, graph consisting of the components (as nodes) that must be provisioned and the provisioning dependencies (as edges). The set of provisioning dependencies Pd is defined as the union of the set of deployment relations and the set of dependencies between components induced by the variability model at provisioning time, i.e. $Pd = D \cup Cd_{prov}$. Only those dependencies are considered that are relevant for the provisioning order, i.e. those where the binding of a variability point at one component can only be done after another component is already provisioned.

The combined provisioning dependency graph is an acyclic graph. This is the case as the deployment graph must be acyclic (Def. 11) and there are no cycles allowed between dependencies induced by the variability model at provisioning time (Constraint 7). In addition, dependencies induced by the variability model at provisioning time between variability point a and b are

only allowed if a does not have a deployment relationship on b (Constraint 8).

Lemma 1 (Acyclicity of the combined provisioning dependency graph). *The combined provisioning dependency graph is acyclic.*

Acyclicity of the combined provisioning dependency graph. The subgraph of the combined provisioning order graph spanned by the components C_{prov} and the deployment relations D is acyclic (Def. 11).

Considering the definition of the set of dependencies induced by the variability model at provisioning time Cd_{prov} (Def. 55) there is one dependency between each two components that are (transitively) connected via a variability point dependency at provisioning time. Thus, it is sufficient to show that for an arbitrary path p in the subgraph of the combined provisioning order graph spanned by the components and deployment relations, there exists no edge in the set of component dependencies that closes the path.

Given such an arbitrary path $p = (sc_0, tc_0) \dots (sc_{m-1}, tc_{m-1})$ where $m \in \mathbb{N}_0$ and $tc_{i-1} = sc_i$ for all $i \in \{1, \dots, m-1\}$ there must not exist an edge $(s, t) \in Cd_{prov}$ that closes the path, i.e. where $s = tc_{m-1}$ and $t = sc_0$ and thus introduces a circle.

This is ensured by constraint 8 which disallows dependencies at provisioning time between a source component s and a target component t if t is in the set of transitively required components of s , i.e. if $t \in C^+(s)$. Given the definition of the transitively required components of s (Def. 13) as the set of components from which a path exists to s spanned by deployment relations, sc_0 would be in the transitively required components of t and thus introducing a dependency at provisioning time would be disallowed by constraint 8. Thus due to constraint 8 it is not possible to add dependencies that introduce cycles in the deployment graph. \square

The combined provisioning dependency graph describes which component depends on which other component either because of a dependency or a deployment relation. When generalizing the ordering rules above, it can be said that a component can only be provisioned if all components on which it has a provisioning dependency, have already been provisioned. When traversing

the graph to provision the components, a provisioning algorithm must simply traverse the dependency graph in the reverse order.

Definition 105 (Provisioning order graph). *The provisioning order graph is then defined as a graph $POG = (C_{prov}, Pd^{-1})$ with the node set C_{prov} and the edge set Pd^{-1} . Where the edge set is defined as follows:*

$$(c, d) \in Pd^{-1} : \Leftrightarrow (d, c) \in Pd$$

The semantics of this provisioning order graph is similar to a workflow that must be executed to deploy the application solution. Each component in the provisioning order graph can be seen as a set of activities that must be called to provision the respective component. Thus the Cafe provisioning infrastructure transforms the provisioning order graph into a workflow that can then be executed by a workflow engine.

Provisioning Order Graph and Provisioning Flows. As said above, the provisioning order graph can be seen as a workflow which contains all steps in the right ordering to provision a customer-specific application solution. Such a workflow is called *provisioning flow* of an application solution.

Two general possibilities to generate such a flow exist as shown in Figure 5.17: The first possibility is to generate a specific provisioning flow process model from the provisioning order graph of a customer-specific application solution. This means that in case the application model of an application is affected by variability, for each customer-specific application solution a new provisioning flow is generated that is then executed. The second possibility is to execute an instance of a generic provisioning flow that interprets the provisioning order graph customer-specific application solution.

In the following the abstract actions that must be performed in such a provisioning flow regardless of whether it is a generic or a specific one are shown. In Section 6.2.10 it is shown how a generic provisioning flow is used in form of a WS-BPEL workflow in the Cafe prototype.

The provisioning order graph determines in which order the necessary actions must be executed. The nodes of the provisioning order graph represent

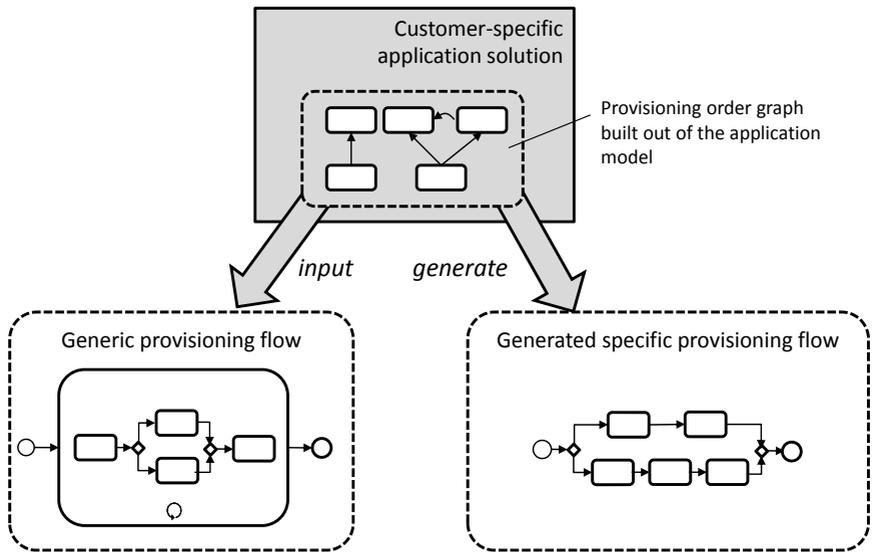


Figure 5.17: Two possibilities to derive provisioning flows

the *provisioning activities*, used to set up the individual components for a customer-specific application solution. The directed edges between components in the provisioning order graph can be seen as control connectors. Once the workflow starts, all components that are sources of the graph are provisioned in parallel. Once a component has been provisioned, the outgoing control connector is evaluated to true. In case multiple control connectors (that represent edges in the provisioning order graph) leave a component, the semantics of those are the semantics of an and-split (or parallel-split) in workflow terminology [VDATHKB03]. Thus all components connected to these control connectors are provisioned in parallel. In case several incoming control connectors exist at a component, these are treated as if they were and-joins in a workflow. Thus the component is only provisioned if all components before it were provisioned successfully.

As the edge set of the combined provisioning dependency graph contains all deployment relations as well as the component dependencies, the work-

flow takes into account all dependencies between components formulated in ordering rule 1 and 2. Additionally, by interpreting outgoing edges from a component as a parallel split in a workflow the workflow engine will ensure that components not having (transitive) dependencies are provisioned in parallel (ordering rule 3).

Figure 5.18 shows a sample combined provisioning dependency graph. By reverting the direction of the edges, the provisioning order graph is derived from the combined provisioning dependency graph as shown in Figure 5.18. In the last step the provisioning order graph is transformed into a process model as shown in the bottom part of Figure 5.18. As can be seen in Figure 5.18, the resulting provisioning flow starts with the parallel provisioning of the *Server (S)* and *BPEL Engine (BE)*. The *Application Server (AS)* and *Database Management System (DBS)* can then be provisioned in parallel after the *Server (S)* has been provisioned. The *Database (DB)* is provisioned after the *DBMS (DBS)* has been provisioned. After the *Database (DB)*, the *SMS Workflow (SMS)* and the *Application Server (AS)* have been provisioned, the *GUI (GUI)* component can be provisioned.

Algorithm 5.2 summarizes the generation of the provisioning flow from the provisioning order graph. For each component corresponding provisioning activities as shown in Section 5.3.5 are generated. These are connected via control connectors that correspond to the edges of the provisioning order graph, i.e. the reversed edges of the combined provisioning dependency graph.

Algorithm 5.2 High-level provisioning flow generation

```

1: for all  $c \in C_{prov}$  do
2:   create provisioning activity  $pa_c$ 
3: end for
4: for all  $d \in Pd^{-1}$  do
5:   create control connector  $e = (pa_{\pi_1(d)}, pa_{\pi_2(d)}, true)$ 
6: end for

```

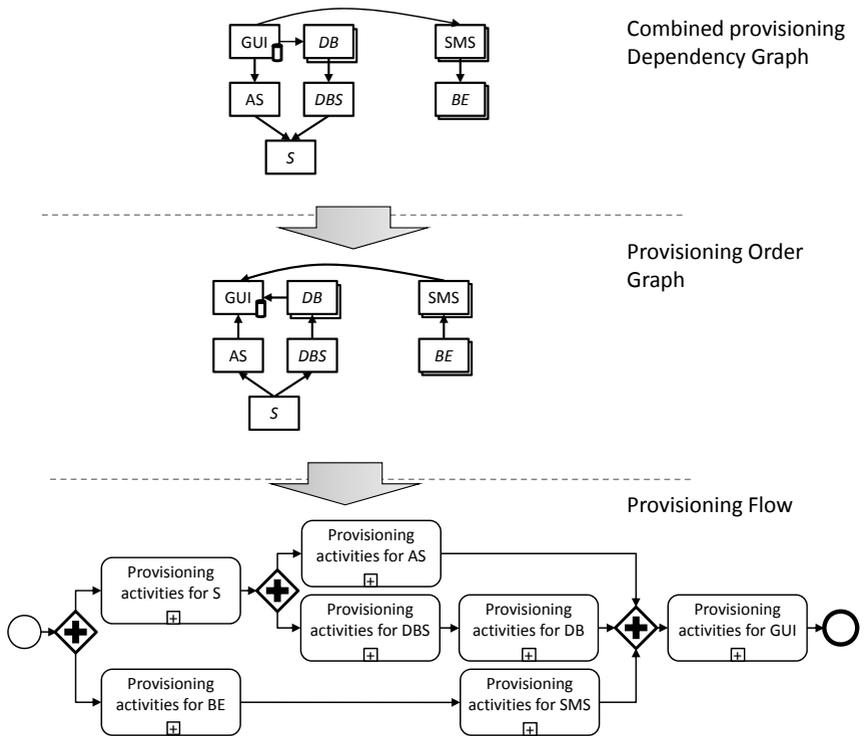


Figure 5.18: Example combined provisioning dependency graph, provisioning order graph and derived provisioning flow

5.3.5 Executing Provisioning Activities

The provisioning flow as shown in Section 5.3.4 executes the provisioning actions that make use of the CPMI interfaces (see Section 5.2) of the individual components that must be provisioned.

As shown in Figure 5.18, the provisioning flow contains a *provisioning activities* subprocess for each component. The actual activities executed in the *provisioning activities* subprocess depend on the realization component binding, i.e., whether this component will be realized by an already provisioned component or a realization provisioning service. The activities to be performed

in the *provisioning activities* subprocess also depend on the multi-tenancy pattern of the component, i.e., whether it is shared with other customers or not. Another important task of the *provisioning activities* subprocess is to do the actual pre-provisioning and runtime configuration of a component. Figure 5.19 shows the generic activities in the *provisioning activities* subprocess for a given component c out of the application model of a solution. The subprocesses shown in Figure 5.19 are explained in detail below.

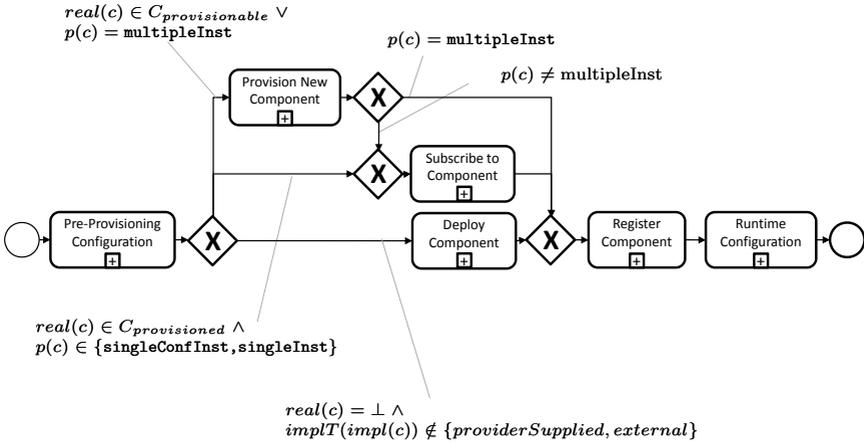


Figure 5.19: Activities of the provisioning activities subprocess

Pre-Provisioning Configuration. As shown in Figure 5.19, the first part of the *provisioning activities* subprocess is the pre-provisioning configuration. In this subprocess the binding of the pre-provisioning variability points of the component c is triggered. Since the provisioning flow follows the ordering rule 2 for the ordering of components during the provisioning, even variability points that are dependent on a variability point of another component that only gets bound at runtime can now be bound.

In the default case all variability points that must be bound during the pre-provisioning are automatic variability points not requiring any human intervention because they only have one associated alternative that can be

selected automatically. However, given the Cafe variability metamodel introduced in Section 3.4, the variability points that are used to configure a component at pre-provisioning time can also be human variability points, i.e. they can require human interaction, for example, by an administrator. To perform the pre-provisioning customization for the component, the provisioning flow calls the corresponding instance of the provisioning customization flow to request the customization document for the component *c* that must be provisioned. This is done by sending a `customizeComponent` message to the customization flow as shown in Figure 5.20. This message contains the pre-provisioning customization document for the component as received from the component flow during the component binding (see Section 5.3.3). The provisioning customization flow then binds the open variability points and returns the completely bound pre-provisioning customization document as shown in Figure 5.20.

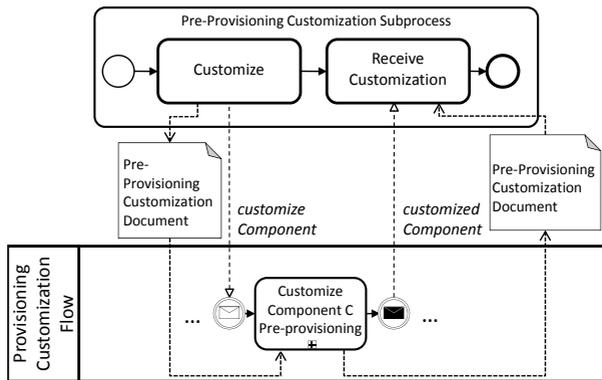


Figure 5.20: Interaction between the pre-provisioning subprocess and the customization flow

Once the customization flow has returned the pre-provisioning customization document for the component *c* via the respective `customizedComponent` message, the provisioning flow can continue.

Depending on the multi-tenancy pattern of the component and whether the component is realized by an already provisioned component or by a realization

provisioning service or if it is an internal component, a different path in the workflow is taken. The different paths that are possible after the pre-provisioning subprocess are shown in Figure 5.19. The details of the different paths are described in detail below:

Provision New Component Subprocess. This subprocess is executed in case the component c to be provisioned at a provider $p \in Prov$ is to be provisioned in the multiple instances pattern (i.e. a new instance of the component must be deployed for the tenant). Additionally, if no existing component can be reused for a component that must be provisioned in one of the other multi-tenancy patterns, this subprocess must also be executed in order to provision a suitable component. Thus, formally this subprocess is executed if the following transition condition holds true: $real(c) \in C_{provisionable}(p) \vee p(c) = multipleInst$, as shown in Figure 5.19.

When a new component must be provisioned, the component has already been reserved in the *select realization component binding* subprocess (see Section 5.3.3). As a result, the component handle is already known as part of the EPR of the component flow for this component. To finally provision the new component, the provision component subprocess must now send the `provision` message to the component flow of the reserved component as shown in Figure 5.21. This message also contains the pre-provisioning configuration that is captured in the pre-provisioning customization document.

The component flow eventually sends a `provisioned` message to indicate that the provisioning of the component is done. This message also includes the variability points that have already been bound by the provisioning environment when the component entered the running phase in the running customization document as shown in Figure 5.21.

Subscribe to Component Subprocess. This subprocess is executed in case the component c that must be provisioned is to be provisioned in the single instance or single configurable instance pattern. If it is not realized by an already existing component this component must be provisioned first, therefore in this case the *provision new component* subprocess is executed before the *subscribe to component* subprocess as shown in Figure 5.22. The *subscribe to*

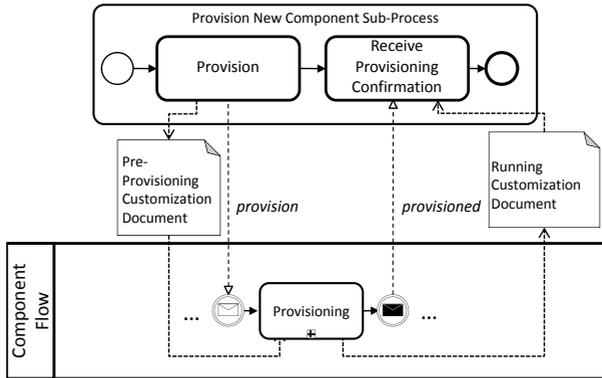


Figure 5.21: Interactions between the provision new component subprocess and the component flow for provisioning

component subprocess is executed directly if the following transition condition as shown in Figure 5.19 evaluates to true: $real(c) \in C_{provisioned} \wedge p(c) \in \{singleInst, singleConfInst\}$. Otherwise, it is executed after the *provision new component* subprocess if $p(c) \neq multipleInst$ is true.

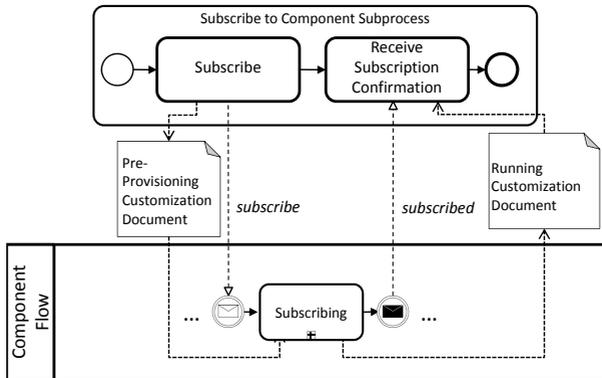


Figure 5.22: Interactions between the subscribe to component subprocess and the component flow for provisioning

The subscribe to component subprocess as shown in Figure 5.22 first con-

structs a `subscribe` message. The message contains the `subscription-id` obtained during the reservation that identifies the subscription of the customer on the already provisioned component. In case such a component has just been provisioned in the *provision new component* subprocess, the `subscribe` to component subprocess must first send a `reserveSubscription` message which will eventually be answered with a `reservedSubscription` message that contains the id that identifies this customer on the newly provisioned component (not shown in Figure 5.22). The `subscribe` message is then sent to the EPR that references the realization component that has been bound to component *c* in the component binding phase or during the *provision new component* subprocess.

Deploy Component Subprocess. The *deploy new component subprocess* is executed in case a component has an associated implementation that is not of type `provider supplied` or `external`, and no already provisioned component or existing provisioning service are annotated as a realization. Thus the *deploy component* subprocess is executed if the following transition condition as shown in Figure 5.19 evaluates to true: $real(c) = \perp \wedge implT(impl(c)) \notin \{provider\ supplied, external\}$. In this case the provisioning flow must call the *provision(deployment-id,pc-doc)* operation on the provider component with the deployment-id of the deployment that has been reserved during the binding step.

Registering a Component. Once a component has been provisioned, configured or deployed, the provisioning flow registers the component with the provisioning services and provisioned components registry. This is necessary, as the component might be reusable for other customers and can be found during the find realizations step of the provisioning. In case a component is newly provisioned by a provisioning service, i.e. the provisioning service turned a provisionable component into an already provisioned component, this component must be registered as a new already provisioned component. The registration also serves documentation purposes so that for every component running in the system it is clear which customers use the component. This is, for example, needed when customers are de-provisioned to determine whether

a component can be de-provisioned or whether it is still needed for other customers. Large IT-shops often employ a CMDB [MSB⁺07] to capture such informations. Thus, the newly provisioned component could be entered as a *configuration item* in such a CMDB.

Runtime Configuration. Having provisioned and registered a component this component is now fully operational. In particular, its variability points that are bound at runtime are available or can be bound. As shown in Figures 5.21 and 5.22 after the provisioning and configuration of a component the respective component flow returns a running customization document (rc-doc). In this rc-doc some of the variability points are already bound. It is now the responsibility of the runtime configuration subprocess to ensure that the remaining open running variability points of the component are bound. Therefore, as shown in Figure 5.23, the runtime configuration subprocess sends a `customizeComponent` message along with the rc-doc to the provisioning customization flow. The provisioning customization flow then binds all remaining open variability points. In addition to that, the variability points of other components depending on running variability points of the component can be bound now, too. The provisioning customization flow then returns the running customization document that contains all bound running variability points for the component to the runtime customization subprocess via a `customizedComponent` message. The runtime customization subprocess then sends a `setProperty` message to the component flow that contains the rc-doc to configure the component. Once this configuration has been done, the component flow returns with a `propertiesSet` message and the runtime configuration subprocess can terminate. The overall provisioning flow can then continue with the execution of the provisioning actions for the next component in the application model.

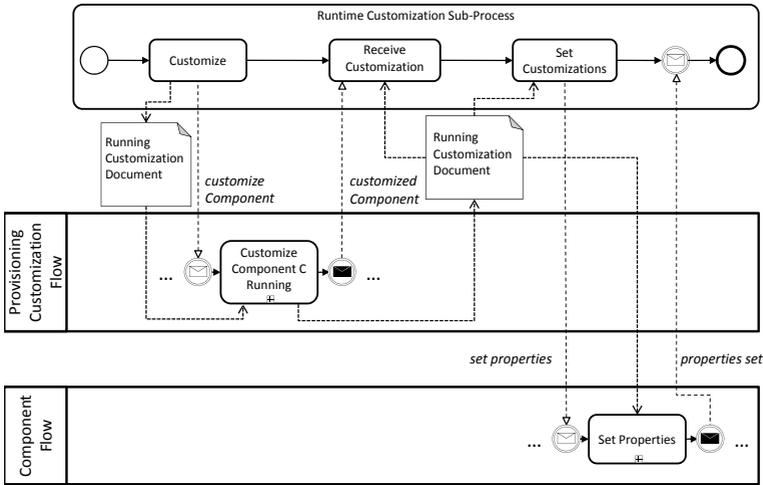


Figure 5.23: Interactions between the runtime configuration subprocess, the customization flow and the component flow for runtime customization

5.4 Optimization of Component Binding

In Section 5.3.3 the complete and correct component bindings have been introduced to describe which combinations of provisionable components of provisioning services or already provisioned components can be used to set up a customer-specific application solution. As already indicated in Section 5.3.3, multiple complete and correct component bindings can exist that fulfill the implicit and explicit requirements of the customer-specific application solution. These different combinations may differ in costs, may have different performance or may result in different utilization rates for already provisioned components, for example. Thus, a provider is interested to find an optimal component binding that is optimal to a specific property, such as cost, utilization rates or performance. The *select realization component binding* subprocess described in Section 5.3.3 must return this optimal component binding.

It is out of scope of this thesis to evaluate and define concrete optimization

algorithms for Cafe, however, the general framework to develop such algorithms for Cafe is presented here. One naive, straightforward algorithm is implemented in the prototype. This approach takes the first found combination of components and provisioning services that fulfills all implicit and explicit requirements of a customer-specific application solution.

5.4.1 Generic Plugin Mechanism for Optimization Algorithms in Cafe

In the following the generic plugin mechanism is described that allows providers to plugin in any existing or new optimization algorithm. For example, in [Feh09, FLM10] several algorithms are investigated how providers can distribute the workload of components of a Cafe application over different resources. In [DKL09] different optimizations for the optimal stratification of transactions are investigated. These algorithms can also be adapted to solve the presented problem of distributing an application model over different provisionable and already provisioned components.

Any optimization algorithm that can be plugged in the generic infrastructure must take the following information as input:

- A set of application models of customer-specific application solutions. These are the customer-specific application solutions that should be included in the optimization. In case only a new customer must be provisioned, only the application model of this customer-specific application solution can be included. In case a re-organisation of existing deployments should be undertaken, multiple application models can be taken as input. The application models include the requirements on the provider supplied components in form of the binding variability points.
- A set of already provisioned components and provisionable components attached to provisioning services. There is a price attached to each provisioned component and provisionable component, as well as their capabilities expressed through their template customization customization documents (tc-docs).

The optimization algorithm must produce the following results:

- A mapping of each provider supplied component in the customer-specific application solutions to an already provisioned component or provisioning service in form of a realization component binding that is associated to each customer-specific application solution via the *rcb* function.

5.4.2 Annotating Already Provisioned Components and Provisioning Services with Cost and Performance Levels

To be able to optimize the distribution of customers to components based on cost or performance, each provisionable and already provisioned component must be annotated with the costs of using it and with its performance. Annotating components with cost is fairly simple for components supplied by IaaS providers where resources are billed per usage. Amazon EC2, for example, bills customers based on a concept called *instance-hour*. One instance-hour is an hour in which an instance of a given amazon machine image is actually run. In [CA07] a generic pricing model for information services is introduced that is based on a concept called *computational elements* (CE). A CE can be, for example, a CPU or a bundle of resource such as multiple CPUs, memory, network traffic and storage. These CEs are then abstractions of concrete resources. Prices for comparable CEs at different providers can be defined using a *pricing scheme* (Q, T, C, U) which assigns a price to a CE that is valid for a specific quantity (Q), time (T), quality class (C) and user profile (U) [CA07]. Components in the own data center can also be annotated with costs. These costs are computed from the costs of buying the necessary hardware and software, for running the resource and for the maintenance of the resource.

Annotation with performance figures such as throughput, number of concurrent users etc. can be done using various techniques whose evaluation is out of scope of this thesis. However, some techniques are briefly listed below:

Capacity planning and performance modeling. Several methods and models exist to compute the capacity of specific resources based on a request mix and the underlying infrastructure. In [MA01] capacity planning models for Web services are introduced that can be used, for example, to estimate the amount of concurrent users and the response rates for a Web application. In

[MAFM99, AAY97] a methodology for the characterization of workload on Web sites and the characterization of behavior of Web servers is introduced. Complex component interconnections in software architectures and their influence on performance are investigated in [GM00]. In [GM00] and in [Kah01], performance models to predict the performance of a distributed application are generated from annotated UML models of that application. In [LFG04] a model is introduced to predict the performance of distributed middleware-based software. The dynamic provisioning of agile multi-tier applications is investigated in [USC⁺08], whereas techniques for optimal resource-aware deployment of distributed component-based applications are investigated in [KK04]. In addition to these approaches, that could be integrated into optimization algorithms for the component binding in Cafe, manufacturers of middleware and software products ship capacity planning tools and documentation for their components that illustrate how the performance of their components can be tuned and predicted [R⁺05, Sch06].

Scalability testing. For each component that is to be offered by a provider the provider can perform scalability testing to determine how many concurrent users a component can handle while guaranteeing certain response times. Therefore, a suitable request mix must be used resembling the utilization rate that is later produced by the application. There exist several approaches from capacity planning [AKR01, SS05, ES06] on how to create suitable request mixes for various application types.

Experience. Since providers aim to host more than one customer per application, they can profit from the knowledge of already deployed customers as the scalability, performance and cost numbers of the resources for these customers are already in the system. These numbers can then be used to obtain the numbers for the components to be deployed for new customers.

5.5 Summary and Conclusion

In this chapter the provisioning of *customer-specific application solutions* into *customer-specific application deployments* is described. It is shown how *provi-*

sioning flows can be generated from the application model of a solution. These provisioning flows are provisioning engine independent, as they make use of the generic Cafe provisioning and management interface (CPMI) that is defined in this chapter. It is shown how different provisioning engines can be integrated in Cafe, thus enabling the provisioning of solutions across multiple data centers and even providers. The binding of provider supplied components in a solution to concrete components at a provider is shown. The notion of a *complete component binding* is defined as one combination of components that fulfills all implicit and explicit requirements imposed on the infrastructure by the customer-specific application solution.

IMPLEMENTATION OF THE CAFE PLATFORM

In this chapter the implementation of the Cafe platform is described. In Section 6.1 the modeling tools for Cafe application templates and their role in the Cafe development process are introduced. In Section 6.2 the *Cafe system* is shown. This is the system realizing the functionality required by a provider offering Cafe applications to its customers. In Section 6.2 the integration of the Cafe system with different provisioning and cloud infrastructures is described, too. The overall Cafe architecture is shown in Figure 6.1. Figure 6.1 shows the main components for both main parts, the Cafe application modeler and the Cafe system. The Cafe application modeler consists of the application model editor, the variability model editor and the component binding editor, as well as the customization flow generator. The Cafe system is comprised out of the Cafe application portal, the customization flow generator, the customization flows, the template and solution repositories, as well as the provisioning related components such as the optimization component, the selection facility, the solution creation service, the provisioning support services, provisioning flows,

provisioning services and component flows. The individual components are described in detail in what follows. Since all necessary concepts such as the serialization of the application and variability models as well as the generation of customization and provisioning flows have been already described in the previous chapters, this chapter focuses on the description of the technologies that have been used to implement the proof-of-concept prototype for Cafe.

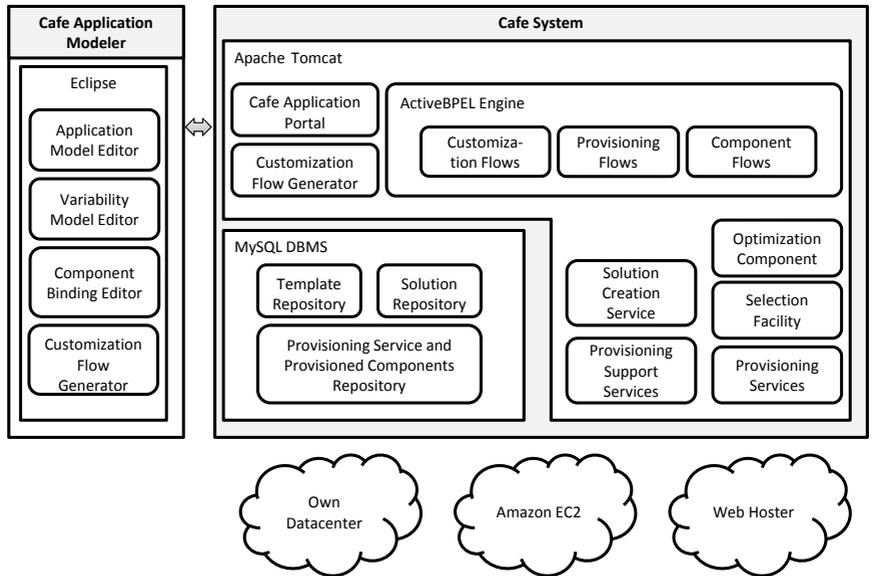


Figure 6.1: Main components of the Cafe architecture

6.1 Cafe Application Modeler

The Cafe application modeler is realized as a set of Eclipse [dRW04] plugins. The reason to choose Eclipse plugins over other implementation technologies are the following: Eclipse as a framework is one of the standard IDEs to develop software today. Several plugins to develop applications using different programming languages (Java, PHP, C++, Ruby, BPEL, ...) and other technologies (XML, WSDL, etc.) exist. Furthermore, plugins to communicate

with version control systems such as CVS or SVN are available. Plugging into an established IDE therefore leaves developers of Cafe application templates with their normal development environment where editors to create or edit the other artifacts needed to build Cafe applications are already present. To build a Cafe application, three main editors are not available in Eclipse, today: an editor to model an application template according to the Cafe application metamodel which can also export this application template as a Cafe archive file (Car file); an editor to model the variability in different artifacts of a Cafe application and an editor to model component bindings. In addition to that, a component is missing that can generate customization flows out of a variability model.

Therefore, in this thesis, prototypes of the three missing editors and the generator have been developed. These four Eclipse plugins shown in Figure 6.1 realize the missing functionality to build Cafe application templates. These four plugins are called the *Cafe Application Model Editor*, the *Cafe Variability Model Editor*, the *Cafe Component Binding Editor* and the *Customization Flow Generator*. In the Cafe prototype the customization flow generator is realized as part of the Cafe application modeler and also as part of the Cafe system which means that customization flows can be generated from the Eclipse tooling and uploaded to the provider. For simplicity if the application developer does not need to modify generated customization flows, another possibility is to generate the customization flows in the Cafe system upon upload of a new application template.

6.1.1 Cafe Application Model Editor

The Cafe Application Model Editor is used to model application templates following the Cafe application metamodel defined in Section 3.3. It is realized as an Eclipse plugin using the Eclipse Modeling Framework (EMF) [MEG⁺03] to define the data model and then Graphical Modeling Framework (GMF) [Ecl] to realize the graphical aspects of the editor.

The Cafe application model editor contains an export and import wizard to export and import a Cafe application model and the referenced files into and

from a Cafe application template archive (Car file). This Car file can then be uploaded to any Cafe application portal. Figure 6.2 shows a screenshot of the modeler.

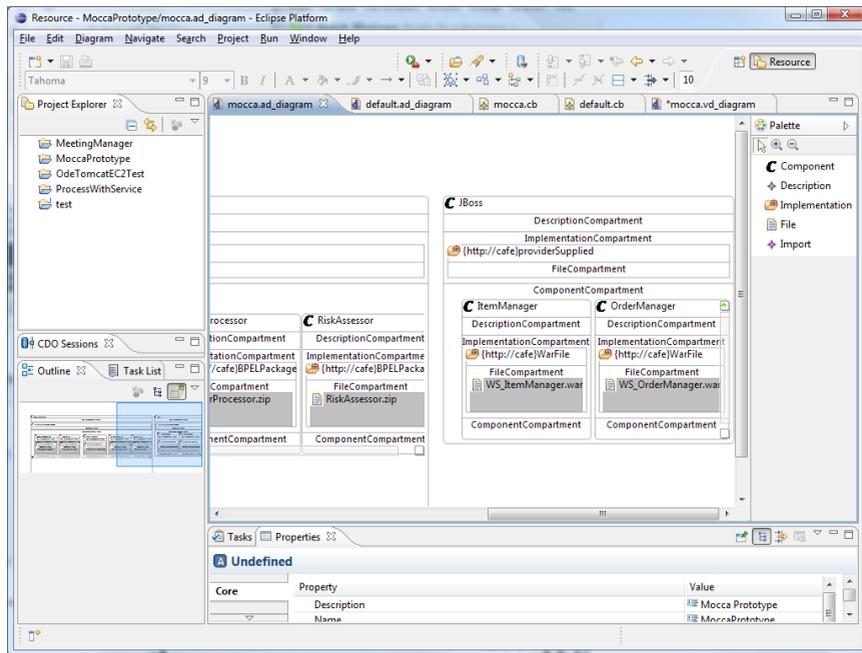


Figure 6.2: Application model editor

6.1.2 Cafe Variability Model Editor

The Cafe variability model editor is also implemented as an Eclipse plugin. The data model is based on EMF and the graphical aspects are built using GMF. The data model is a representation of the variability metamodel shown in Section 3.4.4. The Cafe variability model editor includes an export wizard that can export variability descriptor XML files as described in Section 3.4.4. The Cafe variability model editor is based on the work done in [Din08]. Figure 6.3 shows a screenshot of the variability model editor.

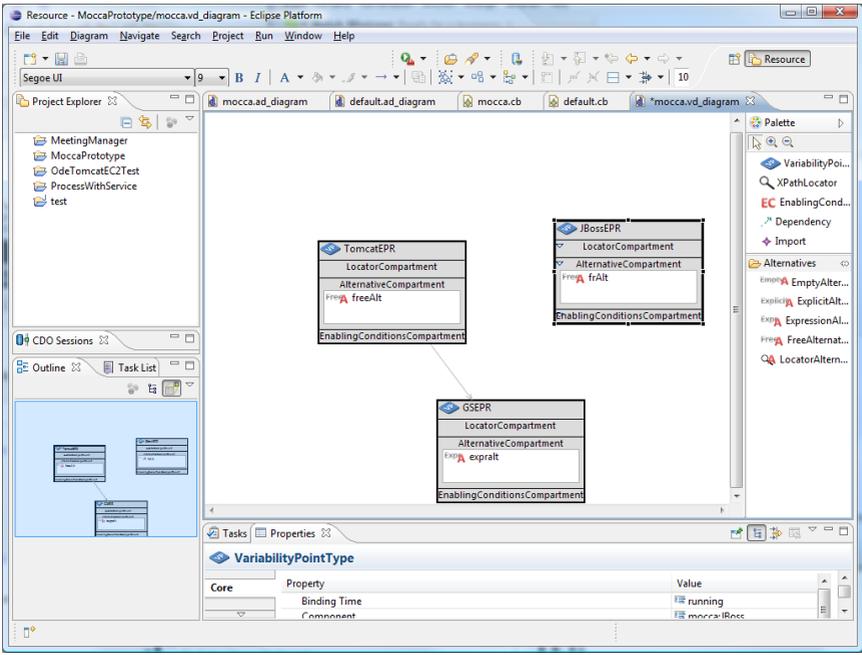


Figure 6.3: Variability model editor

6.1.3 Component Binding Editor

The component binding editor is also implemented as an Eclipse plugin using the Eclipse Modeling Framework. Figure 6.4 shows a screenshot of the component binding editor.

6.1.4 Customization Flow Generator

The customization flow generator is the third eclipse plugin that constitutes the Cafe Application Modeler. The customization flow generator is realized as an Eclipse export wizard. Eclipse export wizards are used to transform and export models or files edited in an editor. In this case the customization flow generator transforms the variability model designed in the Cafe variability model editor into an executable customization flow. The generation rules

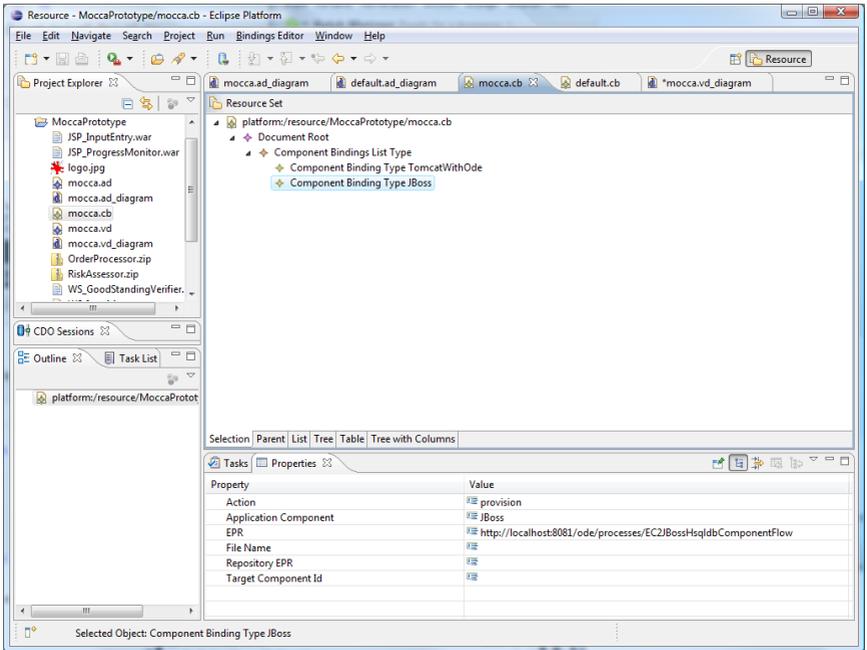


Figure 6.4: Component binding editor

how to generate general workflow models from variability descriptions are described in section 4.2. These rules and the specific rules on how to generate BPEL workflows from a variability descriptor presented in [ML08a, Aue08] are implemented in the customization flow generator. The customization flow generator generates BPEL4People customization flows that are executable on the ActiveBPEL [Act09] BPEL engine. In [Aue08] BPEL processes that are executable on the Intalio BPMS are generated. However, for the Cafe prototype the ActiveBPEL engine has been chosen, as it implements the human interactions following the BPEL4People and WS-HT standards.

To generate a BPEL process, the variability model to be exported is traversed and the corresponding BPEL data structure is built that is then serialized as a BPEL document. Since all customization flows have pre-defined interfaces, the

corresponding WSDL files are referenced from the BPEL document. In addition to the BPEL document a deployment descriptor is generated. After all necessary artifacts for the deployment have been generated, all files are packaged into an Active BPEL business process archive (.bpr) file. Such a file can then be included in the Cafe application template archive (Car) file when uploading the application template to an application portal. Multiple customization flows are generated from the variability model for the different phases following the rules described in Section 4.3.4.

The customization flow generator is also included as a component in the Cafe system. In case no customization flows are included in a Car file, they are automatically created by the Cafe system when an application template is uploaded.

Regarding the Cafe development process the application model editor, the variability model editor and the customization flow generator can be used by application vendors to perform the following steps during the development of an application template: The variability definition, component and variability integration, customization tool development and template packaging steps as described in Section 3.7 are supported by the aforementioned tools.

6.2 Cafe System

The Cafe system is partitioned in several components as shown in Figure 6.1. Customers and application vendors interact with the application portal. The template, solution and deployment repositories hold application templates, application solutions and information about already deployed components. These repositories may be federated with external repositories at other providers that support additional application templates, solutions and information about their deployed components. The provisioning related components of the Cafe system contain the components that are necessary to provision new Cafe applications. These components include customization flows that are used by customers and the provisioning flows to customize application templates. Provisioning flows and services take customized application solutions as input and provision them by making use of the component flows and external provisioning services.

Concrete components and provisioning services are selected by the selection facility and the optimization component that is used to find the most suitable combination of already provisioned components and provisioning services to set up an application for a particular customer.

6.2.1 Application Portal

The Cafe application portal is a Java Web application that runs on a standard Apache Tomcat servlet container. The main parts of the application portal GUI are the *Template Upload*, the *Template Catalogue* and the *My Apps Catalogue* part.

Template Upload. The template upload part is a part in the application portal where providers can upload application templates they acquired from application vendors. In the Cafe prototype this upload is realized by an upload which allows smaller templates to be uploaded. Since templates including the necessary code, for example, for virtual machines can get up to multiple GBs in size, an additional FTP-based upload would be needed in a real product. However, once a Car file is uploaded, the application descriptor is read and the template is added to the repository by moving the files in the Car file to the appropriate folder and by inserting the data from the application descriptor into the database (cf. Section 6.2.3). Then the customization flows if they are not generated by the system, must be uploaded and are installed on the ActiveBPEL engine that will later run them. Afterwards the provider is prompted to perform the template customization for which an instance of the template customization customization flow is started.

Template Catalogue. The template catalogue (cf. Figure 6.5) offers a set of application templates to the customer that is logged in at the application portal. The application templates for the template catalogue are retrieved from the template repository. Customers can select a template that they want to customize and display additional information. On the additional information page of an application template a “customization” link allows a customer to customize the application template to his needs. When clicking

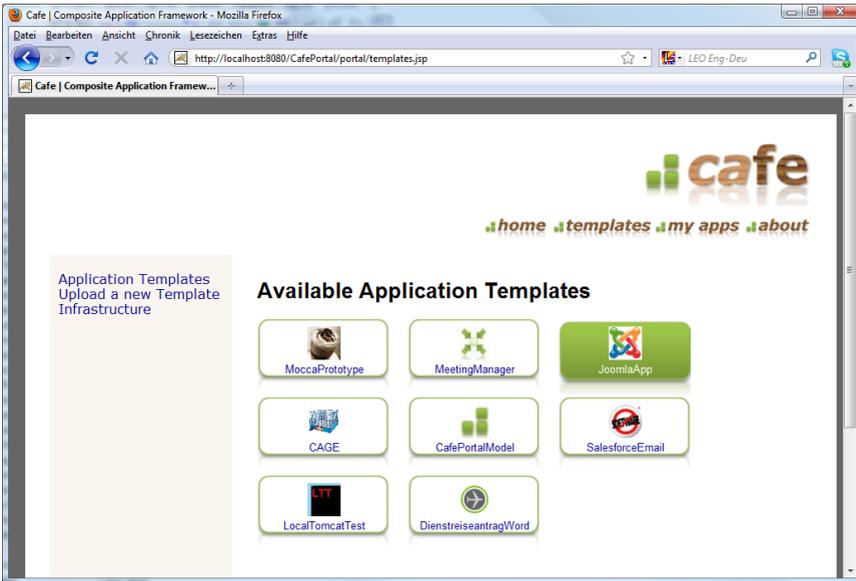


Figure 6.5: Template selection in the Cafe application portal

this link a new instance of the solution engineering customization flow for the chosen template is started if the customization of the template is done via a customization flow. In case the application template developer has assigned a different customization tool, this tool is invoked. In case a customization flow is used, the customer is forwarded to the task list that prompts him for the binding of the variability points as shown in Figure 6.6. The template catalogue realizes the template selection phase as shown in Section 4.5.

MyApps Catalogue. Each registered customer in an application portal can retrieve information about his provisioned applications in the MyApps catalogue. The MyApps catalogue shows a list of installed applications for the customer and shows the state of them, i.e. if they are running, under provisioning or if the solution engineering is under way.

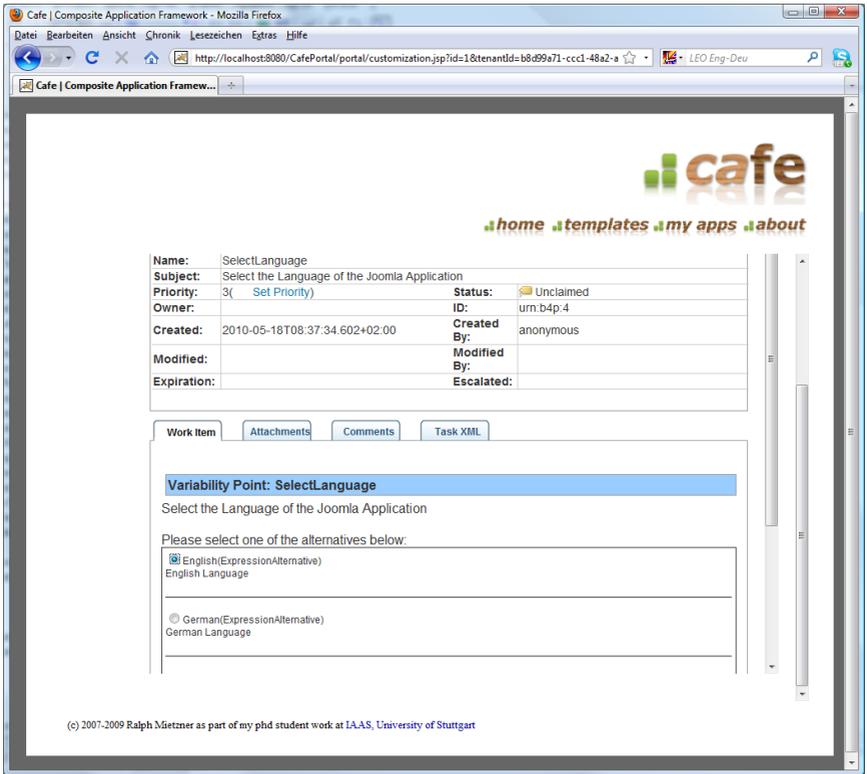


Figure 6.6: Customization flow user interface

6.2.2 Customization Flows

Customization flows are specific for each application template. They have been generated by the customization flow generator (cf. Section 6.1.4) specifically for one application template. Customization flows are implemented in WS-BPEL [OAS07a] using the BPEL4People [Agr07b] extensions for the parts that require customer involvement. Other customization tools for components can be plugged into the general architecture as the general architecture only assumes a standardized interface for these customization tools. Customization flows automatically implement these interfaces if they are generated from

the variability model of an application by the customization flow generator. The Cafe platform provides the runtime for customization flows in form of a BPEL engine. In this thesis the open-source ActiveBPEL [Act09] Engine is used as it was the only open-source BPEL4People engine available at that time. However, other BPEL engines that support human tasks (either in BPEL4People and WS-HT [Agr07a] form or through a proprietary extension) can be used. In [Aue08] customization flows are generated for the Intalio BPEL engine [Int09] to show the general applicability of the approach. Evaluation flows as a special form of customization flows are also generated for the ActiveBPEL engine. Customization flows are instantiated by a `customize` message that contains all customization documents produced during previous customizations for the same template. I.e., an instance of the solution engineering provisioning flow for a template is started with the template customization customization document. This is necessary as the available alternatives at a variability point might depend on the previous bindings of other variability points. In addition to that, some (or all) variability points of the phase the customization flow corresponds to might be bound already so the customization document corresponding to the current phase is also submitted with the `customize` message.

6.2.3 Template Repository

The application template repository contains the application templates uploaded to the portal. It is a file-based repository, i.e., each application template code (the expanded Car file for the template) is stored in a separate directory. A relational database is used to store additional metadata that is needed in the application portal, such as the name and the description as well as the endpoints of the customization flows of the template. Additionally, the template database contains the endpoint under which the customization tools for the template must be invoked and the directory in the directory structure under which the files for an application template reside.

6.2.4 Solution Repository

The solution repository stores the customer-specific application solutions. Similar to the application template repository it consists of two parts, a file-based part where the customization documents are stored and a database-based part. Related metadata is stored in this application solution database. This related metadata contains the reference to the application template from which a solution document has been created as well as the id of the customer that has performed the customer-specific customizations.

6.2.5 Provisioning Services and Provisioned Components Repository

The provisioning services and provisioned components repository stores information about the available provisioning service and the already provisioned components. This includes the information about which component in the environment realizes which component in which application solution for which customer. The application deployment repository additionally contains a list of provisioning services and their provisionable deployments. Thus, the application deployment repository is also consulted during the provisioning to query whether realization components for a certain component are available. Therefore, it is accessed by the selection facility which is described in detail in Section 6.2.6.

The provisioning services and provisioned components repository contains an interface, the *infrastructure registration* interface where providers of components can register component flows (either as complete component flow implementations or as an EPR to an externally hosted component flow) along with the provisionable deployment that provisioning service is able to provision as well as the template customization customization document that identifies how the template customization variability points of the respective component type's variability model have been bound. This is important, for example, if an IaaS provider wants to register the endpoint of component flows which represent provisioning services for components being of a component type that should be made available in Cafe. The infrastructure registration interface is also called by the template upload component when a new template is up-

loaded. Once a template is uploaded to the portal, a new provisioning service is added to the application deployment repository in form of a *standard template component flow* being described as a provisioning service that can provision components of the component type of the template with the multi-tenancy pattern of the template and the components contained in the template. Thus in other templates the component type of the first template can from now on be used as a provider supplied component. If an application template is uploaded to the Cafe portal that itself is of a provider component type (i.e. other components can be deployed on top of it), then a component flow must be registered for that template that overwrites the standard template component flow and includes the operations relevant for deployment (see Figure 5.7). In case a template is of the single (configurable) instance multi-tenancy pattern, a corresponding component flow must be submitted that implements the necessary operations for adding and removing new customers from the component (see Figure 5.8). The infrastructure registration interface also offers operations to remove provisioning services from the Cafe system that should no longer be available for new applications.

The provisioning service and provisioned components repository offers a list of provisionable components and their types that can be used by application vendors to query which component types can be provisioned at that provider.

6.2.6 Selection Facility

The selection facility is responsible for the selection of possible realization deployments and realization provisioning services for a maximal provisioning group or a maximal reuse group. Thus the selection facility takes a maximal provisioning group or a maximal reuse group as an input and returns a set of deployments or provisionable deployments that fulfill the criteria of the respective group as described in Section 5.3.1 and Section 5.3.1. The selection facility is realized as a Web service that can be called from a provisioning flow. The selection facility queries the application deployment repository and returns a set of EPRs corresponding to the component flows of the provisioning services and already provisioned components that match the required criteria.

6.2.7 Optimization Component

The optimization component plays an important role in selecting concrete realization components and realization provisioning services for a customer-specific application solution. The generic optimization component interface therefore accepts a customer-specific application solution as an input and can then make use of the selection facility as well as the deployment repository and the evaluation flows to find an optimal correct and complete component binding for the customer-specific application solution. In the prototypical implementation the optimization component first calls the selection facility to find all realization deployments and realization provisioning services that can realize a maximal provisioning group or maximal reuse group and then calls the evaluation flow for the solution with all possible component bindings. The evaluation flow returns a set of component bindings that are complete and correct from which the optimization component selects the first one. In [Tru10] it is described how such an optimization component could be realized by transforming the problem of finding a cost-optimal component binding into a constraint satisfaction problem (CSP) [Tsa93] which is solved by a *CSP solver*. Different algorithms [Kum92] are investigated to solve the CSP in [Tru10] for an application model that is based to the one presented here but has its own variability mechanism. Applying the results of [Tru10] to the work performed here is not part of this thesis but can be addressed in future work.

6.2.8 Solution Creation Service

The solution creation service is a Web service that is invoked by the provisioning flow. It takes a set of customization documents as well as a variability descriptor as input and applies them to the designated template or solution following the approach described in Section 4.5.2. Thus it takes the locators from the variability descriptor and inserts the selected values for the corresponding variability points at the places the locator points to. Before applying the customizations, a copy of the template is made on which the customizations are then applied.

6.2.9 Provisioning Support Services

The provisioning support services are a set of Web services that are needed by the Cafe platform to perform certain tasks during provisioning. These include an FTP Web service that can upload files from the solution repository via FTP to arbitrary FTP servers, as well as an SCP (secure copy protocol) [Pec07] Web service that can upload files from the solution repository via SCP to servers supporting this protocol (for example Linux-based EC2 instances). These services are referenced from component flows for the *MySQL DBMS* and *Apache Web Server* component types that are hosted at a third party Web hosting company to upload SQL files and PHP Files via FTP. They are also invoked by the component flows for different component types (such as an *Apache Tomcat*, a *MySQL DBMS* or an *Apache Web Server* component type) hosted at Amazon EC2 to upload corresponding files to the AMI instances via SCP.

6.2.10 Provisioning Flows

Provisioning flows are responsible to provision a customer-specific application solution (cf. Section 5.3). Provisioning flows make use of the standardized Cafe provisioning and management interface (CPMI) as described in Section 5.2. The provisioning flow to be executed depends on the customized customer-specific application solution. In Section 5.3 the general structure of a provisioning flow is explained and how it can be generated from the application model of a customer-specific application solution. As shown in Figure 5.17 two general possibilities to generate such a flow are possible. The first possibility is to generate a provisioning flow from the application model of the customer-specific application solution, the second one is to use a generic provisioning flow that interprets the application model. In the Cafe prototype the second possibility is implemented. This second possibility, using the application model as an input for a generic provisioning flow is a more generic approach. Generating specific provisioning flows for an application model is also possible as the specific provisioning flow only contains a subset of the functionality of the generic provisioning flow. The generic provisioning flow in

the Cafe prototype is implemented as a BPEL process model that runs on the ActiveBPEL engine.

Given the rules for the ordering of the provisioning activities for the different components in a customer-specific application solution shown in Section 5.3, on a high-level the generic provisioning flow works as follows:

At first, the customer-specific application model is transformed into a provisioning order graph which is then fed as input to the generic provisioning flow. The generic provisioning flow interprets the provisioning order graph to start the necessary *provisioning actions* for each of the components contained in the provisioning order graph. Therefore, the generic provisioning flow is separated into two main parts, the *provisioning order graph interpretation* part and the *provisioning action executor* part as shown in Figure 6.7. The provisioning order graph interpretation part interprets the provisioning order graph and calls the provisioning actions in the provisioning action executor part as needed. Provisioning actions are those actions that actually provision, deploy or configure individual components as described in Section 5.3.5. These provisioning actions are encapsulated in an event handler in the generic provisioning flow. Since event handlers can be executed multiple times in parallel, this ensures that multiple components can be provisioned in parallel.

Provisioning Order Graph Interpretation. The provisioning order graph interpretation part is essentially a scope which contains a loop that runs until all components have been provisioned. The provisioning order graph interpreter (called interpreter for short) therefore traverses the provisioning order graph, starting from those components that are not dependent on any other components. In each iteration of the loop the interpreter does the following:

1. Find a component ready for provisioning, because all components it depends on are already provisioned.
 - a) If such a component has been found: call the provisioning actions for that component asynchronously.
 - b) If no component has been found: wait until one of the provisioning

actions has signaled that it has finished.

2. Return to step 1.

The loop finishes if all provisioning actions for all components have been executed.

Executing Provisioning Actions. The second part of the generic WS-BPEL provisioning flow is the executor scope. The executor scope is activated in parallel to the interpreter scope. It has an attached event handler that executes the provisioning actions as described in Section 5.3 and shown in Figure 5.19. As shown in Figure 5.19 these actions consist of several steps. The first step is the pre-provisioning configuration for the component. Then, depending on the multi-tenancy pattern as well as whether a provisioning service or an already provisioned component is the realization, the provisioning, deployment or configuration of the component needs to be performed. Afterwards, the component is registered so that it can be tracked which component is used for which application solution. The event handler that contains the provisioning actions is invoked asynchronously. The invocation message contains the EPR of the provisioning service or the EPR of the already provisioned component to be invoked. To perform the pre-provisioning configuration, the event handler calls the associated customization flow with a `customize(component-name, customer-id)` message which asynchronously returns the customization document for the component. The customization document is then copied to the message used to send the `provision` message to the component flow representing the provisionable component that must be provisioned. If an already provisioned component must be configured the customization document is copied to the `subscribe` message that is sent to the component flow that represents the component to be configured.

Ensuring All or Nothing Semantics. Ensuring all or nothing semantics is an important task of the provisioning flow. In particular, this means if the provisioning or configuration of one component fails, the provisioning or configuration of all other components must be aborted or compensated. Given

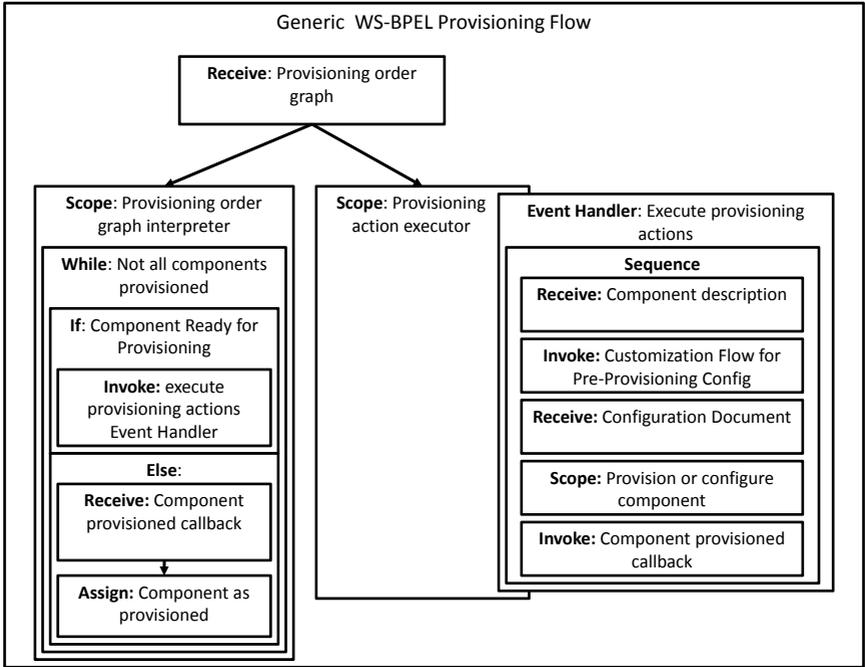


Figure 6.7: BPEL provisioning flow

that the CPMI offers corresponding methods to de-provision components, cancel component reservations as well as cancel and remove deployments and subscriptions, the provisioning flow must ensure that the respective messages are sent so that all previously provisioned and subscribed to components are removed from the environment.

In case a specific provisioning flow is generated from the customer-specific application solution, these compensating actions can be generated directly into the provisioning flow. In case a generic provisioning flow is used, the event handler executing the provisioning actions must ensure that in case of a failure these provisioning actions are compensated. Therefore, the scope directly nested in the event handler that executes the provisioning actions has an associated compensation handler that calls the necessary actions to undo the

provisioning actions. Which actions are performed depends on what has been done to provision the component in the event handler. The `action` variable declared locally in the event handler's scope contains either `provisioned` if a component has been provisioned, `subscribed` if the component has been subscribed to or `deployed` if the component was deployed on another component. The compensation handler then decides based on the value of the `action` variable which actions it has to perform to rollback the corresponding action.

Since the WS-BPEL specification specifies that scopes nested in repeatable constructs such as event handlers must be compensated for each completed iteration, adding a compensation handler to the nested scope of the event handler ensures that all completed scopes are compensated in case the provisioning actions scope in the event handler is compensated. This is ensured by placing a catch-all fault and compensation handler on the provisioning action executor scope as shown in Figure 6.8. These fault and compensation handlers both contain a `compensate scope` activity that points to the provisioning actions scope directly nested in the event handler. As a result, whenever a fault occurs in the provisioning action executor scope or this scope is compensated, all already provisioned or configured components are de-provisioned or the configuration is removed. In case a fault is thrown anywhere else in the provisioning flow (for example in the provisioning order graph interpreter scope), a catch-all fault handler and a compensation handler are attached to the process element containing a `compensate scope` activity that points to the provisioning action executor scope. In case the provisioning order graph interpreter scope faults while the provisioning action executor scope is still running, the provisioning action executor scope is terminated by the fault handler attached to the process element. The default termination handler of the provisioning action executor scope then compensates all completed iterations of the scope nested in the event handler, and thus ensures that all provisioning actions are compensated.

BPEL does not specify any compensation order for scopes contained in repeatable activities that can execute in parallel (such as scopes nested in event handlers). This does impose a problem in the provisioning flow as components

must be de-provisioned in the reverse order of provisioning. Thus in the generic provisioning flow case the compensation handler must ensure that this is the case by traversing the already provisioned components of the provisioning order graph in reverse order and thus executing the rollback operations in reverse order. Reverse order is ensured by maintaining a counter with the order of provisioning for each component and then using this counter in reverse order during rollback.

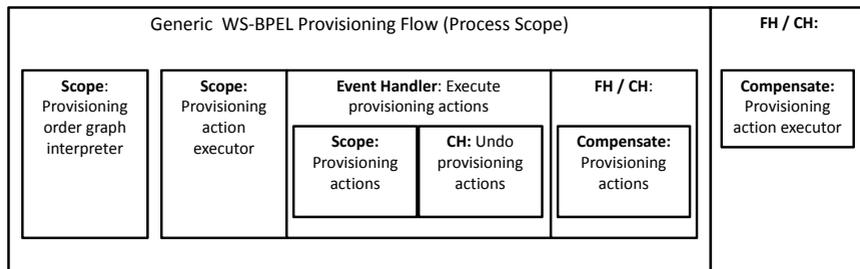


Figure 6.8: Scopes, fault and compensation handlers in the BPEL provisioning flow

6.2.11 Component Flows

Component flows represent components that are already provisioned. Providers of components can provide component flows for these components that are run inside the Cafe application portal as shown in Figure 6.1. However, providers can also run their own component flows outside of the Cafe system. Component flows in the Cafe prototype are realized using WS-BPEL [OAS07a, ML08b] executed on the ActiveBPEL engine. WS-BPEL is used as the implementation language for the component flows in the Cafe prototype, since it is the commonly accepted standard workflow language in Web services-based systems. Numerous BPEL engines are available that ensure reliability, persistence and transactionality which otherwise must be implemented in the component flows themselves. Additionally, BPEL engines allow the parallel execution of multiple process instances which is important as typically several instances of the same

or different component flows are executed in parallel. As a result of using BPEL, the CPMI is implemented as a Web service interface.

Component Flows in the Cafe Prototype. In the Cafe prototype, component flows are implemented for several types of components above a set of different provisioning engines and APIs. As an example the component flow for the component type *Apache Tomcat* on Amazon EC2 is explained in detail here. As all other component flows in the Cafe Prototype this component flow is implemented in BPEL 2.0 and runs on an ActiveBPEL engine.

Components of the component type *Apache Tomcat* are provider components. They accept the deployment of components with the implementation type *Java Web Application*. The component flows for components of type *Apache Tomcat* thus offer the operations and states for normal components and provider components as shown in Figure 5.5 and Figure 5.7. Since the Apache Tomcat hosted on EC2 is not multi-tenant aware the additional states and operations for single configurable instance components as shown in Figure 5.8 are not implemented by the component flow for the Apache Tomcat on EC2.

First the provisioning of a new instance of an Apache Tomcat on EC2 is described.

Reserving a Component. A new instance of the component flow for the Apache Tomcat on EC2 (from now on called “the component flow”) is initiated by a *reserve* message along with the solution engineering customization document. Since Amazon EC2 limits the amount of AMI instances one EC2 user can start, this amount must not have been reached for the account that the component flow is using. Therefore, the component flow accesses a service that increments the number of started instances by one for this reserved instance. This is done so that in case the limit is reached no other components can be provisioned until this reservation is canceled. Thus it guarantees that later on the component can be provisioned. The *reserved* message returns a unique id, the component-id that is set as the *correlation token* [OAS07a] for that instance of the component flow. Future requests must include the component-id so that these requests can be correlated to this process instance. This is not explicitly mentioned in the following message exchanges.

Canceling a Reservation. After a `cancelReservation` message has been received, the reservation of a component must be canceled which is done by calling the service that deals with the number of already started instances of AMIs of one account and the number is decremented by one. Afterwards the instance of the component flow terminates.

Provisioning. Once a component has been reserved it can be provisioned. This is done if the client sends a `provision` message to the instance of the component flow. Depending on the submitted pre-provisioning customization document as well as the solution engineering customization document submitted with the reservation the `runInstance` message to start an instance of the Tomcat AMI on EC2 is prepared. Therefore, the selected alternatives of the customization documents are evaluated, for example, if the selected alternative for the *location* variability point is “Europe”, the corresponding availability zone part of the `runInstance` message is set to “eu-west-1a”. Similar mappings are done for the other variability points such as the requested size of the instance. The `runInstance` message also contains the AMI-Id identifying the AMI that contains the Apache Tomcat. Having mapped all customized parts to the respective parts of the `runInstance` message, the `runInstance` message is sent to the Amazon EC2 API. EC2 then returns an instance-id that is specific to their infrastructure. For all subsequent requests to EC2 this instance-id is reused. However, to communicate with the clients of the component flow, the component-id is still used that thus abstracts from the instance-id. Since EC2 does not offer an asynchronous callback once an AMI has been started, the component flow must poll EC2 by sending `describeInstance` requests containing the instance-id until these requests return a state of running. Once such a response has been received the component flow sends the `provisioned` message to the client that triggered the provisioning. Thus the component wraps the synchronous interface of EC2 into an asynchronous one as required by Cafe.

De-Provisioning. Once a component has been provisioned and it is no longer needed it can be de-provisioned. This is only possible if no other components are deployed on this component any more. De-provisioning is done

if the client sends a `deProvision` message to the instance of the component flow that represents the component. The component flow then sends a `terminateInstance` message to the EC2 API that contains the EC2-specific instance-id obtained during provisioning. Afterwards a de-provisioned message is sent to the client.

Reserve Deployment. Since the *Apache Tomcat* component type is a provider component type, other components that are implemented using the implementation type *Java Web Application* can be deployed on top of a provisioned component of type *Apache Tomcat*. Thus the corresponding component flow must implement the operations and states shown in Figure 5.7. Thus, once a component is in state provisioned and the component flow has returned a `provisioned` message to the client, it can accept `reserveDeployment` messages indicating that a component will be deployed on that component. The `reserveDeployment` message contains all information necessary to deploy the component including the path to the repository where the component code can be found. The handling of the deployment of a new component on a running component is handled in the `deployment event handler` in the BPEL implementation of the component flow. The deployment event handler is implemented as a BPEL event handler. This ensures that multiple components can be deployed on the component in parallel. Each deployment is thus handled by one instance of this deployment event handler. Upon reception of a `reserveDeployment` message the component flow reserves this deployment by adding it as a reservation to a *deployed components* list. This list is an array of deployed components that the instance of the component flow maintains to keep track which components are deployed on top of it. This list can also be used to compute whether additional components can be deployed on top of this component (i.e. based on the amount of processing power needed). The component flow then returns a `reservedDeployment` message containing a deployment-id that uniquely identifies the reserved component among the reserved and deployed components for that provider component.

Cancel Reservation. Deployment reservations can be canceled by sending a `cancelReservation` message containing the deployment-id that has to be

canceled. The component flow then removes the deployment from the list of deployments and sends a `canceledReservation` message to the client. Afterwards the instance of the event handler that handled this deployment terminates.

Deploy. After the deployment of a component has been reserved it is actually deployed once the component flow receives a `deploy` message. The component flow in the Cafe prototype makes use of a “copy Web service” that can copy files from a repository to a specific folder of the running EC AMI instance. This is done using the Secure Copy Protocol SCP. This Web service needs the repository location of Java Web Application Archive (.war file) to copy, as well as the credentials and the IP-Address of the EC AMI instance as input. It also needs the directory on the instance where the files should be copied. Since all this information is available in the component flow, it is sent as a `copy` message to this Web service answering with a `copied` message once it has finished the copying. Since Tomcat allows the deployment of Web applications by simply dropping the .war file into the “Webapps” folder of the Tomcat installation, this folder is given to the copy Web service and no additional actions must be taken.

The deployment of components of other implementation types on other component types might require more complex interactions. In the Cafe prototype a component flow exists that represents a *MySQL DBMS* component type on Amazon EC2. Database components implemented using an implementation type of *SQL Scripts* can be deployed on this component. In this case the component flow must first upload the SQL script to the server as it is done with the Web application in the Tomcat case. After that it must issue the `mysql` command with the SQL script as input to create the new database. This is done by calling an “SSH Web service” that can issue commands, such as the `mysql` command, on the server via SSH.

Undeploy. Once a component deployed on another component is no longer needed, it can be undeployed via the `undeploy` message containing the deployment-id. The deployment-id correlates with the right instance of the component flow BPEL process and the right instance of the deployment event

handler in this instance of the BPEL process. In case of the component flow for the *Apache Tomcat* component type on EC2 an “SSH Web service” is then called with a delete command containing the file name of the .war file that represents the deployment to be removed. Once this has been done a `undeployed` message is sent to the client and the deployment is removed from the list of deployments. In case the list of deployments is empty the component can be de-provisioned.

Again the undeployment can also be realized through more complex actions that require a more complex component flow than the simple one described above.

Get component properties. The component flow for the *Apache Tomcat* component type on EC2 contains a *properties event handler* that becomes active once the reservation of the component started. Thus the reservation and all subsequent actions are located in a scope that has the properties event handler attached. The properties event handler reacts on `getProperties` messages and computes the properties document for this component that contains all visible properties of the component type of the component. The properties document is formatted following the guidelines for WS-RF resource properties documents [OAS07b]. This means that each bound variability point is represented by an XML element in the answer that contains the bound value of this variability point. All these elements are nested in an `getProperties` element.

Set component properties. The *Apache Tomcat* component type does not have any variability points that must be bound at runtime and thus the component flow does not contain a `setProperties` operation implementation.

6.2.12 Abstract BPEL Processes as Blueprints for Component Flows

BPEL allows to specify *abstract processes* which are BPEL process models containing *opaque* parts. These opaque parts must then be specified during the *executable completion* [OAS07a, MML08] when an executable process is built from the abstract process. In Cafe abstract BPEL processes are used to describe

the interfaces for component types. The abstract BPEL process, for example, for the component type *Apache Tomcat* can then be implemented by several providers that provide the component type *Apache Tomcat*. For example, Amazon EC2 (or a third party that wants to make an Apache Tomcat Amazon Machine Image available for Cafe) implements a mapping from the standard operations of that component type to the operations of Amazon EC2.

6.3 Summary and Conclusion

In this chapter the implementation of the Cafe prototype is shown. It is shown how the *application model editor*, the *variability model editor* and the *component binding editor* are realized as Eclipse plugins. These three editors allow the development of Cafe application templates. In addition to that, the variability model editor contains a plugin - the *customization flow generator* that is used to generate the various customization flows for a variability model of an application template. The second part of the chapter describes the implementation of the Cafe system, including the *Cafe application portal*, the *template and solution repositories* as well as the implementation of the provisioning infrastructure that consists of *provisioning flows*, *component flows* and *customization flows*.

APPLICABILITY AND CASE STUDIES

In this chapter the general applicability of the Cafe concepts is evaluated. At first the metamodels are revisited and it is shown how they fulfill the requirements stated in the research issues (cf. Section 1.2). It is also shown how the Cafe approach deals with the deficiencies of existing work as identified in Chapter 2.

Having evaluated the general applicability, several example applications are described where Cafe concepts have been applied. The first application is the E-commerce cOncept (EccO) prototype [Tit09, MUTL09], an automotive point-of-sale application that is hosted in an internal SaaS model. EccO demonstrates how multi-tenancy patterns can be used in a service-oriented application.

The second application is the BPM on-demand application DecidR, a Web-based solution that allows users to design, monitor and execute human-driven workflows. The DecidR platform itself can be modeled as a Cafe application, additionally the applications that users can model inside the DecidR platform can be described as Cafe application templates and automatically configured

and provisioned. In Section 7.3 it is shown how the UI components of the Cafe system can be replaced with custom components and Cafe can be used in a *headless mode* without the graphical user interface.

The third application is a sample application that exploits the full potential of the Cafe platform. This application is a sample credit approval application built on top of a service-oriented architecture. The technical details of this application resemble a real-world cloud application prototype we built as a proof-of-concept prototype for a large enterprise and is described in detail in Section 7.4. As a last evaluation the Cafe application portal itself is described as a Cafe application. This allows providers to set up different application portals for different customers.

In [SML08, SML09] we explore how complex integration scenarios based on EAI patterns [HW03] can be automatically provisioned and offered as a service. This EAI as a service (EaaS) approach benefits from the Cafe application model, as EAI scenarios can be mapped to a Cafe application model and thus can be automatically provisioned using provisioning flows. The EaaS approach is an interesting example of a Cafe application as each integration scenario is completely different, however all integration scenarios are composed of a very limited set of different components that can be reused across different integration scenarios. The EaaS approach is briefly touched in Section 7.6.1.

In the following sections these case studies are described in detail and the benefits of Cafe regarding these applications are described.

7.1 General Applicability

In this section the applicability of the Cafe models, algorithms and implementation regarding the research issues stated in Section 1.2 is evaluated.

7.1.1 Application Modeling and Deployment

Two main research issues, stemming from the deficiencies of existing work to model and package applications, need to be considered when evaluating the applicability of the Cafe application metamodel and Cafe variability meta-

model. The first research issue is the provider-neutral modeling of application components and their relation among each other as well as requirements on the provider infrastructure. The second issue is the separation of application vendors from application providers through a self-contained package format for arbitrary composite applications. Existing packaging standards such as OVF [DMT09] or EAR files [Sun08] only allow the packaging of very specific types of applications or virtual machines. As shown in Section 2.2.11 existing standards such as SCA [Ope07] or the reusable asset specification [OMG05a] do not allow to model the requirements on the provider infrastructure.

The closest solution to the two research issues is the approach presented in [AEK⁺07] that is based on the models introduced in [EMME⁺06]. However, the Cafe approach is more general, as it allows to model arbitrary components from servers up to individual software components implemented in any programming language. In [AEK⁺07] an example of a Web service deployed as an EAR File in a JEE container is given. The Web service is modeled as an *EARUnit* with a requirement on a *J2EE Container* capable of a *J2EE Version* ≥ 1.4 . This requirement is resolved via a *hosting link* to a *Was6Unit* which represents a WebSphere application server during deployment. The *EARUnit* also has a requirement of a *J2EEDatasource* that is linked via a *dependency link* to a *DB2DataSourceUnit* that requires a WebSphere application server (*WasServer*) providing the data source. Again this *WasServer* is mapped to the *Was6Unit* via a *hosting link* during deployment.

In Cafe this application could be modeled as a provider supplied *JEE Container* component of type *JEE Application Server* containing two additional components: The *Web Service* component and the *Data Source* component. The *Web Service* component has an implementation of type *EAR File*, while the *Data Source* component has an implementation of type *DataSource specification* which is a simple XML file describing the data source in the format understood by JEE containers. The version requirement on the JEE container component is modeled as a component binding time variability point refinement in Cafe refining the *JEEVersion* variability point exposed by the *JEE Application Server* component type. Additionally, variability points can be added that configure the deployment descriptor contained in the EAR file with the name of the data

source once that has been deployed. Thus, upon deployment of the application, first a suitable component of type *JEE Application Server* will be searched or deployed if none is available. Then the *Data Source* component is deployed by calling the `deploy` operation of the component flow representing the *JEE Container* component which then in turn calls the deployment interface of the concrete JEE container to deploy the datasource. Having done so, the EAR file is deployed in a similar way.

So far, the two approaches provide similar functionality. The advantage of Cafe over the approach presented in [AEK⁺07], however, lies in the composition of application components. While the *EARUnit* in [AEK⁺07] can only be configured with predefined parameters, variability points for the *Web Service* component in the Cafe example can configure arbitrary parts of the *Web Service*. In addition to that other components in the application (such as a *Web Service client* component) can have variability that affects the variability of the *Web Service* component or the allowed values of the *JEEVersion* variability point. For example, because if a certain functionality is required in the *Web Service* client it must be deployed on a JEE 5.0 compliant application server, while otherwise a J2EE 1.4 compliant application server would be enough. These cross-component variability dependencies can be modeled through dependencies and enabling conditions using the Cafe variability metamodel, but they cannot be modeled in the approach presented in [AEK⁺07], as this approach is not focused on the composition of components in applications, but more on the modeling of complex infrastructure topologies. The Mocca application, presented in Section 7.4 further highlights these advantages of cross-component configuration. The powerful variability mechanism of Cafe also tackles the research issues of a generic variability model, as it allows to specify variability in arbitrary components of an application ranging from low-level infrastructure components to high-level application components. It also allows to define arbitrary complex dependencies via dependencies and enabling conditions, permitting complex configuration across components. The variability metamodel combined with the evaluation flow also provides a solution for the QoS-aware provisioning research issue, as it allows to capture complex QoS dependencies of individual components in the variability model of an application that are

then automatically resolved during provisioning.

As shown in Section 2.2.11 the APS [SWS07] standard allows modeling of Web applications including requirements on the provider infrastructure for this limited set of applications. To emphasize the generality of the Cafe approach it is shown how APS applications can be modeled and provisioned using Cafe. As the APS specification document lacks a concrete example the most popular application from the APS Website's Application Catalog was chosen as a target for evaluation. The most downloaded application packaged using APS is the Joomla [Ope10a] content management system and application framework. The application package consists of a set of folders containing the PHP scripts, HTML pages and PNG images that realize the application. The application metadata needed for an APS-enabled engine to provision the application is contained in the APP-META.xml file in the root directory of the application. As the entire application is essentially one component, it can be modeled as one *JoomlaApp* component in Cafe with the implementation type of *PHP Application*. The folders containing the code are then specified as the files for the implementation in the Cafe application model. The metadata related to the presentation of the APS application, such as version information, screenshots, vendor websites, change logs, licensing information, icons etc. cannot be explicitly modeled in Cafe. However, this data is not necessary to provision the application and can be included in the description tag of the Cafe application descriptor.

It is now shown how the provisioning-related artifacts of the APS metadata descriptor (called *APS descriptor* from now on) are mapped to Cafe. The APS descriptor contains a set of *entry-points* that specify under which URL the application can be accessed. These entry points can be modeled with variability points, that concatenate the server URL with a pre-defined relative path. After provisioning, the customer can then use the values of these variability points to access the application. In the Cafe system the following naming convention exists: Each variability point named "EntryPoint" is presented to the customer after provisioning. Thus Cafe allows the automatic computation and presentation of these entry points. The next important artifact in the APS descriptor are the *settings*. These are essentially variability points that must be filled during

the provisioning. However, settings are independent from each other. These settings can be directly mapped to variability points. The Joomla example APS descriptor contains, for example, settings where an Administrator must set the administrator password and email. These can be mapped to variability points with free alternatives in which the customizer must enter the required values. Another setting is the site name (i.e. the name of the application) including a default value of “My CMS”. This can be modeled using a variability point with a free and explicit alternative. The next setting is the default site language where one of different languages can be chosen. This is modeled in Cafe as a variability point with a set of explicit alternatives representing the languages. As these settings are mapped to variability points that must be bound during solution engineering, a customization flow that prompts the customer for decisions during the solution engineering is generated from the variability model. This customization flow as described and proven in Section 4.1.2 guarantees that the customization is correct and complete and thus the resulting application can be deployed. The generation of customization flows thus satisfies the research challenge of guiding customers during the customization while guaranteeing correct and complete customizations.

The APS descriptor describes a set of requirements on the underlying infrastructure, for example, which version of PHP is needed and which PHP extensions must be present. This can be modeled as follows in Cafe: The *JoomlaApp* component is modeled as being contained in a *ApacheWebServer* component of type *Apache*. The component type *Apache* specifies an abstract variability model containing variability points such as the version of PHP that is supported or which PHP extensions are present. All possible properties for PHP are described in the *PHP aspect* of the APS specification and can be modeled as variability points in Cafe. Other aspects such as an *ASPNET aspect* or a *MySQL aspect* are also available and can be used to derive the required variability points required for such component types in Cafe. Thus these variability points derived from the PHP aspect are imported in the Joomla application variability model and can be refined to values required for the successful provisioning of the Joomla application. Consequently, during the provisioning the selection facility only selects *Apache* components that are configured in the requested

manner. The same holds true for the requirements stated in the APS descriptor for the MySQL database required to run the application. These are mapped to variability point refinements for the imported variability points from the abstract variability model of an additional component of type *MySQL DBMS*. The database required for the Joomla application is supplied in the Joomla package as an SQL script. In Cafe this is another separate component (the *DB* component) with an implementation type of *SQL Script*. The implementation specifies the SQL script as a file realizing the implementation.

The provision part of the APS descriptor describes how the application is provisioned. This is done by specifying a configuration script contained in the application that is called once the files have been copied to the server. This configuration script is a PHP script that takes all the settings from the settings part described above as parameters and performs the necessary actions to apply them. Abstractly speaking, it does the same as the solution creation service in Cafe, namely applying the customization document to concrete artifacts. However, as it is an arbitrary application it can do more than just modify documents. In the Joomla example all actions performed by the configuration script can be mapped to modifications of documents. One example is the selection of the correct SQL script to fill the database with language-specific settings. Depending on the language chosen in the settings, a different SQL script is executed. This is mapped to a variability point in Cafe with a locator pointing into the application descriptor that modifies the value for the file realizing the *DB* component depending on the language selected.

In case more complex configurations are needed that cannot be mapped to modifications of documents, the concept of a configuration script can also be mapped to Cafe concepts. This concept is also present in OVF, in the form of *activation engines* that execute upon start of a new instance of an image to configure certain aspects of the image depending on external settings.

Inclusion of configuration scripts has been implemented in the Cafe prototype in the following way: The component flow for a component of type *Apache* on Amazon EC2 executes a configuration script via SSH after it copied the component implementation to the server. The name of the configuration script is specified in the *ConfigScriptName* variability point of the abstract variability

model of the component type *Apache*.

7.1.2 Variability Modeling

In the previous examples the general applicability of the Cafe approach with regard to the modeling of arbitrary applications and their components was shown. It was also shown how cross-component variability can be achieved. As one research issue to be tackled by Cafe is the definition of variability in composite applications, it is now evaluated how not only the cross-component variability can be tackled but also the variability in one component implementation. This is done by evaluating different approaches to model variability in BPEL processes and by showing how scenarios modeled using these approaches can be modeled using the variability models of Cafe. Given the general nature of the Cafe variability model, the modeling is more complex. In [Gue08] extensions to the variability metamodel of Cafe are introduced facilitating the modeling of variability in BPEL processes. However, all these extensions can be mapped to the basic elements of the metamodel as described in Section 3.4.4.

In [SA08, KSSA09] VxBPEL is presented as one approach to model variability in BPEL processes. The basic approach is to include *variation points* directly in the BPEL process model that describe which parts are variable. A variation point in VxBPEL can contain several *variants* which contain a *BPELCode* element that describes the BPEL code used if this variant is selected. Variation points, according to [KSSA09] can be placed anywhere where activities can be placed in normal BPEL. Thus they only target activities, whereas the Cafe variability mechanism can point to individual attributes in any place of a BPEL process via an XPath locator as shown in Figure 3.17. On page 262 in [KSSA09] the authors present an example where they model *service replacement* with VxBPEL. This is done in the following: Instead of adding an *Invoke* activity directly into the BPEL process that invokes a service via a *partnerLink*, they add a variation point that contains two variants with two different *invoke* activities each of which has a different *partnerLink* associated with it. During customization one variant can be selected and thus the corresponding service is invoked. The concepts of VxBPEL can be mapped directly to the variability metamodel in

Cafe. The variation points are mapped to variability points, the variants to explicit alternatives that contain the BPEL code as value that is contained in the variant. Additionally, a locator has to be specified that points into the part of the BPEL file where the variation point is located. Also the default variant could be left in the location in the BPEL code where the variation point is located and a locator alternative can be specified in the Cafe variability model with the default attribute set to `true` that represents this variant. The service replacement scenario shown in [KSSA09] is ideal to illustrate the power of the Cafe variability mechanism. VxBPEL does not allow to describe dependencies among individual variation points. However, this is needed, as for example, the BPEL process could invoke the same service again, and it must be ensured that the same `partnerLink` is used at both variation points. While this cannot be modeled in VxBPEL, it can be modeled in Cafe using dependencies and an enabling condition. In addition, if the BPEL process is part of a larger application, the Cafe variability allows the inclusion of only the service in the customer-specific application deployment that is actually used by the BPEL process (specified through the `partnerLink`), by adding another variability point that configures the application descriptor of the application to remove the component containing the service targeted by the `partnerLink` that is not used, thus the variability mechanism of Cafe is a superset of the variability mechanism of VxBPEL.

Another approach dealing with the configurability of BPEL process models is the approach presented in [LL07]. This work deals with two aspects: *object-based* customization, and *process-based* customization. Object-based customization deals with the addition, removal or replacement of a single process element (an activity, link, ...). Process-based customizations affect multiple parts in a process. Two cases are given for process-based customization: Restriction of possible providers for an activity implementation (such as the selection of a concrete Web service that implements an activity) and global customization variables that are set once and then haven an influence on how the rest of the process model is executed. Addition, removal and replacement of process elements (such as activities) can be modeled with variability points. Addition can be modeled in Cafe using a variability point with an explicit

alternative and a locator with the action attribute set to *before* to indicate that the value of the explicit alternative must be inserted before the building block to which the locator is pointing. Removals are modeled using a variability point containing an empty alternative. Replacements are modeled using an explicit alternative and a locator action of *replace*. As XPath locators in Cafe allow to address arbitrary XML Nodes in a BPEL process document, all process elements can be modified using these basic mechanisms. In [LL07] *semantic transition rules* are defined, dealing, for example, with the reconfiguration of BPEL Links when an activity is removed. These semantic transition rules can be mapped to additional variability points that are dependent on the original variability point and that, for example, reconfigure the links, in case an activity has been removed. In contrast to the approach in [LL07] these must be explicitly defined in Cafe, as the variability metamodel is not aware of the semantics of BPEL, however, additional tooling on top of the variability metamodel, such as the tooling introduced in [Gue08], can cope with this problem. Restrictions on the selection of providers can be done in Cafe in form of restrictions on how the `partnerLink` for messaging activities can be restricted as shown in the VxBPEL example. Global configuration variables, as introduced in [LL07], are variables that are set before deployment of a process model. The process model then contains control flow artifacts that take choices based on the values of these global configuration variables. This can be modeled in Cafe by adding one variability point per configuration variable with a set of alternatives, specifying allowed values for a configuration variable. The locator of the variability point then points to the literal value of the `assign` activity that initiates the configuration variable.

By showing that all functionality of the two approaches, dealing explicitly with BPEL customizability, can be replicated with Cafe, we showed that despite the general nature of the variability metamodel, even very specific problems can be tackled using the presented approach.

In the following, different cases where Cafe concepts were applied are shown to highlight additional use-cases where Cafe can be employed.

7.2 Ecco

The E-commerce cOncept (eCCo) prototype is an example of an internal SaaS application that has been used to evaluate the multi-tenancy patterns and variability descriptors in a real-life scenario. eCCo is a prototype for a dealer point of sales application in the automotive sector. Dealers use the application to order new vehicles or sell used vehicles from their inventory. eCCo has been developed in conjunction with a partner from the automotive industry [Tit09] incorporating real-life requirements from a global car manufacturer. The eCCo application is built using a service-oriented architecture realized with Web services. The GUI is a Web-based system that accesses a workflow layer that is realized using WS-BPEL. The various BPEL processes invoke additional BPEL subprocesses and other Web services.

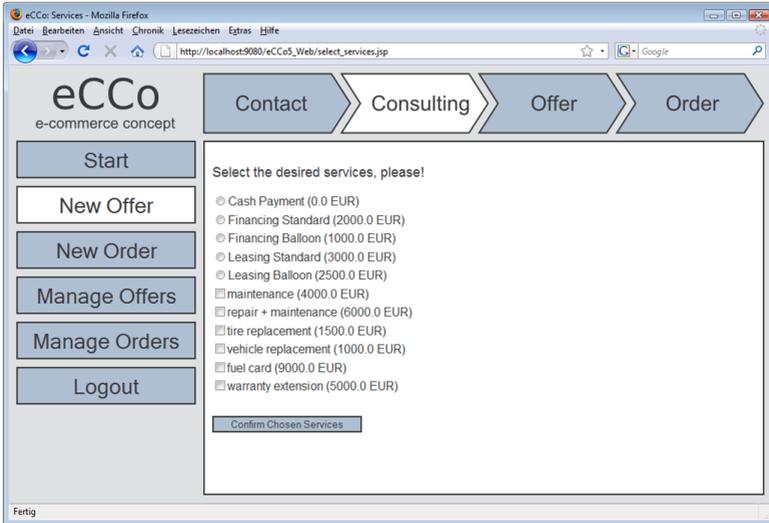


Figure 7.1: eCCo screenshot

The eCCo application is meant to be hosted in an “internal SaaS” delivery model. This means that eCCo is hosted in a central location in the car manufacturer’s data center. The tenants are the different distribution companies

for the different markets the car manufacturer operates in. These distributors subscribe to the application and customize it to the need of their market. These market customizations include different available services, such as different financing options. Additional customizations contain different available languages and tax rates. Customizations do not only affect the services and GUI but also the workflow layer. While the general workflow for the ordering of a car is the same in all companies slight variations of the workflow are possible. For example, when comparing the Spanish tenant with the German tenant, the Spanish tenant requires an additional step in the workflow after an offer has been created. This offer must be signed by a customer according to Spanish law while it does not need to be signed according to German law.

7.2.1 Multi-Tenancy Patterns in eCCo

In the following we investigate how multi-tenancy patterns have been used in the development of eCCo.

Single Instance Services. Several single instance services exist in the eCCo system. These services, such as the vehicle database service are the same for all tenants. Therefore, all tenants can access the same vehicle database and can order all vehicles in that database.

Single Configurable Instance Services. Single configurable instance services in the eCCo system exist on different layers of the application. For example, the GUI is realized as a single configurable instance service. Configuration for the UI is, for example, the set of available languages. Other configuration data includes the form for the aforementioned extra step in the Spanish tenant, which is realized as an additional form that is only available to that tenant. The workflow layer is also partially realized as a single configurable instance service. For example, the overall “vehicle ordering” process is the same for all tenants. However the process is aware which tenant invoked it and therefore includes a tenant context in the invocation of subprocesses. These invocations are routed to the suitable subprocess via a tenant-context based router. For example, upon invocation of the subprocess “create order” the invocation is

routed to the subprocess that corresponds to the correct processing for the country the dealer is in. For example, for a Spanish dealer the subprocess with the extra confirmation step is invoked.

Multiple Instances. Multiple instance services are also present in the eCCo application. As mentioned before multiple “create order” subprocesses exist. There is a special one for the Spanish market that contains an extra confirmation step. Other markets, such as Germany, Austria and the US, do not need this extra step. In this case a mixed pattern is used. Germany, Austria and the US use a single configurable instance of the “create order” subprocess whereas Spain uses an extra instance. This example shows again that mixing multi-tenancy patterns is possible and important.

7.3 Decidr

Decidr [Dec09] is a *business process management as a service* framework. Decidr allows customers of the Decidr platform to model, execute and monitor human-driven business processes in a Web-based environment. Decidr therefore contains a Web-based editor shown in Figure 7.2 to model business processes using a set of pre-defined activity types such as E-Mail activities and human task activities. Once these processes have been modeled, customers can publish them and they can be instantiated by users. Upon publishing, the modeled workflow is transformed into a BPEL workflow which is then deployed on an Apache ODE BPEL engine. The BPEL workflow makes use of pre-defined services such as an e-Mail Web service and a human task Web service. As multiple tenants can use the Decidr application upon deployment of a new workflow it must be checked whether the already deployed Web services can be reused in the new workflow.

7.3.1 Components and Provisioning

Speaking in Cafe terms, one abstract application template exists. This template contains a workflow component, that can be customized via the Web-based editor. Additionally, it contains a set of service components (such as an e-mail

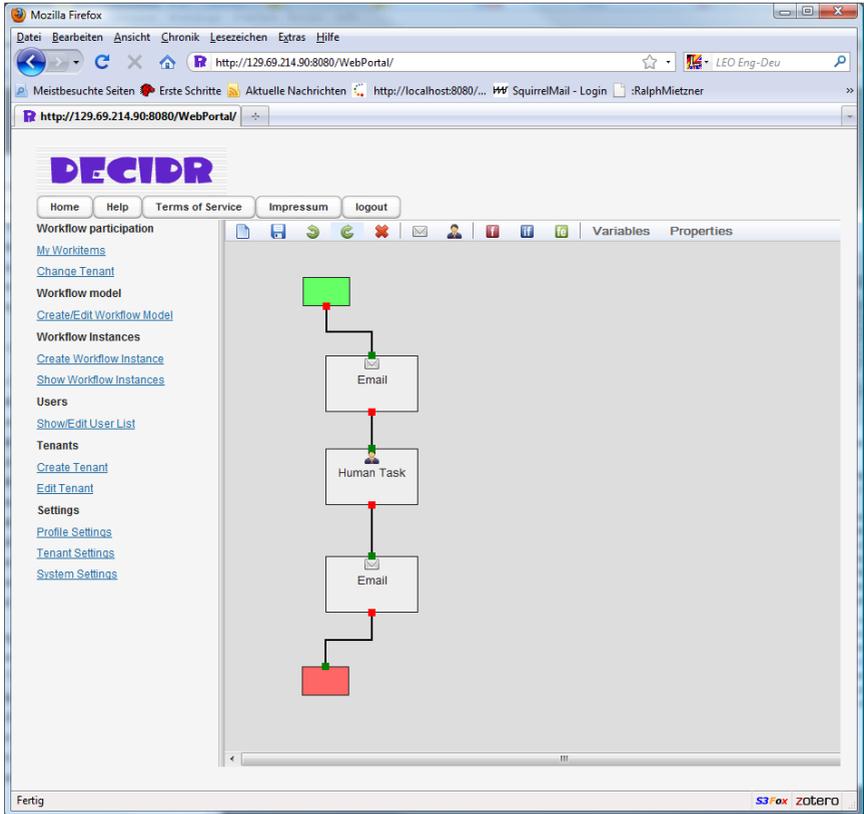


Figure 7.2: DecidR Web-based modeling tool

component and a human task component) that can be customized via the Web-based editor. Additionally, these components are only contained in the customer-specific application model if corresponding activities are contained in the workflow.

Upon publishing of a workflow the customer-specific application solution application model is handed over to the Cafe provisioning components. The application model contains the workflow in the multiple instances pattern as well as the necessary service components in the single configurable instance

pattern. The service components are also marked as “provider supplied”. Thus the provisioning flow first searches for suitable service components and annotates the application model with the found service components and the possible realization provisioning services available in the DecidR platform.

7.3.2 Headless Cafe

DecidR makes use of the Cafe platform without the GUI components. Cafe without the modeling tools, the application portal and the customization tools is called *headless cafe*. DecidR shows the applicability of Cafe in such a setting. Additionally, it shows how individual components of the standard Cafe architecture can be replaced by other components. In DecidR the Eclipse-based generic modeling tools (application modeler and variability modeling tool) are replaced by a domain-specific Web-based workflow editor that produces as output a customer-specific application solution. This customer-specific application solution is then handed over to the headless Cafe platform which takes over the provisioning.

The flexible architecture of the Cafe platform thus allows to replace generic components with specific components that are suitable to a specific domain such as business process management in the DecidR case. The grayed parts of Figure 7.3 are those omitted by the headless installation of Cafe in the DecidR, whereas the other components are present in the standard Cafe architecture as well as in DecidR.

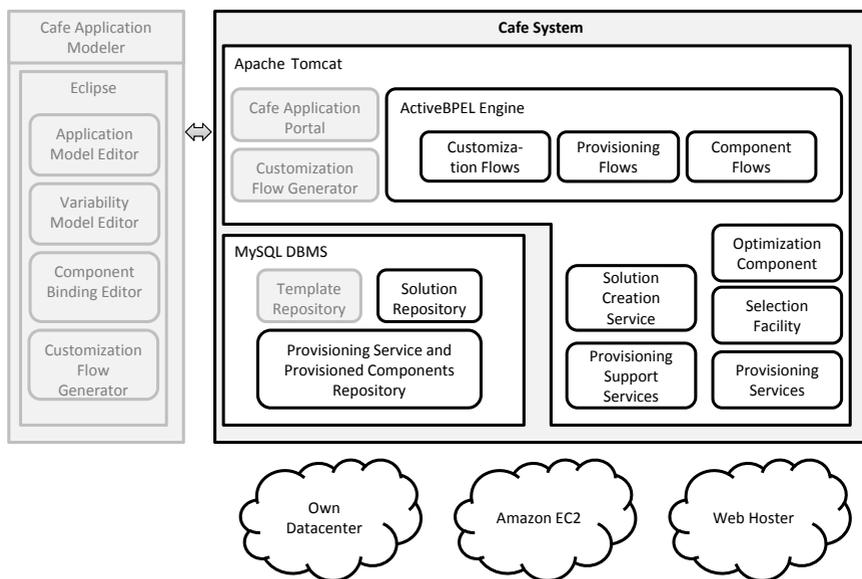


Figure 7.3: Omitted components in the headless Cafe usage in DecidR

7.4 Mocca Sample Application

The Mocca [LDF⁺10] application is a sample application that realizes a credit application process. It is built following the SOA architectural style using Web service technology. It contains a set of UI components realized as Java Web applications as well as some Web services, realized using Java and Apache AXIS and some business processes, implemented in BPEL.

7.4.1 Components

As shown in Figure 7.4 the application consists of nine distinct components. Two of those components are middleware components (*JBoss* and *Tomcat with Ode*). These middleware components are of the implementation type *provider supplied*. No implementation files for these components are delivered with the template.

Two Web service components, the *ItemManager* and the *OrderManager* components need to be deployed on the *JBoss* application server component. The *JBoss* application server component, as well as the *ItemManager* and *OrderManager* components are in the multiple instances component pattern, thus a new *JBoss* application server is started (at Amazon EC2) for each customer, and the Web services are deployed on it afterwards.

The *InputEntry* and *ProgressMonitor* components are standard Java Web archives containing JSP files and HTML files that realize the UI of the sample application. These have a deployment relationship on the *Tomcat with Ode* component. The *GoodStandingVerifier* component is a Web service that is also deployed on the *Tomcat with Ode* component. The *OrderProcessor* and *RiskAssessor* components are of implementation type *BPELPackage* as they are Zip files containing all necessary artifacts to deploy them on the Apache ODE BPEL engine that is contained in the *Tomcat with Ode* component.

7.4.2 Variability

The variability of the Mocca Sample application is mainly concerned with user interface aspects and the configuration of the components that need to invoke each other. Figure 7.5 shows a part of the variability model of the sample application. The first part of the variability model is the solution engineering variability. Thus, variability points with binding time solution engineering exist that target the title and background color for the *InputEntry* component. Another solution engineering variability point exists, that allows selecting whether the application needs to be private or public and thus must be provisioned in the local data center or on Amazon EC2.

The other variability points in the variability model for the sample application are concerned with the configuration of the individual components. For example the *OrderProcessor* BPEL process running on the *Tomcat with Ode* engine must be configured to use the *OrderManager* Web service running on the *JBoss* application server. Thus, after the provisioning of the *OrderManager* and the binding of the *OrderManagerEPR* variability point the *OrderManagerEndpoint* variability point of the *OrderProcessor* BPEL process can be bound with

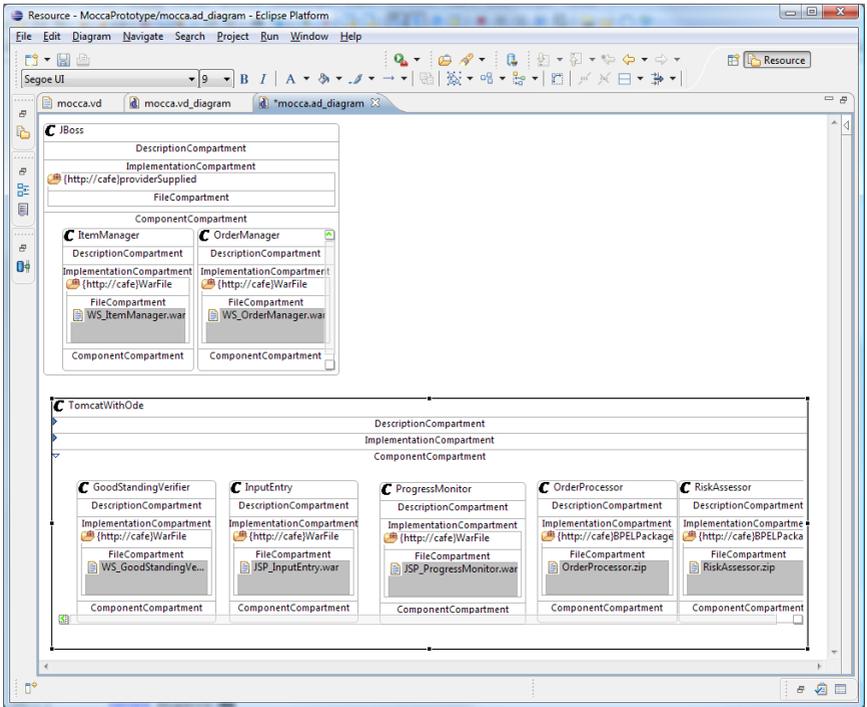


Figure 7.4: The components of the sample application in the application model editor

the EPR of the *OrderManager* Web service.

7.4.3 Provisioning

Having exported the sample application as a Car archive as shown in Figure 7.6, the archive can now be uploaded to the Cafe portal. Once it has been uploaded, tenants can subscribe to the sample application. When they subscribe, at first they are prompted for the solution engineering variability points, then the components are provisioned. In case no tenant has yet subscribed all nine components must be provisioned. In the following it is assumed that the variability has been bound in a way such that the application needs to

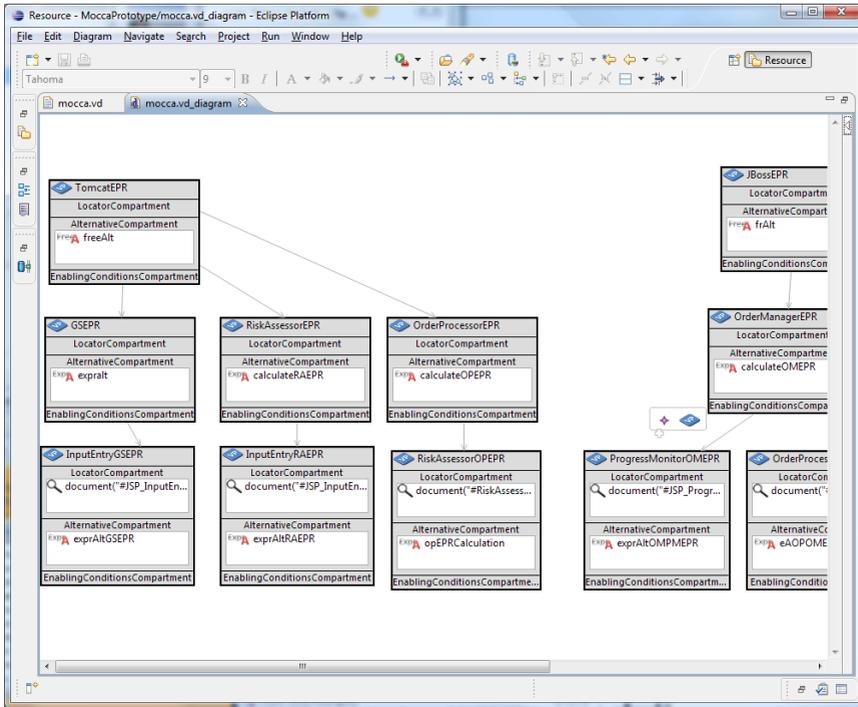


Figure 7.5: Variability of the sample application in the variability model editor

be public and thus must be provisioned on EC2. Thus, at the first, the *JBoss* and the *Tomcat with Ode* components are provisioned in parallel on Amazon EC2 by calling the respective component flows. Then the *OrderManager* and *ItemManager* Web services are deployed on the *JBoss* component. In parallel to that, the *GoodStandingVerifier* Web service is deployed on the *Tomcat with Ode* component. Then, the two BPEL processes, the *OrderProcessor* and *RiskAssessor* components are configured and deployed on the *Tomcat with Ode* component. Having deployed the two processes, the GUI components (*InputEntry* and *ProgressMonitor*) are configured and deployed on the *Tomcat with Ode* component. On average, the configuration and provisioning of the components takes about 7 minutes, depending on the performance of EC2 and the network speed to

upload the components to an EC2 instance.

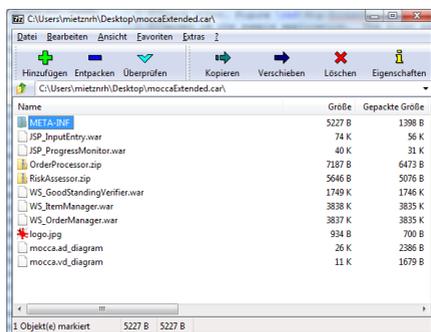


Figure 7.6: Ingredients of the sample application Car archive's root folder

7.5 Bootstrapping Cafe: Cafe System as a Cafe Application

In this case study the implementation of the Cafe system as described in Section 3.3.5 is seen as a Cafe application template itself. The Cafe system part of Figure 6.1 shows the components of the Cafe system application template (*the template* from now on). All components shown in Figure 6.1 are internal components, i.e. the corresponding implementation is given along with the template. Only the *Apache Tomcat*, *Active BPEL* and *MySQL DB* components are provider supplied. In these cases the component name shown in Figure 6.1 is the component type of the provider supplied component. All components of the Cafe system are multiple instances components, thus when deploying a new Cafe system all components need to be provisioned.

Regarding the variability in the template: All components have a variability point *URL* that denotes the URL under which they are available, these variability points are running variability points, i.e. only become available at runtime of the respective component. Components that interact with other components also have a variability point that depends on the URL variability point of the other components and uses the selected value of that variability point to configure the respective component.

The provisioning order for the Cafe system is then the following: At first the middleware components are provisioned in parallel. Then the databases are deployed on top of the *MySQL DBMS* component as they only have incoming dependencies. Once they have been deployed, the components representing the provisioning support services, the solution creation service and the selection facility are provisioned in parallel. Once these components are provisioned, the components representing the optimization component and the generic provisioning flow are deployed as they depend on these services. Then, the *Web portal* component is deployed as it must first be configured with the URLs of all other services and the generic provisioning flow. Component flows and customization flows are uploaded and deployed once a template is uploaded or a component flow is registered as they are not part of the Cafe system but are provided by the application vendors. Note that there must be one Cafe system manually installed before other Cafe systems can be automatically provisioned.

7.6 Further Case Studies

In addition to the case studies described above, further case studies have been conducted to evaluate the applicability of Cafe in different settings. We only sketch them briefly.

7.6.1 EaaS

The EAI as a Service (EaaS) [SML08, SML09] case study shows the applicability of Cafe in an enterprise application integration scenario. In EaaS, EAI patterns [HW03] are used in a Web-based modeling tool to model complex integration scenarios. These complex integration scenarios are then transformed into executable BPEL workflow models that orchestrate a set of pre-defined configurable Web services [SL08, SL09]. These Web services realize the integration logic formalized by the patterns. Cafe functionality is used in this scenario to describe the complex customized customer-specific application solution that can be generated from the model formalized by the EAI patterns. This customer-specific application solution can then be provisioned with the

Cafe provisioning environment. Since the set of Web services that realize the EAI patterns is predefined (and thus the amount of different components that can be used in applications is predefined too) lots of services can be reused in single instance and single configurable instance patterns for different EAI scenarios.

7.6.2 Sametime 3D in the Cloud

In the “Sametime 3D in the Cloud” case study the Sametime 3D application has been transferred to a cloud environment. Sametime 3D [Cor09b] offers a virtual meeting environment based on IBM’s Sametime Instant Messaging. Two components, an *IBM WebSphere CE Application Server* component and an *Open SIM Virtual World Server* component have been defined. These component are implemented by Amazon EC2 Amazon Machine Images (AMIs). On top of these components the *Sametime 3D Web Application* and the *Sametime 3D Virtual World* components can be deployed which are also modeled as Cafe components. An application template for the Sametime 3D application then contains the *Sametime 3D Web Application* as single instance and the virtual world components as multiple instance components. The middleware components are of implementation type *provider supplied*. Upon request of a new meeting room by a customer it is evaluated if an existing application deployment can provide another meeting room. If this is the case, a new *Virtual World component is deployed* and made known to the corresponding instance of the Web application where users can then sign up for the meeting. In case none of the existing virtual world servers can accommodate additional virtual worlds, a new *Open SIM Virtual World Server* component is started. This case study is interesting as it contains components that can accommodate a very different amount of concurrent customers. While the Web application can accommodate about 100 customers (with one active meeting room each) the Open SIM server can accommodate about 6 customers (with one active meeting room each). Since the Cafe provisioning flows can take these different capabilities into account only those components are provisioned that are really needed.

CONCLUSION AND OUTLOOK

8.1 Conclusions

In this thesis a method and implementation to develop and automatically provision customizable composite applications and the required infrastructure to run them at different (cloud) providers are presented. This thesis contains six main contributions. The first contribution is a formal definition and serialization format for an application metamodel enabling application vendors to define arbitrary application templates in terms of their components and the deployment relations among these components. In contrast to existing application metamodels, components can be implemented in any known programming language. An application model derived from the application metamodel can contain internal components whose implementation is shipped with the application as well as external components that are run externally from the application and provider supplied components which must be provided by the provider. The second contribution is a package format to exchange application templates between application vendors and providers that relies on the application metamodel above. This composite application archive (Car) package format and the application metamodel are independent of the implementa-

tion technologies used for the components in an application. They enable the exchange of application templates across different application vendors and providers. The third contribution is a formal variability metamodel and serialization format that describes the variability in an application template. Variability can be attached to any component in the application template and individual variability points can be connected via complex dependencies. Thus the variability metamodel allows the definition of complex cross-component variabilities. Using the locator concept, variability can target components implemented in any programming language. The fourth contribution are algorithms and tools to transform the variability model of an application into multiple customization flows that are used for various purposes. These use-cases include the guidance of a customer through the customization of an application template into an application solution, as well as the evaluation of realization components for a provider supplied component. The presented algorithms guarantee that all customizations performed with the generated tooling can only produce correct and complete customizations. The correct and complete customization of application templates is fundamental for the ability to later execute the provisioned application. The fifth contribution are an architecture and tools to automatically provision the customized application solutions across multiple providers. This architecture includes a definition of a generic interface for arbitrary component flows that abstract from the concrete provisioning engines used to provision components. This abstraction enables the inclusion of arbitrary (cloud) providers and local data centers, managed by different provisioning engines, into the overall system. It is shown how the ordering of the provisioning actions can be derived from the application model and variability model of an application. By automatically deriving the necessary provisioning actions and configuration steps from the models, the development effort for composite applications can be reduced, as application vendors do not need to specify explicitly how an application needs to be deployed at different providers. The sixth main contribution are requirement aware provisioning flows that are generated from the variability model and the application model of an application solution. These provisioning flows take the requirements for provider supplied components (including QoS and

functional requirements) into account by making use of the evaluation flows that evaluate if the customization of a possible realization component binds the variability of the template in a way that guarantees the required functional and non-functional requirements.

Besides these main contributions, a high-level development process for Cafe is defined that takes the specifics of Cafe applications into account, such as the definition of variability and the template customization and solution engineering phases. Corresponding prototypes to support the entire development process are presented in the implementation chapter. To evaluate the general applicability of the approach, it is compared with existing approaches that realize a subset of the functionality of Cafe. It has been shown that the main functionality of these approaches is also provided by Cafe. A set of case studies has been presented that further show the general applicability of the approach.

8.2 Future Work

As indicated in the respective sections, the general approach taken by Cafe enables application vendors, providers and customers, to model, provision and customize a variety of different applications. However, several limitations with the current approach exist that leave room for further research efforts. First of all, the development process for Cafe application templates as described in this thesis is geared towards the development of Cafe application templates from scratch. Extensions to this process could be investigated that describe how existing architecture modeling methodologies and diagrams such as ACME or UML component and deployment diagrams can be transformed into Cafe application models and variability models. Furthermore, a more detailed development process is needed that not only takes into account the development of the application but also related phases such as requirements engineering and testing. Especially the testing phase can benefit from the Cafe functionality, as test environments can be set up automatically and rapidly using the Cafe tools. In software product line engineering the testing of members of product lines is a separate discipline [PBvdL05]. Given a large set of variability points for an application, a high number of different customizations can be derived.

Testing all combinations is not always possible, therefore more research needs to go into how such applications can be tested. One approach would be to test solutions “on demand” with predefined test data after the provisioning has taken place and before the application is “delivered” to the customer.

Another point for future work are the Cafe application metamodel and Cafe variability metamodel. These are primarily focused on describing the components and variability that is needed to set up and execute an application. In addition to that, the models can be extended by other aspects such as security that is of imminent importance in cloud scenarios or the handling of data. The Cafe application metamodel and the variability metamodel thus form the basis of further research in various directions. While the customization flows guarantee the correctness and completeness of customizations, validation algorithms are needed that check whether the variability model of an application itself does not corrupt the artifacts it targets. Another very important aspect to be researched is the movement of data from one Cafe application to another or from on-premise data centers into the cloud. Since Cafe allows the provider-independent definition of application templates and thus a customer can theoretically switch to the same application at another provider, extensions are needed that describe how data can be transferred between different providers. Several approaches can be tackled here: one promising approach would be to extend the CPMI with operations that can retrieve and insert data from or into a component. For example, a DBMS or workflow engine component could offer a `retrieveData` operation that allows exporting all data in the database or the process instance data. This data could then be imported into another similar component via this component’s `insertData` operation. These two operations could then be mapped to concrete middleware operations via the component flows.

Since application models make use of components that are of a specific type, the inclusion of component types is crucial for the acceptance Cafe platform. In this thesis the inclusion of some component types at a limited set of providers for the case studies has been shown. Especially for single configurable instance components the development of multi-tenant aware middleware components, such as BPEL engines etc. must be investigated. In future work the integration

of systems management functionality, for example, from CMDBs or other tools that keep track on the resources in a data center can be investigated.

One drawback of the generation of WS-BPEL4People customization flows from the variability model, are the limited possibilities regarding human interactions of WS-BPEL. For example, with standard WS-BPEL4People it is not possible to redo tasks that have completed. In [VdAWG05] the concept of case handling is described that allows knowledge workers more flexibility when dealing with human-centric workflows (also called *cases*), including the redo of tasks. Employing case handling concepts for the execution of solution-engineering customization flows is a topic that can be investigated to make the solution engineering more flexible for customers while retaining the guarantees that only complete and correct customizations can be made. A first step towards this future work has been performed in [Naa10] where cases are generated from variability models.

As described in Section 5.4 multiple techniques from capacity planning and different optimization techniques can be exploited to optimize the selection of realization components and realization provisioning services. As a first step in [Feh09, FLM10], different optimization algorithms are investigated that can be used to optimize the resource allocation across multiple customers at a provider. In [Tru10] the finding of the optimal component binding for a solution is formulated as a constraint satisfaction problem and multiple approaches to generate CSPs from an application model are investigated. Further approaches can be investigated here, including approaches that do optimize not only based on the already deployed applications but also based on predictions that can be derived from mined data of already deployed solutions. To be able to perform meaningful optimizations the Cafe application metamodel must possibly be extended with communication and data dependency links between components. These links can then be annotated with parameters, for example, to optimize the distribution of components across different machines or providers. Here techniques known from the optimal stratification of transactions [DKL09] can be reused.

Regarding the Cafe provisioning and management interface CPMI as described in Section 5.2, this interface is mainly geared at the provisioning and

de-provisioning of components. Management aspects are only implicitly taken care of, for example, through the get and set operations for properties. However, there are no operations yet, for example, to attach monitoring tools to such components, thus the CPMI can be extended to include these management facilities. Other management functionality such as the elastic growing and shrinking of components via dynamic provisioning techniques can be investigated and included in Cafe. It can be investigated how systems management processes, such as ITIL processes can be supplied with applications, describing how applications can be managed and modified during runtime.

BIBLIOGRAPHY

- [3te09] 3tera inc. Cloud computing for web applications. <http://www.3tera.com>, 2009.
- [AAY97] J. M. Almeida, V. Almeida, and D.J. Yates. Measuring the behavior of a world-wide web server. In *HPN '97: Proceedings of the IFIP TC6 seventh international conference on High performance networking VII*, pages 57–72, London, UK, UK, 1997. Chapman & Hall, Ltd.
- [Act09] Active Endpoints. ActiveBPEL open source engine. <http://www.activevos.com/community-open-source.php>, 2009.
- [AEK⁺07] William Arnold, Tamar Eilam, Michael Kalantar, Alexander V. Konstantinou, and A. A. Totok. Pattern based soa deployment. In *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*, pages 1–12, Berlin, Heidelberg, 2007. Springer-Verlag.
- [AFG⁺09] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, et al. Above

the clouds: A berkeley view of cloud computing. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

- [AGJ⁺08] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1195–1206. ACM New York, NY, USA, 2008.
- [Agr07a] Agrawal, A. et al. Web Services Human Task, June 2007.
- [Agr07b] Agrawal, A. et al. WS-BPEL Extension for People Specification, June 2007.
- [AKR01] M. Arlitt, D. Krishnamurthy, and J. Rolia. Characterizing the scalability of a large web-based shopping system. *ACM Transactions on Internet Technology*, 1(1):44–69, 2001.
- [ALMS09] T. Anstett, F. Leymann, R. Mietzner, and S. Strauch. Towards BPEL in the Cloud: Exploiting Different Delivery Models for the Execution of Business Processes. In *Proceedings of the International Workshop on Cloud Services (IWCS2009) in conjunction with the 7th IEEE International Conference on Web Services (ICWS 2009), Los Angeles, CA, USA, July 10, 2009*. IEEE, 2009.
- [Ama] Amazon Inc. Amazon Simple Storage Service (Amazon S3). <http://aws.amazon.com/s3/>.
- [Ama09] Amazon Inc. Amazon Elastic Computing Cloud (EC2). <http://aws.amazon.com/ec2>, 2009.
- [App09] Appian. Appian Anywhere: BPM SaaS. <http://www.theprocessfactory.com/>, 2009.
- [Arn09] T. Arnold. Extension of a SCA Editor and Deployment-Strategies for Software as a Service Applications. Diploma

thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, February 2009.

- [Aue08] K. Auer. Generierung von WS-BPEL Prozessen aus Variabilitätsbeschreibungen. Diploma thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, May 2008.
- [BCE⁺02] T. Bellwood, L. Clement, D. Ehnebuske, A. Hately, M. Hondo, Y.L. Husband, K. Januszewski, S. Lee, B. McKee, J. Munter, et al. UDDI Version 3.0. *Published specification, Oasis*, 2002.
- [BDS05] B. Benatallah, M. Dumas, and Q.Z. Sheng. Facilitating the rapid development and scalable orchestration of composite web services. *Distributed and Parallel Databases*, 17(1):5–37, 2005.
- [Bec99] Kent Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.
- [Ber96] P.A. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.
- [BFG⁺02] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J.H. Obbink, and K. Pohl. Variability issues in software product lines. *Lecture Notes in Computer Science*, pages 13–21, 2002.
- [BMK⁺01] G. Böckle, J.B. Muñoz, P. Knauber, C.W. Krueger, J.C.S. do Prado Leite, F. van der Linden, L. Northrop, M. Stark, and D.M. Weiss. Adopting and institutionalizing a product line culture. *Development*, 2001.
- [Boe88] BW Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.
- [Boe99] Barry Boehm. Managing software productivity and reuse. *Computer*, 32(9):111–113, 1999.

- [Bos00a] J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. Addison-Wesley Professional, 2000.
- [Bos00b] J. Bosch. Organizing for software product lines. *Software Architectures for Product Families*, pages 117–134, 2000.
- [Bos01] J. Bosch. Software product lines: organizational alternatives. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 91–100, Washington, DC, USA, 2001. IEEE Computer Society.
- [BPSM⁺00] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0. *W3C recommendation*, 6, 2000.
- [BSD03] B. Benatallah, Q.Z. Sheng, and M. Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48, 2003.
- [BTD⁺] J. Balasubramanian, S. Tambe, B. Dasarathy, S. Gadgil, F. Porter, A. Gokhale, and D.C. Schmidt. Netqope: A model-driven network qos provisioning engine for distributed real-time and embedded systems. In *RTAS'08: Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 113–122. Citeseer.
- [But06] D. Butler. Amazon puts network power online. *Nature*, 444(7119):528, 2006.
- [BW96] AW Brown and KC Wallnan. Engineering of component-based systems. In *Second IEEE International Conference on Engineering of Complex Computer Systems, 1996. Proceedings.*, pages 414–422, 1996.
- [BYV⁺09] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype,

and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

- [CA07] A. Caracas and J. Altmann. A pricing information service for grid computing. In *Proceedings of the 5th international workshop on Middleware for grid computing: held at the ACM/I-FIP/USENIX 8th International Middleware Conference*, page 4. ACM, 2007.
- [Car04] N.G. Carr. IT doesn't matter. *IEEE Engineering Management Review*, 32(1):24–32, 2004.
- [CC06] F. Chong and G. Carraro. Architecture strategies for catching the long tail. *MSDN Library, Microsoft Corporation*, 2006.
- [CD01] J. Cheesman and J. Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley, Boston, 2001.
- [CDK⁺02] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet computing*, pages 86–93, 2002.
- [CDPEV05] F. Canfora, M. Di Penta, R. Esposito, and M.L. Villani. An approach for qos-aware service composition based on genetic algorithms. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1069–1075, New York, NY, USA, 2005. ACM.
- [CGRG06] B. Crutchfield George and D. Roach Gaut. Offshore outsourcing to india by u.s. and e.u. companies. *6 U.C. Davis Bus. L.J.* 13 (2006), 2006.
- [Cha99] J. Charles. Middleware moves to the forefront. *Computer*, 32(5):17–19, 1999.

- [Cha04] D. Chappell. *Enterprise Service Bus*. O'Reilly Media, Inc., 1st edition, June 2004.
- [Cha09] D. Chappell. *Introducing Windows Azure*. *DavidChappell & Associates White Paper*, 2009.
- [Cis09] Cisco Systems, Inc. WebEx: Web Conferencing. <http://www.webex.com>, 2009.
- [Ciu09] E. Ciurana. *Developing with Google App Engine*. *Apress Berkeley, CA, USA*, page 164, 2009.
- [Cle95] P.C. Clements. From subroutines to subsystems: Component-based software development. *American Programmer*, 8:31–31, 1995.
- [CLWK00] X. Cai, M.R. Lyu, K.F. Wong, and R. Ko. Component-based software engineering: technologies, development frameworks, and quality assurance schemes. In *Proc. Asia-Pacific Software Engineering Conf*, pages 372–379, 2000.
- [Cor09a] Cordys. Process Factory: BPM On Demand. <http://www.theprocessfactory.com/>, 2009.
- [Cor09b] IBM Corporation. Made in IBM Labs: Secure, 3D Meeting Service Now Available with Lotus Sametime. <http://www-03.ibm.com/press/us/en/pressrelease/27831.wss>, 2009.
- [CS01] WL Currie and P. Seltsikas. Exploring the supply-side of IT outsourcing: evaluating the emerging role of application service providers. *European Journal of Information Systems*, 10(3):123–134, 2001.
- [CSL03] A. Clemm, F. Shen, and V. Lee. Generic provisioning of heterogeneous services - a close encounter with service profiles. *Computer Networks*, 43(1):43–57, 2003.

- [CZZ⁺09] H. Cai, K. Zhang, M. Zhou, W. Gong, J. Cai, and Z. Mao. An end-to-end methodology and toolkit for fine granularity saas-ization. *Cloud Computing, IEEE International Conference on*, 0:101–108, 2009.
- [Dec09] DecidR Project Team. Decidr. <http://www.decidr.eu>, 2009.
- [Dew01] D.T. Dewire. Application Service Providers. *Making Supply Chain Management Work: Design, Implementation, Partnerships, Technology, and Profits*, page 467, 2001.
- [DFH⁺07] T. Dörnemann, T. Friese, S. Herdt, E. Juhnke, and B. Freisleben. Grid workflow modelling using grid-specific BPEL extensions. In *German e-Science Conference*, 2007.
- [Din08] X. Ding. Variability Points in WS-BPEL Prozessen. Diploma thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, April 2008.
- [DKL09] O. Danylevych, D. Karastoyanova, and F. Leymann. Optimal stratification of transactions. In *Proceedings of the fourth international conference on internet and web applications and services (ICIW 2009) IEEE Computer Society*, 2009.
- [DKM⁺09] M.D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis, and A. Vakali. Cloud computing: Distributed internet computing for it and scientific research. *IEEE Internet Computing*, 13(5):10–13, 2009.
- [DMT09] DMTF. Open Virtualization Format Specification Version 1.0.0. http://www.dmtf.org/standards/published_documents/DSP0243_1.0.0.pdf, 2009.
- [Dri09] Driver, M. Cloud Application Infrastructure Technologies Need Seven Years to Mature, 2009.
- [dRW04] J. des Rivières and J. Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

- [DW07] A. Dubey and D. Wagle. Delivering software as a service. *The McKinsey Quarterly*, pages 1–12, 2007.
- [EBC⁺05] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S.L. Price. Grid service orchestration using the business process execution language (BPEL). *Journal of Grid Computing*, 3(3):283–304, 2005.
- [Ecl] Eclipse Foundation. Eclipse graphical modeling framework (GMF). Online: <http://www.eclipse.org/gmf>.
- [Ela09] Elastra Corporation. Elastra. <http://www.elastra.com/>, 2009.
- [Emm00] W. Emmerich. Software engineering and middleware: a roadmap. In *Proceedings of the Conference on The future of Software engineering*, page 129. ACM, 2000.
- [EMME⁺06] K. El Maghraoui, A. Meghranjani, T. Eilam, M. Kalantar, and A.V. Konstantinou. Model driven provisioning: Bridging the gap between declarative object models and procedural provisioning tools. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, pages 404–423. Springer-Verlag New York, Inc. New York, NY, USA, 2006.
- [Erl05] T. Erl. *Service-oriented architecture: concepts, technology, and design*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.
- [ES06] A. El Saddik. Performance measurements of web services-based applications. *IEEE Transactions on Instrumentation and Measurement*, 55(5):1599, 2006.
- [Ete09] Etelos Inc. MySMBStore. <http://www.mysmbstore.com>, 2009.
- [Fag07] J.C. Fagan. Mashing up Multiple Web Feeds Using Yahoo! Pipes. *Computers in Libraries*, 27(10):8, 2007.

- [Feh09] C. Fehling. Provisioning of Software as a Service Applications in the Cloud. Diploma thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, July 2009.
- [Fel03] C. Fellenstein. *On Demand Computing: Technologies and Strategies, First Edition*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
- [FFB02] D. Fey, R. Fajta, and A. Boros. Feature modeling: A meta-model to enhance usability and usefulness. *Lecture Notes in Computer Science*, 2379:198–216, 2002.
- [FK04] I. Foster and C. Kesselman. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann, 2004.
- [FKT01] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200, 2001.
- [FLM10] C. Fehling, F. Leymann, and R. Mietzner. A Framework for Optimized Distribution of Tenants in Cloud Applications. In *Proceedings of the 2010 IEEE International Conference on Cloud Computing (CLOUD 2010)*. IEEE, Juli 2010.
- [Fow02] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.
- [FYL⁺08] S.S. Fu, J. Yang, J. Laredo, Y. Huang, H. Chang, S. Kumaran, J.Y. Chung, and Y. Kosov. Solution Templates Tool for Enterprise Business Applications Integration. In *Proceedings of the 2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (suc 2008)-Volume 00*, pages 314–319. IEEE Computer Society, 2008.

- [FZRL08] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10, 2008.
- [Gee08] J. Geelan. Twenty one experts define cloud computing. *Virtualization, August*, 2008.
- [Gen09] Generic Objects Technologies. Webservicex.net. <http://www.webservicex.net>, 2009.
- [GGS07] D. Gerlach, N. Gueven, and D. Schleicher. Vergleich von Provisioning-Tools, Mai 2007.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: elements of reusable object-oriented software. *Addison-Wesley Professional Computing Series*, 1995.
- [GM00] H. Goma and D.A. Menascé. Design and performance modeling of component interconnection patterns for distributed software architectures. In *Proceedings of the 2nd international workshop on Software and performance*, pages 117–126. ACM New York, NY, USA, 2000.
- [GM02] D. Greschler and T. Mangan. Networking lessons in delivering Software as a Service—part I. *International Journal of Network Management*, 12(5):317–321, 2002.
- [GMW00] D. Garlan, R.T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. *Foundations of component-based systems*, pages 47–68, 2000.
- [Goo09a] Google. Google Apps. <http://www.google.com/apps>, 2009.
- [Goo09b] Google. Google Mashup Editor. <http://code.google.com/gme>, 2009.
- [Gri08] E. Griffin. *Foundations of Popfly: Rapid Mashup Development*. Springer, 2008.

- [GSH⁺07] C.J. Guo, W. Sun, Y. Huang, Z.H. Wang, B. Gao, and B. IBM. A framework for native multi-tenancy application development and management. In *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on*, pages 551–558, 2007.
- [Gue08] N. Gueven. Variability Patterns for WS-BPEL. Diploma thesis, October 2008.
- [GWJV⁺09] F. Gottschalk, T.A.C. Wagemakers, M.H. Jansen-Vullers, W.M.P. van der Aalst, and M. La Rosa. Configurable process models: experiences from a municipality case study. In *Proceedings of the 21st international conference on advanced information systems engineering (CAiSEŠ09)*. Springer, Heidelberg. Springer, 2009.
- [Ham97] G. Hamilton. JavaBeans. *Sun Microsystems*, 1997.
- [Ham07] J. Hamilton. On designing and deploying internet-scale services. In *Proceedings of the 21st conference on Large Installation System Administration Conference table of contents*. USENIX Association Berkeley, CA, USA, 2007.
- [HAY08] B. HAYES. Cloud Computing. *Communications of the ACM*, 51(7):9–11, 2008.
- [HBR08] A. Hallerbach, T. Bauer, and M. Reichert. Managing Process Variants in the Process Life Cycle. In *Proc. 10th Int. Conf. on Enterprise Information Systems*, pages 154–161. Citeseer, 2008.
- [HC01] G.T. Heineman and W.T. Council. *Component-based software engineering: putting the pieces together*. Addison-Wesley USA, 2001.

- [HH00] M. Hancox and R. Hackney. IT outsourcing: frameworks for conceptualizing practice and perception. *Information Systems Journal*, 10(3):217–237, 2000.
- [HKSH04] J. Hutchinson, G. Kotonya, I. Sommerville, and S. Hall. A service model for component-based development. In *Proceedings of the 30th EUROMICRO Conference*, pages 162–169. Citeseer, 2004.
- [HLW94] R. Hirschheim, M. Lacity, and L. Willcocks. Realising Outsourcing Expectations: Incredible Expectations, Credible Outcomes. *Information Systems Management*, 11(4):7–18, 1994.
- [HP03] G. Halmans and K. Pohl. Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, 2(1):15–36, 2003.
- [Hum06] J. Humphreys. System Virtualization: Sun Microsystems Enables Choice, Flexibility, and Management. *Sun Microsystems*, 2006.
- [HW03] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, October 2003.
- [IBM09] IBM Corp. IBM Dynamic Infrastructure for SAP Business Suite. <http://www-03.ibm.com/solutions/sap/doc/content/landingdtw/1463476130.html>, 2009.
- [IBM10] IBM Corporation. IBM Tivoli Service Automation Manager. <http://www-01.ibm.com/software/tivoli/products/tsam-facts.html>, 2010.
- [Int09] Intalio. Intalio BPMS. <http://community.intalio.com/>, 2009.
- [JA07] D. Jacobs and S. Aulbach. Ruminations on multi-tenant databases. *BTW Proceedings*, 2007.

- [JB04] M. Jaring and J. Bosch. A taxonomy and hierarchy of variability dependencies in software product family engineering. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, pages 356–361, 2004.
- [JGJ97] I. Jacobson, M. Griss, and P. Jonsson. *Software reuse: architecture, process and organization for business success*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 1997.
- [JMF09] S. Jha, A. Merzky, and G. Fox. Using clouds to provide grids higher-levels of abstraction and explicit support for usage modes. *Concurrency and Computation: Practice and Experience*, 21(8):1087–1108, 2009.
- [JRG05] M.C. Jaeger, G. Rojec-Goldmann, and G. Muhl. QoS aggregation in Web service compositions. In *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05) on e-Technology, e-Commerce and e-Service*, pages 181–185. IEEE Computer Society Washington, DC, USA, 2005.
- [KA98] D. Krieger and R.M. Adler. The emergence of distributed component platforms. *IEEE computer*, 31(3):43–53, 1998.
- [KAB⁺04] M. Keen, A. Acharya, S. Bishop, A. Hopkins, S. Milinski, C. Nott, R. Robinson, J. Adams, and P. Verschueren. *Patterns: Implementing an SOA Using an Enterprise Service Bus*. 2004.
- [Kah01] P. Kahkipuro. UML-based performance modeling framework for component-based distributed systems. *Lecture Notes in Computer Science*, pages 167–184, 2001.
- [KB98] W. Kozaczynski and G. Booch. Component-based software engineering. *IEEE software*, 15(5):34–36, 1998.

- [KB04] A. Keller and R. Badonnel. Automating the Provisioning of Application Services with the BPEL4WS Workflow Language. *Proc. DSOM 2004*, 2004.
- [KC03] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, pages 41–50, 2003.
- [KEK⁺09] A. V. Konstantinou, T. Eilam, M. Kalantar, A. A. Totok, W. Arnold, and E. Snible. An architecture for virtual solution composition and deployment in infrastructure clouds. In *VTDC '09: Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing*, pages 9–18, New York, NY, USA, 2009. ACM.
- [KK04] T. Kichkaylo and V. Karamcheti. Optimal resource-aware deployment planning for component-based distributed applications. In *Proceedings of the Thirteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, pages 150–159, 2004.
- [KKL⁺98] K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-; oriented reuse method with domain-; specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.
- [KLD02] K.C. Kang, J. Lee, and P. Donohoe. Feature-oriented product line engineering. *IEEE software*, pages 58–65, 2002.
- [Kob00] C. Kobryn. Modeling components and frameworks with UML. 2000.
- [Kru99] P. Kruchten. *The Rational Unified Process*. Addison-Wesley, 1999.
- [KSBB08] S. Kounev, K. Sachs, J. Bacon, and A. Buchmann. A methodology for performance modeling of distributed event-based

- systems. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:13–22, 2008.
- [KSSA09] M. Koning, C. Sun, M. Sinnema, and P. Avgeriou. VxBPEL: Supporting variability for Web services in BPEL. *Information and Software Technology*, 51(2):258–269, 2009.
- [Kum92] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI magazine*, 13(1):32, 1992.
- [LA94] F. Leymann and W. Altenhuber. Managing business processes as an information resource. *IBM Systems Journal*, 33(2):326–348, 1994.
- [Lau06] K.K. Lau. Software component models. In *Proceedings of the 28th international conference on Software engineering*, page 1082. ACM, 2006.
- [Law08] G. Lawton. Developing Software Online With Platform-as-a-Service Technology. *Computer*, 41(6):13–15, 2008.
- [LDF⁺10] F. Leymann, S. Dustdar, C. Fehling, R. Mietzner, and A. Nowak. Moving applications to the cloud: An approach based on application diagrams enrichment. *Int. J. Cooperative Inf. Syst.*, (under submission), 2010.
- [Ley06] F. Leymann. Choreography for the Grid: towards fitting BPEL to the resource framework. *Concurrency and Computation: Practice and Experience*, 18(10), 2006.
- [Ley09] F. Leymann. Cloud Computing: The Next Revolution in IT. In *Proc. 52th Photogrammetric Week*, pages 1–10. Online, September 2009.
- [LFG04] Y. Liu, A. Fekete, and I. Gorton. Predicting the performance of middleware-based applications at the design level. *ACM SIGSOFT Software Engineering Notes*, 29(1):166–170, 2004.

- [LGDK05] H. Ludwig, H. Gimpel, A. Dan, and B. Kearney. Template-based automated service provisioning-supporting the agreement-driven service life-cycle. *Lecture notes in computer science*, 3826:283, 2005.
- [LKN⁺09] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm. What's Inside the Cloud? An Architectural Map of the Cloud Landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 23–31. IEEE Computer Society, 2009.
- [LL07] A. Lazovik and H. Ludwig. Managing Process Customizability and Customization: Model, Language and Process. In *Web Information Systems Engineering WISE 2007: 8th International Conference on Web Information Systems Engineering, Nancy, France, December 3-7, 2007: Proceedings*, page 373. Springer, 2007.
- [LR00] F. Leymann and D. Roller. *Production Workflow Concepts and Techniques*. Prentice-Hall, 2000.
- [LRLS⁺07] M. La Rosa, J. Lux, S. Seidel, M. Dumas, and A.H.M. ter Hofstede. Questionnaire-driven Configuration of Reference Process Models. In *Advanced Information Systems Engineering*, page 424. Springer, 2007.
- [LSW09] S. Luan, Y. Shi, and H. Wang. A Mechanism of Modeling and Verification for SaaS Customization Based on TLA. In *Web Information Systems and Mining: International Conference, Wism 2009, Shanghai, China, November 7-8, 2009, Proceedings*, page 337. Springer, 2009.
- [LW03] J. Lee and B. Ware. *Open source Web development with LAMP: using Linux, Apache, MySQL, Perl, and PHP*. Addison-Wesley Professional, 2003.

- [LW05] K.K. Lau and Z. Wang. A taxonomy of software component models. In *Proceedings of the 31st EUROMICRO Conference*, pages 88–95. Citeseer, 2005.
- [LYC⁺98] S.D. Lee, Y.J. Yang, E.S. Cho, S.D. Kim, and S.Y. Rhew. COMO: A UML-based component development methodology. In *Proc. of the 6th Asia Pacific Software Engineering Conf. Takamatsu: IEEE Computer Society Press*, volume 63, 1998.
- [LYJP09] J. Laredo, J. Yang, J.J. Jeng, and G. Perez. SIMPLE: Template Based Service Integration. In *Proceedings of the 2009 Congress on Services-I-Volume 00*, pages 465–466. IEEE Computer Society, 2009.
- [MA01] D.A. Menasce and V. Almeida. *Capacity Planning for Web Services: metrics, models, and methods*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2001.
- [Ma07] D. Ma. The Business Model of Software-As-A-Service. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 701–702, 2007.
- [MAFM99] D.A. Menascé, V. Almeida, R. Fonseca, and M.A. Mendes. A methodology for workload characterization of e-commerce sites. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 119–128. ACM New York, NY, USA, 1999.
- [MCO07] M. Mendonca, D. Cowan, and T. Oliveira. A Process-Centric Approach for Coordinating Product Configuration Decisions. In *Proc. HICSS 2007*, 2007.
- [MEG⁺03] E. Merks, R. Eliersick, T. Grose, F. Budinsky, and D. Steinberg. *The Eclipse Modeling Framework*, 2003.
- [Mer06] D. Merrill. Mashups: The new breed of Web app. *IBM Web Architecture Technical Library*, 2006.

- [MHP⁺07] A. Metzger, P. Heymans, K. Pohl, P.Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Requirements Engineering Conference, 2007. REŠ07. 15th IEEE International*, pages 243–253, 2007.
- [Mic09] Microsoft. Windows Azure Platform. <http://www.microsoft.com/windowsazure/>, 2009.
- [Mie08] R. Mietzner. Using variability descriptors to describe customizable SaaS application templates. Technical report, Technical Report 2008/01, Fakultät Informatik, Universität Stuttgart, 2008.
- [Mie10a] R. Mietzner. Cafe application metamodel schema. <http://www.cloudy-apps.com/schemas/ad.xsd>, 2010.
- [Mie10b] R. Mietzner. Cafe variability metamodel schema. <http://www.cloudy-apps.com/schemas/vd.xsd>, 2010.
- [MKL09] R. Mietzner, D. Karastoyanova, and F. Leymann. Business Grid: Combining Web Services and the Grid. *Lecture Notes In Computer Science*, pages 136–151, 2009.
- [ML97] M.H. Meyer and A.P. Lehnerd. *The power of product platforms: building value and cost leadership*. Free Press, New York, 1997.
- [ML08a] R. Mietzner and F. Leymann. Generation of BPEL Customization Processes for SaaS Applications from Variability Descriptors. In *Proceedings of the International Conference on Services Computing, Industry Track, SCC 2008*. IEEE, 2008.
- [ML08b] R. Mietzner and F. Leymann. Towards provisioning the cloud: On the usage of multi-granularity flows and services to realize a unified provisioning infrastructure for saas applications. In

SERVICES '08: Proceedings of the 2008 IEEE Congress on Services - Part I, pages 3–10, Washington, DC, USA, 2008. IEEE Computer Society.

- [MLP08] R. Mietzner, F. Leymann, and M.P. Papazoglou. Defining composite configurable saas application packages using sca, variability descriptors and multi-tenancy patterns. In *ICIW '08: Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services*, pages 156–161, Washington, DC, USA, 2008. IEEE Computer Society.
- [MLU10] R. Mietzner, F. Leymann, and T. Unger. Horizontal and Vertical Combination of Multi-Tenancy Patterns in Service-Oriented Applications. *Enterprise Information Systems - 13th International IEEE EDOC Enterprise Computing Conference (EDOC 2009)*, 4(3):1–18, Juli 2010.
- [MLW⁺09] R. Mietzner, T. Lessen, A. Wiese, M. Wieland, D. Karastoyanova, and F. Leymann. Virtualizing Services and Resources with ProBus: The WS-Policy-Aware Service and Resource Bus. In *Proceedings of the 2009 IEEE International Conference on Web Services-Volume 00*, pages 617–624. IEEE Computer Society, 2009.
- [MMB95] K. McLellan, B.L. Marcolin, and P.W. Beamish. Financial and strategic motivations behind IS outsourcing. *Journal of Information Technology*, 10(4):299–321, 1995.
- [MML08] R. Mietzner, Z. Ma, and F. Leymann. An algorithm for the validation of executable completions of an abstract bpel process. In Martin Bichler, Thomas Hess, Helmut Krcmar, Ulrike Lechner, Florian Matthes, Arnold Picot, Benjamin Speitkamp, and Petra Wolf, editors, *Multikonferenz Wirtschaftsinformatik*. GITO-Verlag, Berlin, 2008.

- [MMLP09] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *PESOS '09: Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 18–25, Washington, DC, USA, 2009. IEEE Computer Society.
- [MO05] V. Massol and T. O'Brien. *Maven: a developer's notebook*. O'Reilly Media, Inc., 2005.
- [MS05] M. Mernik and A.M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.
- [MSB04] Z. Maamar, Q.Z. Sheng, and B. Benatallah. On composite web services provisioning in an environment of fixed and mobile computing resources. *Information Technology and Management*, 5(3):251–270, 2004.
- [MSB⁺07] H. Maddurt, SB Shi, R. Baker, N. Ayachitula, L. Shwartz, M. Surendra, C. Corley, M. Benantar, and S. Patel. A configuration management database architecture in support of IBM Service Management. *IBM Systems Journal*, 46(3):441, 2007.
- [MUTL09] R. Mietzner, T. Unger, R. Titze, and F. Leymann. Combining Different Multi-Tenancy Patterns in Service-Oriented Applications. In *Proceedings of the 13th IEEE Enterprise Distributed Object Conference (EDOC 2009)*. IEEE, 2009.
- [Naa10] S. Naatz. Customization of Cloud Applications with a Case Handling System. Diploma thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, April 2010.

- [Nit09] Nitu. Configurability in saas (software as a service) applications. In *ISEC '09: Proceeding of the 2nd annual conference on India software engineering conference*, pages 19–26, New York, NY, USA, 2009. ACM.
- [NWG⁺09] D. Nurmi, R. Wolski, C. Grzegorzczuk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid-Volume 00*, pages 124–131. IEEE Computer Society, 2009.
- [OAS07a] OASIS. Web Services Business Process Execution Language Version 2.0 Standard, 2007.
- [OAS07b] OASIS. Web Services Resource Framework (WSRF) TC. www.oasis-open.org/committees/wsrf/, 2007.
- [OMG05a] OMG. Reusable Asset Specification Version 2.2, 2005.
- [OMG05b] OMG. UML 2.0 specification. <http://www.omg.org/technology/documents/formal/uml.htm>, 2005.
- [OMG09] OMG. BPMN Version 2.0 - Beta 1. <http://www.omg.org/spec/BPMN/2.0/>, 2009.
- [Ope07] Open SOA Collaboration (OSOA). SCA Service Component Architecture, Assembly Model Specification Version 1.00. http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf, 2007.
- [Ope09] Open Cloud Manifesto Supporters. Open cloud manifesto. <http://www.opencloudmanifesto.org>, 2009.
- [Ope10a] Open Source Matters, Inc. Joomla! open source content management system. <http://www.joomla.org/>, 2010.

- [Ope10b] OpenQRM. Open Source Systems Management Solution. <http://www.openqrm.org/>, 2010.
- [Pap03] M.P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering*, volume 10. NW Washington: IEEE Computer Society, 2003.
- [Par66] D.F. Parkhill. *The challenge of the computer utility*. Addison-Wesley Pub. Co., 1966.
- [Par72a] DL Parnas. A technique for software module specification with examples. 1972.
- [Par72b] DL Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1058, 1972.
- [Par76] DL Parnas. On the Design and Development of Program Families. *Transactions on Software Engineering*, pages 1–9, 1976.
- [PBvdL05] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.
- [Pec07] J. Pechanec. How the SCP protocol works. http://blogs.sun.com/janp/entry/how_the_scp_protocol_works, 2007.
- [Pel03] C. Peltz. Web services orchestration and choreography. *Computer*, pages 46–52, 2003.
- [PG03] M.P. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Communications of the ACM*, 46(10):25–28, 2003.
- [PM06] K. Pohl and A. Metzger. Variability management in software product line engineering. In *Proceedings of the 28th international conference on Software engineering*, page 1050. ACM, 2006.

- [PS98] F. Plášil and M. Stal. An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM. *Software-Concepts & Tools*, 19(1):14–28, 1998.
- [R⁺05] B. Roehm et al. WebSphere Application Server V6 Scalability and Performance Handbook, IBM Redbook, 2005.
- [Rap04] M.A. Rappa. The utility business model and the future of computing services. *IBM Systems Journal*, 43(1):32–42, 2004.
- [RBL⁺09] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, et al. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4), 2009.
- [Rig09] Rightscale Inc. Rightscale. <http://www.rightscale.com>, 2009.
- [RLM⁺09] F. Rosenberg, P. Leitner, A. Michlmayr, P. Celikovic, and S. Dustdar. Towards composition as a service - a quality of service driven approach. *Data Engineering, International Conference on*, 0:1733–1740, 2009.
- [Roy87] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.
- [RRVDAM06] J. Recker, M. Rosemann, W. Van Der Aalst, and J. Mendling. On the Syntax of Reference Model Configuration-Transforming the C-EPC into Lawful EPC Models. *Lecture Notes in Computer Science*, 3812:497, 2006.
- [RS05] J. Rao and X. Su. A survey of automated web service composition methods. *Lecture Notes in Computer Science*, 3387:43–54, 2005.

- [Run09] RunMyProcess. On-demand business integration. <http://www.runmyprocess.com/content/product-desc>, 2009.
- [RvdA07] M. Rosemann and W. M. P. van der Aalst. A configurable reference modelling language. *Inf. Syst.*, 32(1):1–23, 2007.
- [S⁺06] Q. Sheng et al. *Composite web services provisioning in dynamic environments*. PhD thesis, Awarded by: University of New South Wales. Computer Science and Engineering, 2006.
- [SA08] C. Sun and M. Aiello. Towards Variable Service Compositions Using VxBPEL. *Lecture Notes in Computer Science*, 5030:257, 2008.
- [SAB⁺07] L. Schwartz, N. Ayachitula, M. Bucu, G. Grabarnik, M. Surendra, C. Ward, S. Weinberger, and H. IBM. IT Service Provider’s Multi-Customer and Multi-Tenant Environments. In *E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on*, pages 559–566, 2007.
- [Sal09a] Salesforce.com. Trust.salesforce.com - system status. <http://trust.salesforce.com/trust/status/>, 2009.
- [Sal09b] Salesforce.com, Inc. Salesforce CRM. <http://www.salesforce.com>, 2009.
- [SAP09] SAP AG. SAP Business ByDesign: The Best of SAP, On Demand. <http://www.sap.com/sme/solutions/businessmanagement/businessbydesign>, 2009.
- [Sch95] K. Schwaber. Scrum Development Process. *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 1995)*, Austin, Texas, USA, pages 117–134, 1995.

- [Sch06] T. Schneider. *SAP Performance Optimization Guide*. Galileo Press, 2006.
- [Sch08] D. Schleicher. Service-orientiertes Provisioning fuer Software as a Service. Diploma thesis, Januar 2008.
- [SD07] M. Sinnema and S. Deelstra. Classifying variability modeling techniques. *Inf. Softw. Technol.*, 49(7):717–739, 2007.
- [Ses97] R. Sessions. *COM and DCOM: Microsoft's vision for distributed objects*. John Wiley & Sons, Inc. New York, NY, USA, 1997.
- [SH94] M.P. Singh and M.N. Huhns. Automating workflows for service provisioning: Integrating AI and database technologies. *IEEE Expert*, 9(5):19–23, 1994.
- [SH00] A.W. Scheer and F. Habermann. Making ERP a success. *Communications of the ACM*, 43(4):57–61, 2000.
- [Shi10] Q. Shi. Automatic Provisioning of Sametime 3D on a Cloud Infrastructure. Diploma thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, February 2010.
- [SL08] T. Scheibler and F. Leymann. A framework for executable enterprise application integration patterns. In *Fourth International Conference I-ESA*. Springer, 2008.
- [SL09] T. Scheibler and F. Leymann. From Modelling to Execution of Enterprise Integration Scenarios: the GENIUS tool. In *Proceedings of 16th Fachtagung Kommunikation in Verteilten Systemen (KiVS 09)*. Springer, March 2009.
- [Slo06] A. Slomiski. On using BPEL extensibility to implement OGSI and WSRF Grid workflows. *Concurrency and Computation: Practice and Experience*, 18(10):1229–1241, 2006.

- [SM07] C.R. Senna and E.R.M. Madeira. A middleware for instrument and service orchestration in computational grids. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2007), Rio de Janeiro, Brazil. IEEE Computer Society Press, Los Alamitos. Citeseer, 2007.*
- [SML08] T. Scheibler, R. Mietzner, and F. Leymann. EAI as a service-combining the power of executable EAI patterns and SaaS. In *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE*, pages 107–116, 2008.
- [SML09] T. Scheibler, R. Mietzner, and F. Leymann. EMod: platform independent modelling, description and enactment of parameterisable EAI patterns. *Enterprise Information Systems*, 3(3):299–317, 2009.
- [SMLF09] B. Sotomayor, R.S. Montero, I.M. Llorente, and I. Foster. Virtual Infrastructure Management in Private and Hybrid Clouds. *IEEE Internet Computing*, 13(5):14–22, 2009.
- [SP06] A. Schnieders and F. Puhlmann. Variability mechanisms in e-business process families. In *9th International Conference on Business Information Systems (BIS 2006)*, 2006.
- [SS05] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, page 84. USENIX Association, 2005.
- [ST05] M. Salehie and L. Tahvildari. Autonomic computing: emerging trends and open problems. In *Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, page 7. ACM, 2005.

- [Sun08] Sun Microsystems. Packaging Applications - The Java EE 5 Tutorial. <http://java.sun.com/javaee/5/docs/tutorial/doc/bnaby.html#indexterm-47>, 2008.
- [Sun09] Sun. Sun N1 Service Provisioning System. http://www.sun.com/software/products/service_provisioning/, 2009.
- [SWS07] SWSOft Inc. Application Packaging Standard (APS). <http://apsstandard.com/r/doc/package-format-specification-1.0.pdf>, 2007.
- [SZG⁺08] W. Sun, X. Zhang, C.J. Guo, P. Sun, and H. Su. Software as a service: Configuration and customization perspectives. In *IEEE Congress on Services Part II, 2008. SERVICES-2*, pages 18–25, 2008.
- [Tao01] L. Tao. Shifting paradigms with the application service provider model. *Computer*, 34(10):32–39, 2001.
- [TBB03] M. Turner, D. Budgen, and P. Brereton. Turning software into a service. *Computer*, 36(10):38–44, 2003.
- [Tit09] R. Titze. Web Service Deployment Strategien. Diploma thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, April 2009.
- [Tru10] I. Trummer. Cost-Optimal Provisioning of Cloud Applications. Diploma thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, February 2010.
- [Tsa93] E. Tsang. *Foundations of constraint satisfaction*. Academic New York, 1993.
- [UML09] T. Unger, R. Mietzner, and F. Leymann. Customer-defined Service Level Agreements for Composite Applications. *Enterprise*

Information Systems - Towards Model-driven Service-oriented Enterprise Computing *12th International IEEE EDOC Enterprise Computing Conference (EDOC 2008)*, 3(3):369–391, August 2009.

- [USC⁺08] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1):1–39, 2008.
- [V⁺97] S. Vinoski et al. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, 1997.
- [vdADG⁺08] W. M. P van der Aalst, M. Dumas, F. Gottschalk, A. H. M. ter Hofstede, M. La Rosa, and J. Mendling. Correctness-preserving configuration of business process models. In *In Proc. Fundamental Approaches to Software Engineering*, pages 46–61. Springer, 2008.
- [VDATHKB03] W.M.P Van Der Aalst, A.H.M Ter Hofstede, B. Kiepuszewski, and AP Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003.
- [VdAWG05] W.M.P Van der Aalst, M. Weske, and D. Grünbauer. Case handling: a new paradigm for business process support. *Data & Knowledge Engineering*, 53(2):129–162, 2005.
- [vdL02] F. van der Linden. Software Product Families in Europe: The Esaps & Café Projects. *IEEE Software*, 19(4), 2002.
- [vdMLA02] T. von der Maßen, H. Lichter, and R. Aachen. Modeling variability by UML use case diagrams. In *Proceedings of the International Workshop on Requirements Engineering for product lines*, pages 2002–033. Citeseer, 2002.
- [VGBS01] J. Van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Working IEEE/IFIP*

Conference on Software Architecture, 2001. Proceedings, pages 45–54, 2001.

- [VMw08] VMware, Inc. VMware ESX. <http://www.vmware.com/products/esx/>, 2008.
- [Vog08] W. Vogels. A Head in the Clouds - The Power of Infrastructure as a Service. In *First workshop on Cloud Computing and in Applications (CCAŠ08)*, 2008.
- [Vou08] M.A. Vouk. Cloud computing—Issues, research and implementations. *Journal of Computing and Information Technology*, 16(4):235–246, 2008.
- [VRMCL08] L.M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.
- [W3C99] W3C. XML Path Language, W3C Recommendation, 1999.
- [W3C01] W3C. Web Services Description Language (WSDL) 1.1, W3C Note, 2001.
- [W3C06] W3C. Web Services Policy 1.2 - Framework (WS-Policy), W3C Member Submission, 2006.
- [Wal02] C.A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36:181–194, 2002.
- [WB09] C.D. Weissman and S. Bobrowski. The design of the force.com multitenant internet application development platform. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 889–896. ACM, 2009.
- [WCL⁺05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D.F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-*

Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More. Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.

- [Wie08] A. Wiese. Konzeption und Implementierung von WS-Policy- und WSRF-Erweiterungen für einen Open Source Enterprise Service Bus. Diploma thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, February 2008.
- [Wil07] D.E. Williams. Virtualization with Xen: including XenEnterprise, XenServer, and XenExpress. 2007.
- [WL99] D.M. Weiss and C.T.R. Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.
- [YBCD08] J. Yu, B. Benatallah, F. Casati, and F. Daniel. Understanding mashup development. *IEEE Internet Computing*, 12(5):44–52, 2008.
- [YBDS08] L. Youseff, M. Butrico, and D. Da Silva. Toward a Unified Ontology of Cloud Computing. In *Grid Computing Environments Workshop, 2008. GCE08*, pages 1–10, 2008.
All links have last been accessed on July 10th 2010

LIST OF FIGURES

1.1	Cafe vision	19
2.1	Two customers of a software product line, vs. two customers of a Cafe application	34
2.2	Classification of as a service models	38
3.1	Ingredients of an application tenplate	60
3.2	Roles in Cafe	62
3.3	High-level application development process	64
3.4	Cafe application metamodel	68
3.5	Example application	69
3.6	Component with type	71
3.7	Component with associated implementation	73
3.8	Relations between implementation, files and building blocks	75
3.9	Single instance component icon	76
3.10	Single configurable instance component icon	78
3.11	Multiple instances component icon	81
3.12	Two components contained in a third component	85
3.13	Variability metamodel	93
3.14	Example variability model	94
3.15	Refining variability models	98
3.16	Phases during the provisioning of a component	101

3.17	Examples of an XPath and a properties file locator	105
3.18	Dependencies between components induced by variability	124
3.19	Shortcut notation for component dependencies induced by variability . .	125
3.20	Modeling requirements on provider supplied components	130
3.21	Shortcut notation for requirements on provider supplied components . .	132
3.22	Application requirements and provider capabilities	133
3.23	Entities of the provider metamodel	134
3.24	Example of a Cafe archive (Car file)	140
3.25	Steps and artifacts in template engineering	141
4.1	States of a variability point	154
4.2	Customization document	162
4.3	Mapping a variability point to activities and control connectors	172
4.4	Additional activities and control connectors created for enabling condi- tion refinements	174
4.5	Variability activities for human and auto enabling conditions	179
4.6	Mapping dependencies to control connectors	180
4.7	Mapping of alternatives to input and output data of variability activities	182
4.8	Variability model and generated customization flows	188
4.9	Component spheres and their influence on the generation of the provi- sioning customization flow	194
4.10	Solution engineering process and artifacts	197
5.1	Different granularity of provisioning services	205
5.2	Using a provisioning flow to overcome granularity differences	208
5.3	Different component flows for the same component type but different provisioning engines	212
5.4	Provisioning and management infrastructure	213
5.5	Basic component states and messages	215
5.6	Messages and exchanged customization documents during successful provisioning of a component	216
5.7	Additional states and operations for provider components	220
5.8	Additional states and operations for single configurable instance compo- nents related to subscriptions	222
5.9	States for single configurable instance provider components	224
5.10	Overview of the provisioning subprocesses and required artifacts	225

5.11	Condensed representation of an application model and maximal provisioning groups	229
5.12	Possible provisioning services for an application model	232
5.13	Possible (partial) realization deployments of an application model	236
5.14	Activities for the evaluation of multiple equivalence classes in the evaluation flow	245
5.15	Select realization component binding subprocess, component flow interaction	247
5.16	Ordering relevant dependencies and resulting provisioning order example	249
5.17	Two possibilities to derive provisioning flows	254
5.18	Example combined provisioning dependency graph, provisioning order graph and derived provisioning flow	256
5.19	Activities of the provisioning activities subprocess	257
5.20	Interaction between the pre-provisioning subprocess and the customization flow	258
5.21	Interactions between the provision new component subprocess and the component flow for provisioning	260
5.22	Interactions between the subscribe to component subprocess and the component flow for provisioning	260
5.23	Interactions between the runtime configuration subprocess, the customization flow and the component flow for runtime customization	263
6.1	Main components of the Cafe architecture	270
6.2	Application model editor	272
6.3	Variability model editor	273
6.4	Component binding editor	274
6.5	Template selection in the Cafe application portal	277
6.6	Customization flow user interface	278
6.7	BPEL provisioning flow	286
6.8	Scopes, fault and compensation handlers in the BPEL provisioning flow	288
7.1	eCCo screenshot	305
7.2	DecidR Web-based modeling tool	308
7.3	Omitted components in the headless Cafe usage in DecidR	310
7.4	The components of the sample application in the application model editor	312
7.5	Variability of the sample application in the variability model editor	313

7.6 Ingredients of the sample application Car archive's root folder 314

LIST OF TABLES

2.1	State of the art evaluation of packaging formats	57
3.1	Additional properties of a variability model	96
3.2	Additional properties of a variability point	103
3.3	Additional properties of an alternative	114
3.4	Allowed dependencies between variability points of different components	128

LIST OF ALGORITHMS

4.1	Navigating through a variability model	164
4.2	Process model generation algorithm	185
5.1	Navigation logic of an evaluation flow	244
5.2	High-level provisioning flow generation	255

LISTINGS

3.1	Application model pseudo schema	88
3.2	Application model example XML	89
3.3	Variability descriptor pseudo XML schema	97
3.4	Import pseudo schema	99
3.5	Variability point pseudo schema	103
3.6	Variability point refinement pseudo schema	104
3.7	XPath locator example XML	107
3.8	Alternatives example XML	113
3.9	Dependency example XML	115
3.10	Enabling condition example XML	119
3.11	Refinement enabling conditions and alternative example XML	122
3.12	Component binding descriptor pseudo schema	138

LIST OF MATHEMATICAL SYMBOLS

In this list, sets and mathematical symbols used throughout this thesis are listed. Definition or section numbers are given where the set or symbol is introduced in detail.

Table 8.1: Symbols and Main Functions

Notation	Meaning
AM	The set of application Models (Def. 1)
$ad : CT \rightarrow \rho(IT)$	Assigns a set of implementation types to a component type to indicate that components with one of these implementation types can be deployed on components of that component type (Def. 15)
$allowedVal : Alt^{Free} \rightarrow \rho(B)$	Assigns the set of building blocks allowed at a free alternative (Def. 36)
Alt	The set of all alternatives in a variability model (Def. 29)
$Alt_{ref}(vp)$	The set of alternatives associated to a variability point and its refinement (Def. 52)
$AltR : Alt \rightarrow Alt$	Assigns an alternative as a refinement to another alternative (Def. 51)
$AT = AM \times VM \times \rho(CB)$	The set of application templates (Def. 61)
$avm : CT \rightarrow VM$	Assigns an (abstract) variability model to a component type (Def. 20)

B	A set of building blocks (Def. 7)
$B(c)$	All building blocks in files of a component c (Def. 8)
B_{la}	The set of building blocks referenced by a locator alternative la (Def. 34)
$B^{Loc}(vp)$	The set of building blocks referenced by all locators of a variability point vp (Def. 38)
$bindT : VP \rightarrow Phases$	Assigns a phase to a variability point in which the variability point must be bound (Def. 23)
$BL = \rho(B) \times \{\text{replace, before, after}\}$	The set of building block locators (Def. 27)
$blockOf : B \rightarrow F$	Assigns building blocks to files (Def. 7)
$buildingBlocks : E \rightarrow \rho(B)$	Returns all building blocks referenced by an expression (Def. 33)
C	The set of components in an application model (Def. 2)
$\vec{C}(c)$	The set of components required to deploy a component c (Def. 12)
$C^+(c)$	The set of transitively required components to deploy a component c (Def. 13)
$C_{prov}^+(c)$	The set of components another component depends on during provisioning (Def. 55)
C_{atomic}	The set of atomic components in an application model (Def. 14)
$C_{composite}$	The set of composite components in an application model (Def. 15)
$C_{deploy}(am)$	Components in an application model that can be deployed (Def. 92)
$C_{prov}(am)$	Components in an application model that can be provisioned (Def. 90)
$C_{provisionable}(p)$	The set of components a provider can provision (Def. 59)
$C_{provisioned}(p)$	The set of components already provisioned at a provider (Def. 58)

$C_{reuse}(am)$	Components in an application model that can be reused (Def. 91)
$cb : C_a \rightarrow C_{provisioned}(p) \cup C_{provisionable}(p)$	A component binding assigns a provisioned or provisionable component to a component in an application model (Def. 60)
CB	The set of component bindings (Def. 60)
$CB_{complete}(am, cb, p)$	The set of complete component bindings possible for an application model at a provider (Def. 100)
$Cd \subseteq C \times C$	The set of component dependencies induced by variability (Def. 54)
Cd_{prov}	Component dependencies (variability) relevant for provisioning (Def. 55)
Co	The set of conditions declared in a variability model (Def. 46)
$cmCust(vm) : VMC \times \rho(VP) \rightarrow \{\text{true}, \text{false}\}$	Returns whether a customization is complete (Def. 81)
$cond : EC \rightarrow Co \cup \{\perp\}$	Assigns a possibly empty condition to an enabling condition (Def. 46)
$crCust(vm) : VMC \times \rho(VP) \rightarrow \{\text{true}, \text{false}\}$	Returns whether a customization is correct (Def. 82)
$crmCust(vm)t : VMC \times \rho(VP) \rightarrow \{\text{true}, \text{false}\}$	Returns whether a customization is correct and complete (Def. 83)
CT	The set of all component types (Def. 3)
$CT_{provider}$	The set of provider component types (Def. 17)
$D \subseteq C \times C$	The set of deployment relations among components (Def. 10)
$DG = (C, D)$	The directed, acyclic deployment graph where C are the nodes and D are the edges of the graph (Def. 11)
$Dp \subseteq VP \times VP$	The set of dependencies between variability points (Def. 40)
DP	The set of deployments at all providers (Def. 57)
$DP_{compatible}(g, p, am, Dep)$	The set of compatible deployments at a provider given a group of components g , an application model am and a set of deployments Dep (Def. 95)

$DP_{\text{partialReal}}(mg, am, p)$	The set of partial realization deployments for a maximal reuse group at a provider p (Def. 99)
E	The set of expressions used in a variability model (Def. 33)
$ec : VP \cup VPR \rightarrow \rho(EC)$	Assigns a set of enabling conditions to a variability point or variability point refinement (Def. 45)
$ec_i(vp)$	The i -th enabling condition at a variability point vp (Def. 50)
EC	A set of enabling conditions in a variability model (Def. 45)
$EC_{\text{auto}}(vp)$	The set of automatic enabling conditions at a variability point vp (Def. 48)
$EC_{\text{true}}(vp, s)$	The set of enabling conditions that evaluates to true at a step s (Def. 68)
$ecR : EC \rightarrow EC$	Assigns an enabling condition to another enabling condition, that the first enabling condition refines (Def. 53)
$ECRef_{\text{true}}(ec, s)$	The set of enabling condition refinements of enabling condition ec that evaluate to true at step s (Def. 69)
$emptyValue : Alt^{\text{Empty}} \rightarrow \perp$	Assigns the empty value to an empty alternative (Def. 32)
$enAlts : EC \rightarrow \rho(Alt)$	Assigns a set of enabled alternatives to an enabling condition (Def. 46)
$enAltS : VP \cup VPR \times S \rightarrow \rho(Alt)$	Returns the enabled alternatives at a variability point or refinement at a given step (Def. 72)
$EqCl(s)$	The set of all equivalence classes of component bindings at step s (Def. 101)
$evaluate : Co \times S \rightarrow \{true, false\}$	Boolean value of a condition at a step (Def. 67)
$explicitValue : Alt^{\text{Explicit}} \rightarrow B$	Assigns an explicit value to an explicit alternative (Def. 31)
$expression = Alt^{\text{Expression}} \rightarrow E$	Assigns an expression to an expression alternative (Def. 33)
F	The set of files in an application model (Def. 6)

$f_{true} : VP \times States \rightarrow EC$	Returns the first enabling condition that evaluates to true (Def. 70)
$f_{true}^{ref} : EC \times States \rightarrow EC$	Returns the first enabling condition refinement that evaluates to true (Def. 71)
$files : I \rightarrow \rho(F)$	Assigns a set of files to an implementation (Def. 6)
$freeValue : Alt^{Free} \rightarrow B \cup \{\perp\}$	Returns the value entered at a free alternative (Def. 35)
I	The set of implementations in an application model (Def. 4)
$impl : C \rightarrow I$	Assigns an implementation to a component (Def. 4)
$implT : I \rightarrow IT$	Assigns an implementation type to an implementation (Def. 5)
IT	The set of implementation Types (Def. 5)
$inputData : Co \rightarrow \rho(B \cup Alt)$	Input data for a condition (Def. 46)
L	The set of locators (Def. 28)
$locators : VP \cup VPR \rightarrow \rho(L)$	Assigns a set of locators to a variability point or refinement (Def. 28)
$MG(am)$	The set of maximal reuse groups in an application model (Def. 97)
$\Omega_{VP_{ec}}$	The set of strict total orders over the enabling conditions of a variability point (Def. 49)
$p : C \rightarrow P$	Assigns a multi-tenancy pattern to a component (Def. 9)
P	A set of multi-tenancy patterns (Def. 9)
$Pd = D \cup Cd_{prov}$	The set of dependencies (deployment and variability) between components at provisioning time (Def. 104)
$PG(am)$	The set of provisioning groups in an application model (Def. 93)
$PG_{max}(am)$	The set of maximal provisioning groups in an application model (Def. 94)
$Phases$	A set of phases in which variability points can be bound (Def. 22)
$POG = (C_{prov}, Pd^{-1})$	The provisioning order graph (Def. 105)

$possVal : Alt \rightarrow \rho(B)$	Returns the allowed values in form of building blocks at an alternative (Def. 37)
$Prov$	The set of Providers (Def. 56)
$provDep : PS \rightarrow \rho(DP)$	Returns the provisionable deployments of a provider (59)
$provDp : Prov \rightarrow \rho(DP)$	Returns the set of all deployments available at a provider (Def. 57)
$ps : Prov \rightarrow \rho(PS)$	Returns the provisioning services available at a provider (Def. 59)
PS	The set of provisioning services (Def. 59)
$PS_{possible}(pg, am, p)$	The set of possible provisioning services for a provisioning group an application model at a provider (Def. 96)
$rcb : SOL \rightarrow CB$	Assigns a component binding as a realization component binding to a solution (Def. 102)
$real : C \rightarrow C_{provisioned}(p) \cup C_{provisionable}(p) \cup \perp$	Returns the realization component for a component (Def. 102)
$ref : VM \rightarrow \rho(VM)$	Imports a set of variability models to be refined into a variability model (Def. 19)
RL	The set of roles that can perform customization (Def. 25)
$roles : VP \rightarrow \rho(RL)$	Assigns a set of roles that can bind a variability point (Def. 25)
S	Steps in a customization (Def. 62)
$selAlt : VP \rightarrow Alt \cup \{\perp\}$	Returns the selected alternative for a variability point (Def. 76)
$selVal : VP \rightarrow B \cup \{\perp\}$	Returns the selected value for a variability point (Def. 77)
SOL	The set of customer-specific application solutions (Def. 88)
$States$	States a variability point can have (Def. 63)
$type : C \rightarrow CT \cup \{\perp\}$	Assigns a component type to a component (Def. 3)
$VG = (VP, Dp)$	A directed acyclic graph of variability points and dependencies (Def. 41)
VM	The set of variability models (Def.18)

VP	A set of variability points in a variability model (Def. 21)
$VP(c)$	The set of variability points affecting a component c (Def. 39)
$VP(p)$	The set of variability points that must be bound in Phase p (Def. 24)
$VP^{\leftarrow}(vp)$	A set of variability points a variability point depends on (Def. 43)
$VP^{*\leftarrow}(vp)$	A set of variability points a variability point transitively depends on (Def. 44)
$VP^{Automatic}(s)$	The set of variability points that can be auto-bound at a given step s (Def. 78)
$VP^{Independent}$	A set of independent variability points in a variability model (Def. 42)
$VP_{potDisabled}(s)$	The set of potentially disabled variability points at step s (Def. 74)
$VP_{potActive}(s)$	The set of potentially active variability points at step s (Def. 75)
$VP_{st}(s)$	Variability points with a given state st at step s (Def. 65)
VMC	The set of customizations of a variability model (Def. 80)
$vmcCB : CB \rightarrow VMC$	The variability model customization of a component binding (Def. 101)
VPR	The set of variability point refinements in a variability model (Def. 26)
$VPrefine : VPR \rightarrow VP$	Assigns a variability point that a refinement refines (Def. 26)