Diploma Thesis

Feature-based Configuration Management of Applications in the Cloud

submitted by

Xi Luo

born 09.10.1986 in Beian, China

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Supervisor: Dr. Andreas Rummler, Dipl.-Medieninf. Julia Schröter
Professor: Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted 30.04.2013

Acknowledgements

- I express my most sincerest gratitude to Dr. Andreas Rummler and Dipl. - Medieninf. Julia Schröter, the supervisors of my thesis, for their constant supervision, sufficient consideration, great patience, complete support and utmost dedication. I deeply appreciate their valuable as well as constructive advice when commenting and discussing in the while process of thesis writing.

- I am also profoundly grateful to my parents and all the friends, who have been granting me their unconditional favor for the thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last decades, cloud computing became increasingly hot and created the most buzz. Cloud computing is the utilization of computing resources (hardware and software) that are delivered as a service over a network (typically the Internet). It entrusts remote services with a user's data, software and computation. End users can access cloud-based applications through a web browser or a mobile application, and their data are stored in servers at a remote location. Thus, cloud computing is an efficient way for IT to add capabilities or increase capacity on the fly without investing in new infrastructure or licensing new software.

The two most normally cited examples of cloud offering are Amazon and Google. Basically, both of them rent their inner data-center resources to outside customer [Wei09]. For example, in Amazon's Elastic Compute Cloud (EC2)[1], customers can rent virtual-machine instances and run their applications on Amazon's hardware. Like in the case of Amazon, in cloud computing, applications are usually installed in remote servers instead of customers' local computers. Therefore, the cloud computing resources (hardware and software) are shared by various unrelated users. To enable sharing of resources and costs across a large pool of users, multi-tenancy is regarded as one of the essential characteristics of cloud computing [ZCB10]. In the principle of multi-tenancy, a single instance of the software runs on a server and serves multiple client organizations (tenants). With a multi-tenant architecture, a software application can virtually partition its data and configuration. Thus, multiple tenants share the same application, running on the same operating system, on the same hardware, with the same data-storage mechanism, but they do not share or see each other's data.

[1]aws.amazon.com/ec2/

1.1 Motivation

In the provision of a cloud-based application, various stakeholders with different objectives are involved, i.e. providers of all cloud stack layers as well as tenants and their users [MG11]. Normally, according to their objectives, providers can offer different computing resources that are shared by various tenants. For example, infrastructure providers offer different kinds of infrastructure resources, i.e. servers and networks. Based on the supplied resources, platform providers offer platform resources, such as frameworks. Then application providers rent platform resources and provide their applications with configurable functionalities. Finally, tenants as well as their users can rent applications on demand and pay only for the services they use and for a certain usage time. In the above example, applications in the cloud environment require different configuration stages from different stakeholders. In addition, during the configuration process, new tenants can be added and removed dynamically. Therefore, a dynamic, yet scalable configuration management is required for providing highly configurable applications for multiple tenants and their associated users in a shared cloud environment [SMM+12].

The purpose of this thesis is to develop a configuration management that is able to manage and create tenant configurations for cloud-based applications. By modeling a configuration workflow, the configuration management enables a centered and structured configuration process. The configuration workflow intents to manage the variability of cloud-based applications. After staged configuration of related stakeholders, a complete configuration per user is created and all variability is bound. Additionally, the configuration workflow must allow for the dynamical stakeholders integrating during or after the configuration process. Based on Eclipse and extended feature models, the concepts are prototypically implemented in a tool. By using a concrete case study and experiments, the concepts and the applicability of the solution to the SAP Netweaver Cloud are evaluated.

1.2 The Structure of This Document

The structure of the thesis is listed as follows:

- Chapter 2 describes the background and relevant technologies which are utilized in our approach.

- Chapter 3 introduces an example of a cloud-based application as a case study. Based on the example and the literature [SMM+12] we identify the requirements for configuration management of cloud-based applications.

- Chapter 4 introduces our configuration management concept, including configuration management specification and configuration workflow adaptations. Furthermore, we also describe how to adapt our concept to the use case introduced in Chapter 3.
- Chapter 5 shows how the concept is implemented based on Model-driven engineering (MDE) and demonstrate the feasibility of our concept with the help of the use case.
- Chapter 6 concludes the thesis by outlining our approach and contributions. Furthermore suggestions for future work are also discussed.

Chapter 2

Background

As stated in the introduction, this thesis focuses on proposing concepts for configuration management of cloud-based applications. In this chapter we introduce the related background and used methods as well as models. In addition, we also show the selected literatures related to the research area covered in this thesis.

2.1 Cloud Computing

The cloud computing is an approach of providing computing resources (e.g. networks, servers, applications, storage and services) as a service under the definition from National Institute of Standards and Technology (NIST) [MG11]. Therefore, computing resources are dynamically configurable depending on the demand of a tenant and its users. According to the abstraction level of capability and service model of providers, cloud computing services are divided into three classes, namely: (1) Infrastructure as a Service, (2) Platform as a Service, and (3) Software as a Service. Figure 2.1 depicts the layered organization of the cloud stack from physical infrastructure to applications.

1. Infrastructure as a Service: IaaS provides virtualized resources (such as computation, storage, and communication) on demand [SMLF09]. The services from this layer are considered to be the bottom layer of cloud computing systems [NWG+09]. The cloud owner who offers IaaS is called an IaaS provider. Examples of IaaS providers include Amazon EC2[1], GoGrid[2] and Flexiscale[3].

[1]Amazon Elastic Computing Cloud, aws.amazon.com/ec2

[2]Cloud Hosting, Cloud Computing and Hybrid Infrastructure from GoGrid, http://www.gogrid.com

[3]FlexiScale Cloud Comp and Hosting, www.flexiscale.com

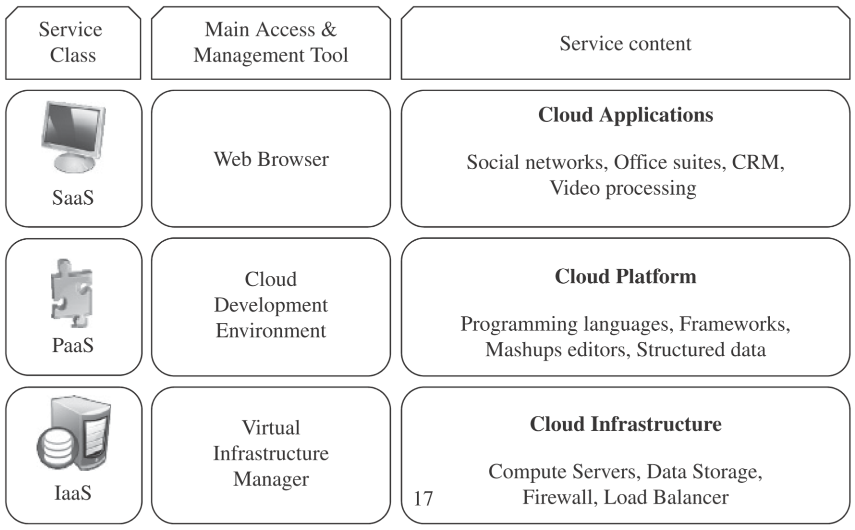

Service Class
Main Access & Management Tool
Service content
SaaS
Web Browser
Cloud Applications
Social networks, Office suites, CRM, Video processing
PaaS
Cloud Development Environment
Cloud Platform
Programming languages, Frameworks, Mashups editors, Structured data
IaaS
Virtual Infrastructure Manager
Cloud Infrastructure
Compute Servers, Data Storage, Firewall, Load Balancer
17

Figure 2.1: The cloud computing stack [VBB11]

2. Platform as a Service: PaaS refers to offering an environment (operating system support and software development frameworks) on which developers create and deploy applications. It makes a cloud easily programmable so that developers do not necessarily need to know how many virtualized resources that applications will be using. Examples of PaaS providers include Google App Engine[4], Microsoft Windows Azure[5] and Force.com[6].

3. Software as a Service: SaaS refers to offering applications over the Internet. In this business model applications are rented to customers (tenants) on demand. Therefore, tenants pay only for the services they use and for a certain usage time. Examples of SaaS providers include Salesforce.com[7], Rackspace[8] and SAP Business ByDesign[9].

In this thesis we focus on configurable cloud applications allowing for tailored functionality. In order to save costs, some cloud applications are based on a multi-tenant architecture. A multi-tenant application allows customers to share the same hardware resources by providing them one shared application and database instance. The application is configured to fit tenants' needs as

[4] Google App Engine, URL http://code.google.com/appengine
[5]Windows Azure, www.microsoft.com/azure
[6] Salesforce CRM, http://www.salesforce.com/platform
[7]Salesforce CRM, http://www.salesforce.com/platform
[8]Dedicated Server, Managed Hosting, Web Hosting by Rackspace Hosting, http://www.rackspace.com
[9]SAP Business ByDesign, www.sap.com/sme/solutions/businessmanagement/businessbydesign/index.epx

if it runs on a dedicated environment [BZP+10]. Usually, a tenant groups a number of users, which are the stakeholders in an organization renting a multi-tenant SaaS solution. With this model, application deployment becomes easier for service providers, as only one application instance has to be deployed.

In order to configure cloud-based multi-tenant aware applications, methods from software product line engineering (SPLE) can be used to deal with the commonality and variability. In next section, we introduce SPLE and its utilization.

2.2 Software Product Line Engineering

Software product lines (SPL) refers to software engineering methods, tools and techniques for producing a set of similar software products that share more commonalities than variability [BSRC10]. Generally, the customers' requirements are the same and no customization is performed. In order to achieve customer's personalization, software product line engineering (SPLE) promotes the production of families of software products from common features rather than production of individual products.

Division of Software Product Line Engineering

The main difference between SPLE and normal software development is that SPLE has a logical separation within the development of core, i.e. *domain engineering* and *application engineering* [PBVDL05].

- **Domain Engineering:** This process is responsible for establishing the reusable platform and defining commonality as well as the variability of the product line. The platform is comprised of software artefacts (requirements, design, realization, test etc.).

- **Application Engineering:** This process is responsible for deriving product line applications by reusing domain artefacts and exploiting the product line variability.

The above split allows for separating two concerns to build a robust platform and to build customer-specific applications. The interaction of the two processes must be beneficial with each other. For example, the design of the platform must be of use for application development, and application development must assist in using the platform.

The terms *Problem Space* and *Solution Space* together represent the development phases of SPLE, as in [KU00] introduced by Czarnecki and Eisenecker.

- **Problem Space:** The problem space generally refers to systems' specifications during the domain analysis and requirements engineering phases. It is used to describe the desired combination of problem variability to implement a product variant.
- **Solution Space :** The solution space refers to the concrete systems created during the architecture, design and implementation phases. It describes the composing assets of a product line and its relation to the problem space, i.e. rules for how elements of the platform are selected when certain values in the problem space are included in a product variant.

The four-part division resulting from the combination of the problem space and solution space with domain and application engineering is shown in Table 2.1.

	Problem Space	Solution Space
Domain Engineering	Variability within the problem area	Structure and selection rules for the solution elements of the product line platform
Application Engineering	Specification of the product variant	The needed platform elements (and additional application elements if required)

Table 2.1: Overview of SPLE activities [BD07]

Feature Model

In SPLE, *Feature Models* are widely used for variability and commonality management [KCH$^+$90]. A feature model describes the features of a set of software products in the domain as well as relationships between them. A feature is a system attribute relevant for some stakeholders and is used to describe commonality and variability. According to the stakeholders' interest a feature can be a requirement, a non-functional characteristic or a technical function. Features are organized in *feature diagrams*. A feature diagram is a visual notation of a feature model, which is an *and/or* tree of different features. The root of a feature diagram represents a concept (e.g. a software product), and its nodes are features.

Figure 2.2 depicts a simplified feature model of a mobile phone that illustrates how features are used to specify and build software for mobile phones. Based on the feature model we show the following relationships and constraints among features of a basic feature model [BSRC10]:

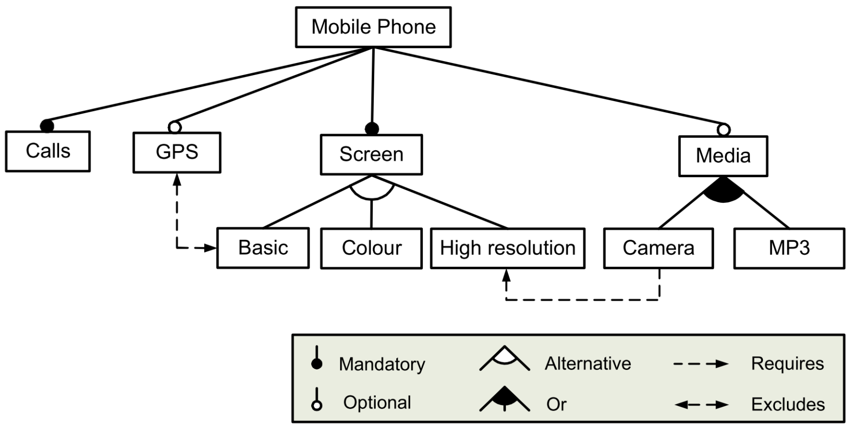


Figure 2.2: A sample feature model of a mobile phone [BSRC10]

1. Relationships between a parent feature and its child features (subfeatures)

    - **Mandatory:** A *mandatory* relationship between a child feature and its parent feature depicts that the child feature must be included in all products in which its parent feature appears. For example, every *mobile phone* must include support for *calls*.
    - **Optional:** An *optional* relationship between a child feature and its parent feature depicts that the child feature can be optionally included in all products in which its parent feature appears. In the example, a *mobile phone* may optionally provide support for *GPS*.
    - **Alternative:** An *alternative* relationship between a set of child features and their parent feature depicts that only one child feature can be included in a product in which the parent feature appears. For instance, a *mobile phone* may support for only one feature among *basic*, *color* and *high resolution* for a *screen*.
    - **Or:** An *or* relationship between a set of child features and their parent feature depicts that one or more child features can be selected when the parent feature is part of the product. In the example, *camera*, *mp3* or both of them can be selected when *media* is selected.

2. Cross-tree constraints

    - **Requires:** The inclusion of a feature A implies the inclusion of a feature B in a product if A *requires* B. For example, a *mobile phone* with *camera* must include support for a *high resolution* screen.
    - **Excludes:** A feature A and a feature B cannot be included in the same product if A *excludes* B. For example, *GPS* and *basic* screen

cannot be included in a *mobile phone*.

Based on the basic feature model, sometimes it is necessary to extend feature models to include more features' information, so-called *feature attributes*. These types of models with additional information are called *Extended Feature Models* (EFM) [BSRC10]. In the seminal report on feature models of FODA [KCH$^+$90] the inclusion of additional information in feature models is already contemplated. For example, relationships between feature and feature attributes were presented. Furthermore, the inclusion of attributes in feature models is also proposed in [CK05], [BRCT05], [BTRC05] and [BBRC06]. Many proposals agree that an attribute should contain a *name*, a *domain* and a *value*. Therefore feature attributes can be used to describe functional or non-functional information to support the feature.

Besides, some authors also propose extending feature models with *cardinalities* [CHE05a, RBSP02]. In our concept we utilize the *group cardinality*. A group cardinality is an interval denoted $<n..m>$ with n as lower bound and m as upper bound. The interval limits the number of child features that may be included in a product if their parent feature is selected.

As stated in the literature [MMLP09], [RA11] and [SCG$^+$12], methods from SPLE are convenient to handle the commonality and variability of SaaS applications. In this thesis, we focus on managing the variability of cloud-based applications. With the help of EFM we set out the variability and define the configuration space of cloud-based applications. By utilizing EFM, a multi-tenant SaaS application can be configured by involved stakeholders according to their requirements.

2.3 Role Based Access Control

In the provision of a cloud-based application, various stakeholders are involved. According to their roles, the stakeholders have different permissions to select application characteristics. Therefore, a view concept is required to determine stakeholders' possible configuration choices based on their roles.

Role-Based Access Control (RBAC) is an approach to restricting system access to authorized users [FKC92]. Usually, access control decisions are determined by the roles on which individual users take as part of an organization. This may include responsibilities, qualifications and duties. For example, roles in a hospital include doctor, nurse, clinician and pharmacist. The concept of RBAC began with multi-user and multi-application on-line systems pioneered in the 1970s. The central notion of RBAC is to associate permissions with roles, and to assign users to appropriate roles. In an organization to different job functions, different roles will be defined. According to their respective responsibilities users are assigned to corresponding roles. With RBAC the management of permissions is extremely simplified.

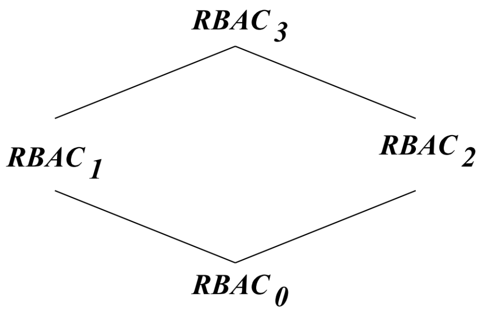


Figure 2.3: Relationships among RBAC models [SCFY96]

In [SCFY96], according to various dimensions of RBAC a family of four conceptual models are defined. Figure 2.3 shows the relationship between the four models. $RBAC_0$ is the base model that indicates the minimum requirements for any system supporting RBAC. $RBAC_1$ and $RBAC_2$ are called advanced models, which include $RBAC_0$ and additional independent features. $RBAC_1$ adds the concept of role hierarchies that are depicted in later paragraph. $RBAC_2$ adds constraints. As the consolidated model, $RBAC_3$ includes $RBAC_1$ and $RBAC_2$ and, by transitivity, $RBAC_0$. Considering the current common cloud-based applications, various stakeholders may have role hierarchies. For instance, various tenants inherit same permissions from their parent role, but hold different roles for the purpose of identification (e.g. two tenants from different organizations). Therefore, in this thesis we are only interested in the $RBAC_1$ with the concept of role hierarchies. In the following part we describe the base model $RBAC_0$ and the concept of role hierarchies.

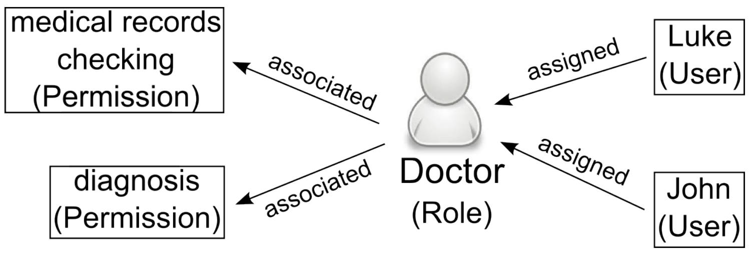


Figure 2.4: Role relationships

The base model of RBAC consists of *Role*, *User* and *Permission.* A *role* is a job function and can be thought of as a collection of permissions that a user or a set of users perform within the context of an organization. A *user* is normally a human being. A *permission* is an approval of a particular model of access to one or more objects in a system and allocated to roles in an organization. The relationships among individual roles, users and permissions are depicted in Figure 2.4. In the figure, the role *Doctor* is assigned to users *Luke* and *John*, so that both of them have the permissions of *diagnosis* and *medical records checking* which are associated with the role *Doctor*.

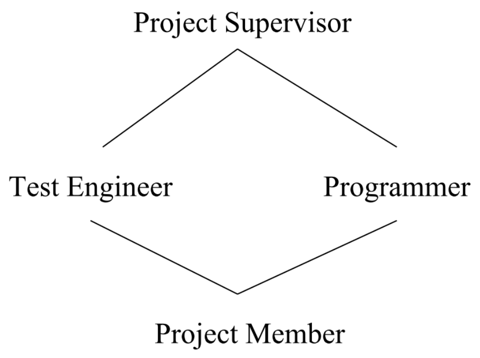


Figure 2.5: Example of role hierarchies [SCFY96]

In the literature [FKC92], [HDT95] and [NO94] a concept of role hierarchies is discussed. Usually, the concept is implemented in systems that provide roles. Role hierarchies refer to the situations where roles can inherit permissions from other roles. An example of role hierarchies is shown in Figure 2.5. In the example, more powerful roles are shown towards the top of the diagram while less powerful roles towards the bottom. The programmer role and test engineer role inherit all permissions from project member role. In addition to the inherited permissions they can have individual permissions. Besides, role hierarchies also support multiple inheritance of permissions. For example, project supervisor role inherits from both test engineer and programmer roles.

In this thesis we use the $RBAC_1$ model as a basis to define the specialization steps performed by stakeholders on EFM. The details is introduced in Chapter 4.

2.4 Staged Configuration

As mentioned in Section 2.2, a feature model can be used to depict the configuration space of a software product family. By selecting the desired features from a feature model an application engineer can specify a product in a product family. During the configuration process, the configuration operations are performed in a certain order. According to Czarnecki et al. [CHE04] the above process is called *staged configuration*, if the process may also be performed in stages, where each stage can eliminate some configuration choices. In a staged configuration each stage takes a feature model as an input and yields a specialized feature model as an output. The set of software products described by the output feature model is a subset of the products described by the input feature model.

In a *staged configuration* there are two types of processes: *configuration process* and *specialization process*.

- **Configuration Process:** The process of deriving a configuration is referred to as *configuration process*, while a *configuration* comprises the features that are selected according to the constraints in a feature diagram.

- **Specialization Process:** A *specialization process* is a transformation process that takes a feature diagram and yields another feature diagram. The collection of configurations denoted by the latter diagram is a true subset of the configurations denoted by the former diagram. The latter diagram is also referred to as a *specialization* of the former one. In addition, a fully specialized feature diagram expresses only one configuration.

Generally there are two extremes to perform a configuration process: 1) a configuration is derived from a feature diagram directly and 2) a configuration is derived by specializing a feature diagram top down to a fully specialized feature diagram. Therefore, a staged configuration can be achieved by successive specialization processes.

In a staged configuration, at each stage some *specialization steps* are applied and the last stage is followed by deriving a configuration from the most specialized feature diagram in the specialization process sequence. A *specialization step* refers to the removal of a certain configuration choice. There are six categories of specialization steps: a) refining a feature cardinality, b) refining a group cardinality, c) removing a grouped feature from a feature group, d) assigning a value to an attribute which only has been given a type, e) cloning a solitary sub-feature, and f) unfolding a feature reference. More details about each specialization step can be found in [CHE04].

A configuration stage can be defined in terms of three dimensions [CHE05a]:

- **Time:** A configuration stage may be defined in terms of different phases of a product lifecycle, such as product design, deployment, testing, etc.

- **Roles:** A configuration stage may be defined according to different roles with different responsibilities. In a staged configuration, each role is responsible to eliminate different variability.

- **Targets:** A configuration stage may be defined by a target system for which a given software needs to be configured.

In our concept we design a configuration workflow model so that a multi-tenant SaaS application can be configured by applying a staged configuration process. In the workflow model, each stage is defined in terms of *roles* dimension and thereby is performed by stakeholders. In addition, some stakeholders' configuration choices (e.g. tenants) depend on the pre-configuration process of other stakeholders (e.g. providers).

2.5 Workflow Modeling

The goal of this thesis is to model a configuration workflow that enables centered and structured configuration process for cloud-based applications. A workflow is a description of a sequence of a series of tasks (activities) through which work is routed [OAWtH10]. It is also used as a synonym for *a business process* and may be seen as any abstraction of real work.

2.5.1 Concept

The concept of a workflow is closely related to other concepts used to describe organizational structure, such as functions, teams and projects. A *workflow model* (also denoted as *workflow schema*) is used to specify which tasks need to be executed and in what order. Basically, a workflow model comprises a set of *nodes* that represent *start/end nodes*, *tasks* or *control connectors*, and a set of *control edges* between them [RW12].

- **Start/End Nodes:** The start node and end node separately indicates where a particular workflow will start or end.

- **Tasks:** A task (or action) is usually associated with an invokable application service and can either be atomic or complex. An atomic task represents an automated task or a manual task. While an automated task is automatically performed without human interaction, a manual task is made available as work items to users. A complex task refers to a subprocess and allows the modularization of a business process.

- **Control Connectors:** A control connector is used to describe *split* or *join* in the control flow of a workflow. For example, an *XOR-split* allows selecting one out of several outgoing branches, whereas an *OR-split* allows selecting at least one out of several outgoing branches.

- **Control Edges:** A control edge is used to depict the precedence relationship between nodes of a workflow.

To a workflow specification, a distinction is made between the *type level* and *instance level* [RW12]. While type level defines the schemes for executable workflow model at build time, instance level refers to the execution of related workflow instance at run time. Once a workflow executes, new workflow instances can be created and executed. Figure 2.6 depicts the life cycle of a workflow instance.

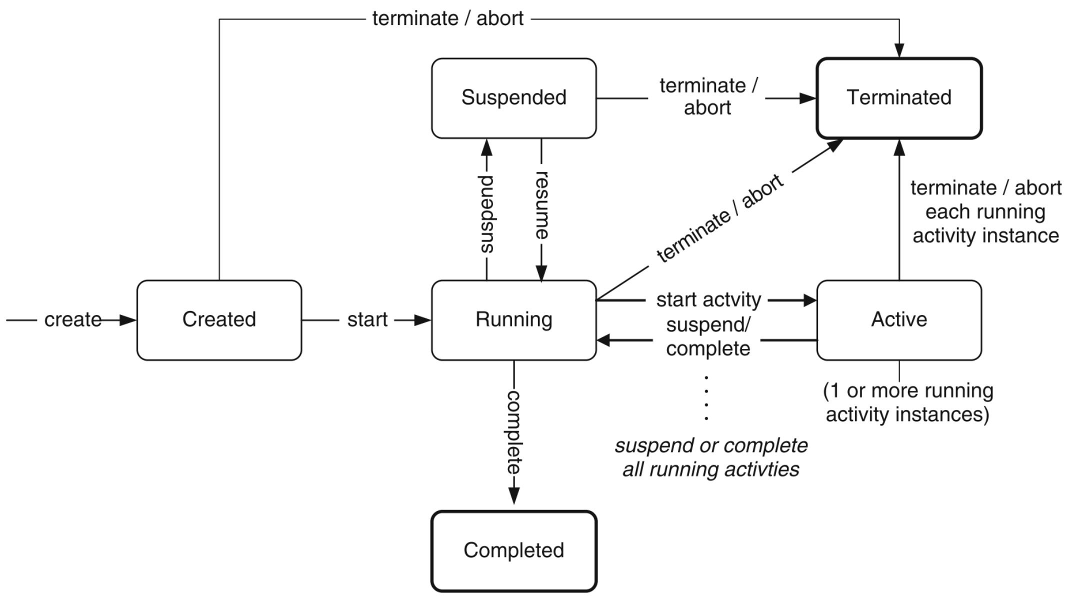


Figure 2.6: State transitions of a workflow instance [RW12]

In Figure 2.6, a newly created workflow instance has state *created*. When the instance starts to be executed, its state changes to *running*. When at least one of the enabled tasks is running, the instance enters state *active*. In addition, a workflow instance has the state *suspended* if it has no running tasks. Furthermore, if a workflow instance is abnormally terminated or aborted, different actions are required, depending on the concrete state of the instance. In the end, a workflow instance is *completed* if the end node is achieved.

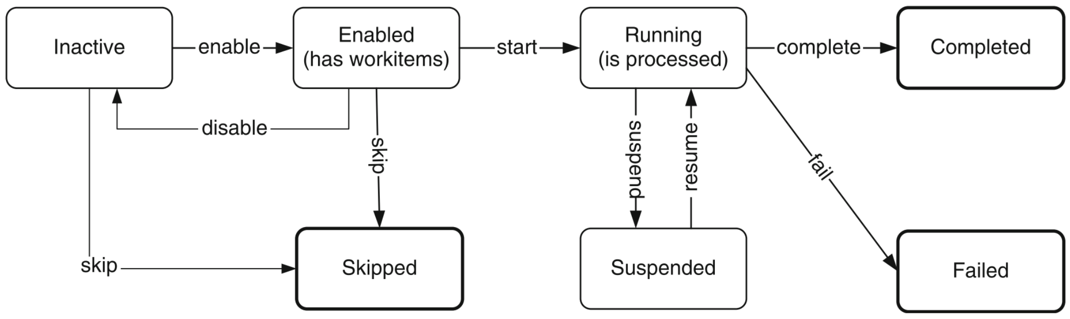


Figure 2.7: State transitions of a task instance [RW12]

During the execution of a workflow instance, a task instance represents an invocation of a task. A task instance utilizes data associated with its corresponding workflow instance and produces data utilized by succeeding tasks. Figure 2.7 shows the life cycle of a task instance [RW12]. When the preconditions for executing a task are met, the state of the task instance changes from *inactive* to *enabled*. If the task instance starts to run, its state changes to *running*. When a task instance completes, its state changes to *completed*. Besides, in order to cover more advance scenarios, three additional states (*skipped*, *suspended* and *failed*) are involved. First, a task instance in state *inactive* or *enabled* may be skipped (i.e. it enters state *skipped*) if an alternative path is selected for execution. Secondly, a task instance with state of *running* may be suspended (i.e. it enters state *suspended*) and resumed later (i.e. it reenters state *running*). Finally, a task instance switches to state *failed* if it fails because of errors.

2.5.2 Workflow Modeling Languages

Workflow languages are used to design workflow models and enable their execution [Wes12]. The commonly used workflow languages include: dedicated workflow specification languages (e.g. YAWL[10], XPDL[11]); executable process definition languages based on web services (e.g. BPEL[12]); workflow products (e.g. Websphere[13]). In order to specify workflows it is also possible to use languages for business process modeling, such as BPMN[14] and UML Activity Diagrams.

[10]http://www.yawlfoundation.org/
[11]http://www.xpdl.org/
[12]http://bpel.xml.org/about-bpel
[13]http://www-01.ibm.com/software/websphere/
[14]http://www.bpmn.org/

2.5.3 Adaptive Workflow

A workflow should provide *flexibility* because of its various changes. Thus, *adaptive workflow* is involved in order to deal with the dynamic modification of a workflow model. Adaptive (or dynamic) workflow refers to the extending of a static workflow in such a way that the workflow model can be modified or expanded in some way when changes occur [VDAVH04]. The changes of a workflow include, for example, users add tasks, delete tasks, postpone tasks' execution, etc. Usually, such behavioral changes of a workflow instance require *structural adaptations* of its corresponding workflow model [RW12]. Structural adaptations include, for example, insertion, deletion, movement of a task, etc.

The structural adaptations of a workflow model can be classified into two levels according to their scope or impact: *structural changes* and *ad-hoc changes* [VDAVH04].

- **Ad-hoc Changes:** In ad-hoc changes a single instance of a workflow is affected during the run time to cope with unanticipated exceptions.
- **Structural Changes:** In structural changes a workflow model is modified so that all new instances of the workflow benefit from the changes. A structural change is typically associated with a business process redesign (BPR).

Both levels of structural adaptations can be conducted by using *adaptation patterns*, which are described in Section 2.5.4.

2.5.4 Adaptation Patterns

Adaptation patterns allow users to structurally modify workflow models. Generally, a workflow model can be transformed into another workflow model by applying an adaptation pattern. Thus, two different approaches can be used to fulfill the structural adaptation [WRRM08]. One approach is based on *change primitives* that operate on single elements of a workflow model at a low abstract level, such as *add node*, *remove node*, *add edge*, or *remove edge*. Another approach is based on *high level change operations* that combine a set of change primitives to enable adaptations at a high abstract level, such as *add task*, *delete task* or *move task*.

Adaptation patterns constitute abstractions of high level change operations. An adaptation pattern contains one high level operation. Table 2.2 shows the catalog of adaptation patterns as identified in [WRRM08]. In the table, the term *process fragments* refers to specific parts of a workflow model.

Pattern category	**Pattern**
Adding or deleting process fragments	AP1: Insert process fragment AP2: Delete process fragment
Moving or replacing process fragments	AP3: Move process fragment AP4: Replace process fragment AP5: Swap process fragment AP14: Copy process fragment
Adding or removing process levels	AP6: Extract subprocess AP7: Inline sub-process
Adapting control dependencies	AP8: Embed process fragment in loop AP9: Parallelize process fragments AP10: Embed process fragment in conditional branch AP11: Add control dependency AP12: Remove control dependency
Change transition conditions	AP13: Update condition

Table 2.2: Catalog of adaptation patterns (AP) [WRRM08]

In our proposed concept we model the configuration workflow for cloud-based applications. We specify the workflow using UML Activity Diagrams as workflow modeling language. According to state transitions of task instances the proposed configuration workflow is able to be executed. In addition, a workflow is a directed graph and thus some characteristics as well as methods about graph are also suitable for workflow. With the help of *workflow adaptation patterns* and *graph transformation* (see Section 2.6) we realize the workflow's dynamic changes for supporting stakeholders' integration at run time.

2.6 Graph Transformation

Generally, a variety of problems that are typical to software engineering, can be represented as graphs or diagrams. To most activities in the software process, a lot of visual notations have been proposed, such as stage diagrams, control flow graphs, process modeling notations, and the UML family of languages. These notations produce the models that can be seen as graphs. Therefore, *graph transformation* is involved when specifying how the models are built and interpreted.

Graph Transformation or *Graph Rewriting* refers to a technique of creating a new graph from an original graph algorithmically [AEH$^+$99]. The basic idea of *graph transformation* is applying a rule to a graph and iterating this process. A graph consists of a set of vertices V and a set of edges E. Each edge e in E

has a source vertex $s(e)$ and a target vertex $t(e)$ in V. A rule r has the form of $L \to R$, in which L is called *left-hand side* of the rule and R is called *right-hand side* of the rule. In addition, a rule may also contain *application conditions* that ensure the performance of a graph transformation in a controlled way. Each rule application can transform a graph by replacing an occurrence of the left-hand side in the graph with a copy of the right-hand side. Thus, a graph transformation from an original graph G to a resulting graph H can be denoted by $G \Rightarrow H$. A collection of graph transformations is called *graph transformation system*.

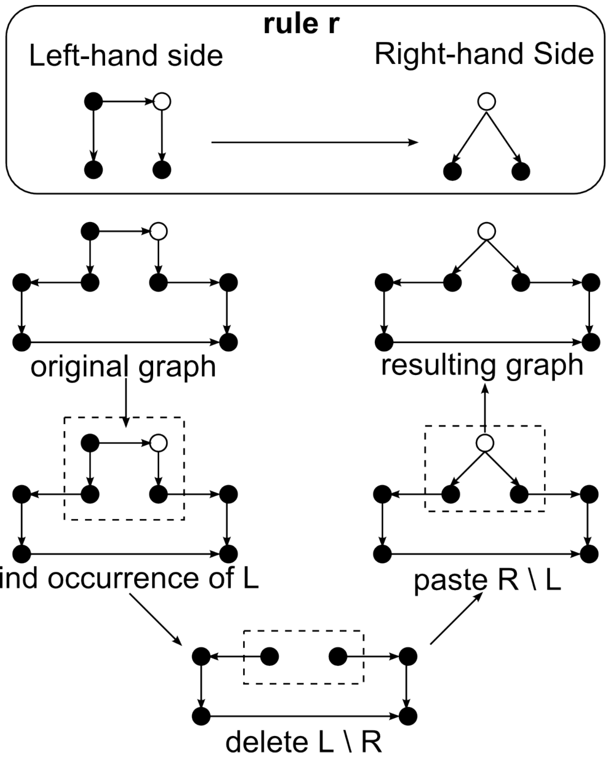


Figure 2.8: Illustration of a graph transformation step [AEH+99]

As depicted in [Hec06], a graph transformation can be performed in three steps. Figure 2.8 illustrates the steps which have to be performed when applying a rule r from the graph G to the graph H.

1. Find an occurrence of the left-hand side L in the given graph G.
2. Delete from G all vertices and edges matched by $L \backslash R$ (i.e. all vertices and edges that are in L but not in R).
3. Paste a copy of $R \backslash L$ (i.e. all vertices and edges that are in R but not in L) to yield the derived graph H.

Several properties of graph transformation need to be considered, if graph transformation is regarded as a specification and programming method [AEH+99]. Generally, the realization of these properties is not requested but may be helpful in certain contexts. Here we display two of them being relevant for our concept.

- **Confluence:** A graph transformation system is *confluent* if for each two graph transformations $G \Rightarrow G_1$ and $G \Rightarrow G_2$ there is a graph H such that $G_1 \Rightarrow H$ and $G_2 \Rightarrow H$ are valid. The property confluence implies that each graph can be transformed into at most one irreducible graph.
- **Termination:** A graph transformation system is called *terminating* if there is no infinite derivations $G \Rightarrow G_1 \Rightarrow G_2 \Rightarrow G_3 \Rightarrow ....$

Our proposed configuration workflow model utilizes the concept of graph transformation with consideration of above two properties. Graph transformation enables our configuration workflow model to integrate stakeholders dynamically at run time. Based on *workflow adaptation patterns* (see Section 2.5) we design *transformation rules* of graph transformation that is depicted in Chapter 4 in detail.

2.7 Related Work

In this section, we introduce work regarding configuration management, which tackles different research fields.

Rühl and Andelfinger propose utilizing SPL techniques to create highly customizable and configurable SaaS applications [RA11]. In order to achieve this goal a vision of an architectural model (a catalog) is presented. During domain engineering, the catalog is created to depict the flexibility of applications. Thus, the created catalog can be used to configure applications per tenant.

Further concept of applying SPL techniques on configurable SaaS application is given by Mietzner et al. [MMLP09]. In order to support SaaS providers in managing the variability of SaaS applications, the authors use variability modeling techniques from software product line engineering. Specially, they use explicit variability models to derive customization and deployment information for individual SaaS tenants. The authors argue that the already deployed application services can influence a tenant's configuration decisions. Therefore, the configurable variability of new tenants is restricted.

Hubaux et al. propose a view-oriented configuration processes based on feature models [HHS+11]. They introduce scheduled configuration workflows with concern-specific configuration views per stakeholder. A view is defined on a feature model to present features that are relevant for a certain stakeholder. Based on these views the configuration process is driven by a workflow.

Mendonca et al. propose a process-centric approach to collaborative product configuration [MCO07]. This approach enables decision makers to define priority schemes, and thus can anticipate and solve decision conflicts. In order to derive executable process models a novel algorithm is designed.

In addition, an approach to design and configure multi software product lines (MPLs) is proposed by Rosenmüller et al. [RS10]. They use composition models to describe how an MPL is composed from multiple SPL instances. The proposed models allow users to automate the configuration process of MPLs. Furthermore, configuration generators can be automatically derived to simplify the configuration process.

Chapter 3

Analysis

In this chapter, we present an example of a cloud-based application, in which various products are derived after multiple stakeholders' configurations. Through analyzing the example and related research presented in Chapter 2, we identify the requirements for the configuration workflow of a cloud-based application. These requirements are the basis for our main contributions.

3.1 Illustrative Example

In this section, we present a sample configuration scenario based on a well-known example from the industry automation area - a Yard Management System (YMS). As a case study for our approach, we consider the yard management system, which is used as a case study in the INDENICA project[1]. The main purpose of INDENICA is to abstract from service heterogeneity in a service-oriented environment. It provides methods, architectures, components, tools and assets for reuse-based creation of the adapted platforms. By utilizing SPL techniques, INDENICA creates a multi-tenant aware virtual service platform (VSP) that can abstract, integrate and enhance external services. Therefore, an application developed on top of this platform is independent of the underlying services and can make use of enriched features provided by the virtual platform itself, e.g. multi-tenancy [Con11b]. As a case study of INDENICA, the YMS is integrated into the VSP, which is multi-tenant aware. The VSP provides tenants various services which derive from the integrated platforms. Tenants may be SaaS providers with developers as potential users. They can manage their business through VSP by using one combined application.

We use the YMS to demonstrate the feasibility of our approach. On the one hand, yard management is a general and famous commercial scenario from

[1]http://www.indenica.eu/

the industry automation area. On the other hand, a prototype of the YMS is available and is able to run in a cloud-based environment (e.g. SAP NetWeaver Cloud[2]) as a SaaS application. The YMS itself is not multi-tenant aware, i.e. each instance has only one end user. We focus on the configuration process and identify the requirements for configuration management of cloud-based applications. During the configuration process, multiple stakeholder can perform the specialization steps and finally a configuration can be derived. The newly derived configuration will replace current configuration of the running application. In the following part, we introduce the yard management domain and the existing platform. Moreover, we also describe the provision process of the yard management system as a SaaS application in a cloud-based environment.

3.1.1 Domain and Exiting Platform

The Yard Management Domain

Large distribution centers have to handle ten thousands truckloads each year with up to 900 trucks a day. 800 employees have to collaborate to provide fast and reliable shipment completion of those truckloads. In order to assure the continuous flow of goods in the yard, people have to work hand in hand. The incoming trucks have to be registered at the gate guard and assigned to free docks for unloading the trailer. Some goods must be unloaded at particular docks, while others goods can be unloaded at the docks with a shorter storage distance. Every loading process has to be handled by warehouse staff, so the incoming trucks need to be notified on new delivery tasks. Furthermore, the yard jockeys are responsible for fetching trailers from the parking lot and taking them to a specific dock. All the processes are administered and monitored by the yard manager.

In order to solve the above problem, Yard Management Systems can be used for regulating trucks and trailer movements in the yard. Yard Management Systems are software systems designed to manage the movement of trucks and trailers in the yard of a manufacturing facility, warehouse, or distribution center [HTHS07]. They are often used in conjunction with Warehouse Management Systems (WMS) as well as Transportation Management Systems (TMS). This allows for exchanging information, such as advanced shipping notice and optimized material flow in the warehouse. Current Yard Management Systems support several features. Dock Door Scheduling allows for automatic scheduling of inbound deliveries by assigning arriving trucks to free docks based on business rules. Moreover, real-time information on the location of trailers is provided. Based on the real-time information, yard employees can move trailers

[2]https://www.sapcloudappspartnercenter.com/

from staging to docks to fill orders in an efficient manner. Some Yard Management Systems support identification of trailers via RFID or similar techniques.

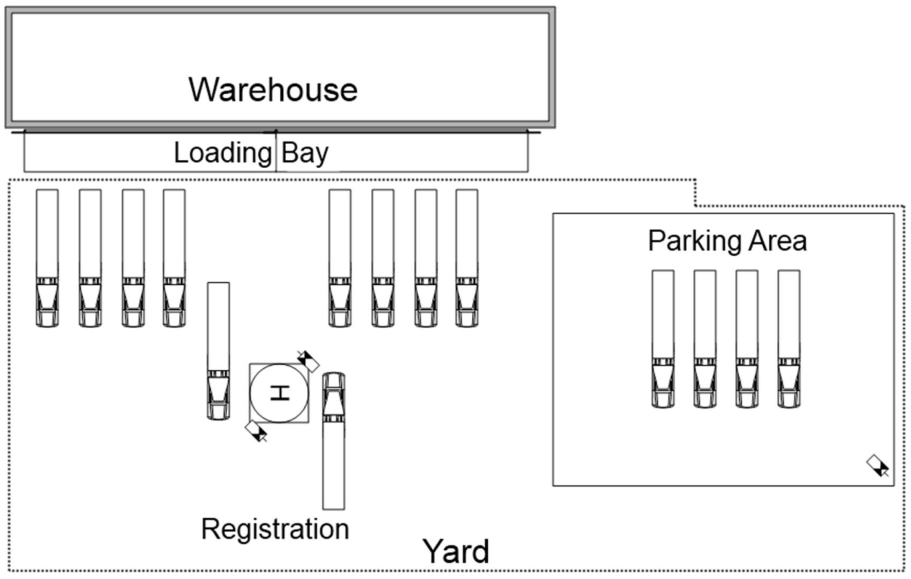


Figure 3.1: Schematic overview of a large warehouse [Con11a]

Figure 3.1 displays a schematic overview concentrating on the important parts of our scenario [Con11a]. It shows a warehouse with a loading bay and a yard including a parking area as well as a gateway. In the gateway, the trucks can register when they enter the yard and deregister when they leave the yard to be scheduled and coordinated during the load and unload operations. The scheduling can be planned in advance with the help of shipping notifications of transport providers. The goods are loaded or unloaded by the trucks in the loading bay. For large warehouse a parking place is required to be able to handle a large amount of trucks. Especially, during peak hours a large number of trucks arrive and leave within a short period of time. In addition, cameras are used to monitor the trucks and the activities on the yard. Therefore, the yard personnel are coordinated to run all processes efficiently.

Sample Application Use Cases

This subsection depicts the sample application use cases implemented for the YMS in INDENICA. The YMS supports two main features of yard management: Dock Door Scheduling (DDS) and Yard Jockey Support. In the following part, we describe the two scenarios in detail.

- **Dock Door Scheduling** The supplier or transport service provider sends a shipping notice to the warehouse. With the information about the arrival time and loading content, the warehouse manager plans further actions for a loading or unloading process and informs yard manager to prearrange the occupation of the dock doors. Truck drivers are able to update the warehouse manager about a more concrete arrival time so that the warehouse manager can plan more concretely and reschedule assignments and tasks. When the truck arrives at the yard, the driver checks in via a terminal and requests information about the assignment. The yard manager assigns either a dock door or parking area if no appropriate dock door is available.

- **Yard Jockey Support** The yard jockeys are able to get informed about new tasks. Such tasks can be: 1) Contact a truck driver after a service request, 2) Inform the truck driver who waits about loading start, and 3) Pick up a trailer for loading. The assignment of yard jockeys to tasks is based on their location and further schedule.

The YMS Services

The YMS consists of a base platform as well as several domain services. The base platform provides common services that are necessary for the development of most of the features of the yard management platform variant. Typical services which are provided by the base platform are persistence, messaging, authentication and web development support in general. The base platform is combined with several domain services to produce a domain-specific platform variant. These domain services include general support for yard management as well as additional services for data interchange, mobile communication and collaboration, which are depicted as follows.

- **Yard Management Service (YM):** This service provides basic logic to handle common yard management processes. It registers new shipping tasks, schedules and assigns arriving trucks free docks or waiting area in case of a dock unavailable, so-called Dock Door Scheduling (DDS). The service is used by the yard manager for administration and monitoring. The gate guard can also use the service to register new trucks and communicate with scheduled docks.

- **Yard Jockey Service (YJ):** This service allows for scheduling of yard jockey tasks, including fetching or relocating trailers on the yard. It maintains trailer location that allows intelligent tasks scheduling and optimizes the path of yard jockeys.

- **Mobile Communication Service (MC):** This service provides functionality for communicating with mobile devices. It is used to distribute notifications and monitor yard entities state in real time. For example, truck drivers can receive information about their assigned docks. Yard jockeys can receive notification on new tasks and update their states.

- **Location Service (LS):** This service allows mobile devices to be used to communicate their positions via GPS. It provides yard jockeys with accurate trailers positions. With the position information, yard manager can assign fetching tasks to the best suited yard jockey.

The YMS Variability

As mentioned in Chapter 2, variability is something that is captured and described in the problem domain. Therefore, it is usually a decision point that is visible to the customer or user. The management of variability is the key issue to be addressed by INDENICA project. As a case study of INDENICA, the YMS defined multiple variation points, which are described as follows.

The YMS uses OSGI framework[3]. An OSGI-based platform allows an easy exchange as well as extension of bundles, and thus gives users freedom to adapt the platform on demand. The base platform of the YMS provides the following tree variation points: persistence, connectivity and authentication.

- **Persistence:** For the persistence of the domain objects a relational database is used. There are two access variants available: JDBC[4] and Java Persistence API (JPA)[5].

- **Connectivity:** For the connectivity, three possibilities are provided to consume HTTP-Requests: Remote Function Call (RFC), SOAP-based services and REST.

- **Authentication:** For Authentication, the YMS uses Java Authentication and Authorization Service (JAAS) as an API which allows connecting Java-based applications with services for authentication and access rights.

Besides the variation points about base platform, the mentioned domain services also provide different variability that can be tailored to meet customer needs. YM and YJ can be extended by other services. By using MC, truck drivers and yard jockeys are allowed to use mobile devices. YM provides

[3]OSGi http://www.osgi.org/

[4]http://www.oracle.com/technetwork/java/overview-141217.html

[5]http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

two scheduling types, *next* and *fitting*, which describe how arriving trucks are scheduled. It also allows several additional functions, such as special dock types and special vehicle types. YJ can be extended with GPS-support via LS to allow truck drivers and yard jockeys to update their precise positions via mobile devices. In that case, MC must be applied. In addition, LS enables users to display their positions through textual coordinates and road map view as well as satellite map view.

3.1.2 Yard Management System as a SaaS Application

The current YMS runs on SAP NetWeaver Cloud which uses an OSGi-based JavaEE6 application server [Con12]. As mentioned above, the YMS defines multiple variation points that can be tailored to meet customer needs. During the process of providing a runnable YMS variant, various stakeholders are involved, e.g. platform providers, application providers and tenants as well as their users. According to their roles, the various stakeholders focus on different types of application characteristics. In addition, they can eliminate some configuration choices (services) in the light of their requirements. After the configuration process, different customized configuration variants can be derived. Dependencies among several stakeholders exist, for example, tenants renting the application depend on the application provider. Therefore, the configuration of the application provider has to be applied first. Due to stakeholders' different responsibilities, their possible decisions are restricted. For instance, end users are not allowed to configure the used database as it is the responsibility of the providers. Additionally, during or after the configuration process tenants can be added or removed. As a SaaS application, the YMS will be used to identify the requirements and illustrate our concepts in Section 3.2 and Chapter 4 respectively.

3.2 Requirements Identification

In the previous work, seven requirements for configuration management of cloud-based applications have already been identified [SMM+12]. Based on the requirements and the above scenario, we highlight the following concrete requirements that will be addressed by our proposed concepts.

- REQUIREMENT 1. *Specification of functional variability* As mentioned above, cloud-based applications provide different functionalities to fulfill various requirements of customers. In the YMS, the described domain services provide different variability, such as different type of data base for persistence or various views for GPS-supported location service. According to the stakeholders' objectives, the provided variability can be tailored to fulfill their need. Therefore, we need to handle the variability of functionality.

- REQUIREMENT 2. *Specification of stakeholder views* During the configuration process of cloud-based applications, various stakeholders with different objectives are involved. They can remove a certain configuration choice according to their concerns. For example, in the YMS a provider configures fundamental application properties (e.g. persistence type and connectivity type), whereas a tenant chooses only from high-level application functionality (e.g. mobile devices with different GPS views). Thus, we need a view concept defining the configuration operations which a stakeholder is allowed to perform.

- REQUIREMENT 3. *Specification of structured configuration workflow* A configuration workflow of a cloud-based application differs from other business processes. In the workflow, different configuration operations of various stakeholders are involved. In addition, some stakeholder decisions have global impact while others have only local one. In the YMS, changes made by the application provider directly impact the remaining configuration choices of other stakeholders (e.g. tenants and users). Tenants can make decisions only if the application provider's configuration is completed. Therefore, the specification for a structured configuration workflow is required to document, analyze and explain the workflow logic. As mentioned in our example, during or after the configuration process tenants can be added or removed. Therefore, the configuration process has to be capable of adding and decommissioning stakeholders at run time.

- REQUIREMENT 4. *Dynamic integration of stakeholders* During the design time of a cloud-based application, not all tenants and their users are known explicitly. As mentioned in the YMS, during or after the configuration process tenants can be added or removed. Therefore, the configuration process has to be capable of adding and decommissioning stakeholders at run time.

- REQUIREMENT 5. *Mapping between problem space and solution space* In the YMS, after a configuration is derived, the configuration needs to be integrated into a YMS instance. In order to achieve this goal, we need to map the derived configuration onto the YMS configuration. This issue refers to the mapping between problem space and solution space as introduced in Section 2.2. Therefore, a mapping used to adapt different variability expressions from different systems are needed.

In Chapter 4, we propose a dynamic configuration workflow that addresses the above identified requirements. With the help of the workflow, the illustrated configuration scenario will be modeled and simulated.

Chapter 4

Concept

In this chapter, we introduce our proposed concepts of the configuration management for multi-tenant cloud-based applications. In Chapter 3 we have illustrated a cloud-based application. Through the analysis of the example, we identified various requirements for the configuration management. Based on these requirements, we will construct our concepts.

In Section 4.1 we specify the configuration management, which is based on an *extended feature model*, a *view model* and a *configuration workflow model*. In Section 4.2 we introduce the concept of configuration workflow adaptations. With the help of the workflow adaptations, our configuration management is able to integrate dynamic stakeholders into the configuration workflow. In Section 4.3 we present an EFM mapping solution used to adapt other variability expressions from different systems to our EFM. In Section 4.4 we show the complete concepts by simulating a configuration process regarding the YMS.

4.1 Configuration Management Specification

Model-driven engineering (MDE) is a software development methodology focusing on creating domain models. As a promising approach, MDE is meant to effectively express domain concepts effectively, address platform complexity and simplify design process as well as teams working [Sch06]. Due to the improved abstraction of DSLs, the problem descriptions are much clearer and simpler. This not only increases the speed of development, but also provides clear understanding of domain concepts within the project. Furthermore, the evolution of the software is greatly simplified through the separation of the technical illustration and professional models. Therefore, we use MDE to specify our configuration management. To describe our concepts, we refer to the concepts stated in [SMM$^+$12] and we define the following three models based on the MDE.

- **Extended Feature Model (EFM)** is used to express the functionality variability and define the configuration space of cloud-based applications.
- **View Model (VM)** is used to define stakeholders and their views on the EFM.
- **Configuration Workflow Model (CWM)** is used to capture the ordering between configuration stages and stakeholders' specialization steps at run time.

These models are able to reduce the complexity of configuration process and support reuse as well as separation of concerns (SoC). In the following sections, we elaborate on the proposed models and utilize them to model the YMS example illustrated in Section 3.1.

4.1.1 Variability Modeling

In cloud computing, normally a provider hosts a cloud-based application that is rented to tenants and accessed by the users of tenants over the internet. Therefore, various stakeholders which involved in a cloud-based application can have different objectives and requirements. As one of the most popular ways in SPLE, *feature modeling* is convenient to handle the commonality and variability of a cloud-based application [MMLP09, RA11, SCG^{+}12]. Hence, utilizing an Extended Feature Model (EFM) can address Requirement 1 stated in Chapter 3.

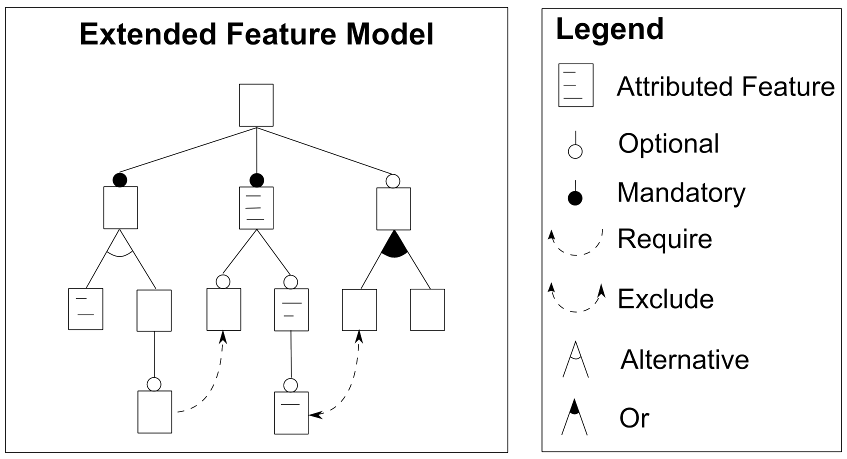


Figure 4.1: The extended feature model

As depicted in Section 2.2, an EFM is a feature model including more information about features. In our concepts, we utilize an EFM with feature attributes and group cardinalities. In the EFM, features and feature attributes depict the application functionality and functionality properties respectively. Each feature has a *selected state* with three possible values: *unbound*, *selected* and *deselected*. The selected state of a feature depends on specialization steps (see Section 2.4) of the related stakeholder. Each feature attribute has a *value* and refers to a discrete or continuous *value domain*, which defines the possible specialization steps for the feature attribute. Therefore, the functionality attributes are not just selectable or deselectable for product configurations, but further allow for a more fine-grained valuation.

We use a cardinality-based syntax to express valid ranges ($k..l$) of child features that are to be selected from groups with n features. For example, cardinality (n..n) denotes *mandatory*, (0..n) *optional*, (1..1) *alternative*, and (1..n) *or* groups. In order to specify constrains on valid product configurations, the EFM also includes crossing tree hierarchies that are given as *require* and *exclude* edges between two features. As a visual notation, a feature diagram is used to depict our EFM. The feature diagram of our EFM is shown in Figure 4.1.

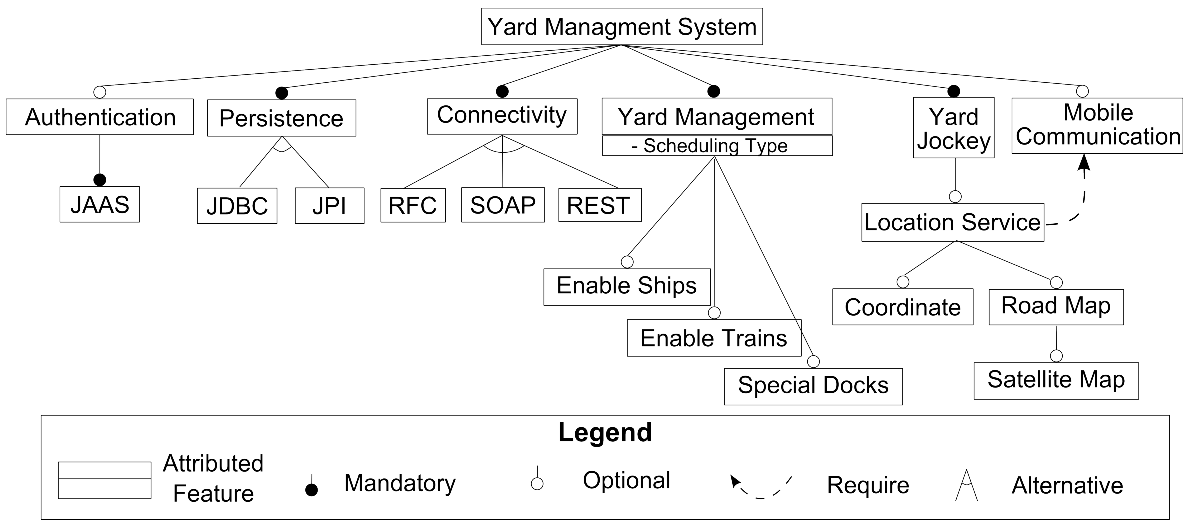


Figure 4.2: Extended feature model for the Yard Management System

As displayed in Figure 4.2, we utilize the above EFM to model the commonality and variability of our example illustrated in Chapter 3. In the example, we use Yard Management System (YMS) as the root of our feature diagram. The base platform services and domain services are presented by the features while their properties are presented by the features attributes. Among the services, several dependencies and relationships exist and are described by feature relationships and cross-tree constrains respectively. For instance, we express Yard

Management Service (YM) as a feature and the property Scheduling Type of YM as a feature attribute. The Scheduling Type has a discrete value domain: *next* and *fitting*. The *next* is used to retrieve the next possible appointment for the given time while the *fitting* is used to find an appointment that fits best.

YM has also four additional functions: enable mobile device (MC), enable train in the yard (Enable Trains), enable ship in the yard (Enable Ships), and enable special docks (Special Docks). These functions are presented by child features of YM. In contrast to YM, Yard Jockey Service (YJ) has only one child feature Location Service (LS) that allows GPS-supported location service. In order to use LS, MC must be applied. This dependency is presented by the *require* edge between MC and LS.

4.1.2 Stakeholder Views Modeling

In Section 4.1.1 we use the EFM to define the configuration space for an application. From a stakeholder's point of view, domain features organized in a feature diagram represent selectable product characteristics. According to their roles, different stakeholders may have various objectives and permissions to perform specialization steps on the EFM, i.e. every stakeholder owns a tailored configuration view that integrates stakeholder relevant concerns. Thus, a view concept is required to limit stakeholders' possible specialization steps based on their roles.

In our concepts, we define a View Model (VM) as the view concept to address Requirement 2. The VM is based on the advanced role based access control model ($RBAC_1$) and refers to the concepts in [SMM$^+$12]. As introduced in Section 2.3, $RBAC_1$ includes the RBAC base model and the concept of role hierarchies. Thus, our VM is visualized in Figure 4.3 and described as follows.

The VM includes *Stakeholders* and their *Views* on the *Specializations Steps*. A stakeholder either represents a person, a member of an organization, or a third party that is involved in the configuration process and has certain concerns regarding the configuration of parts of the EFM. It refers to a view that can be thought of a set of specialization steps on the EFM. Each specialization step allows to bind variability until a feature model is created that corresponds to a variant configuration. According to Czarnecki et al. [CHE05b], specialization steps in our EFM include refinement of group cardinalities, selection and deselection of features, as well as setting attribute values.

The VM allows the concept of role hierarchies, i.e. *Inheritance*. Thus, stake-

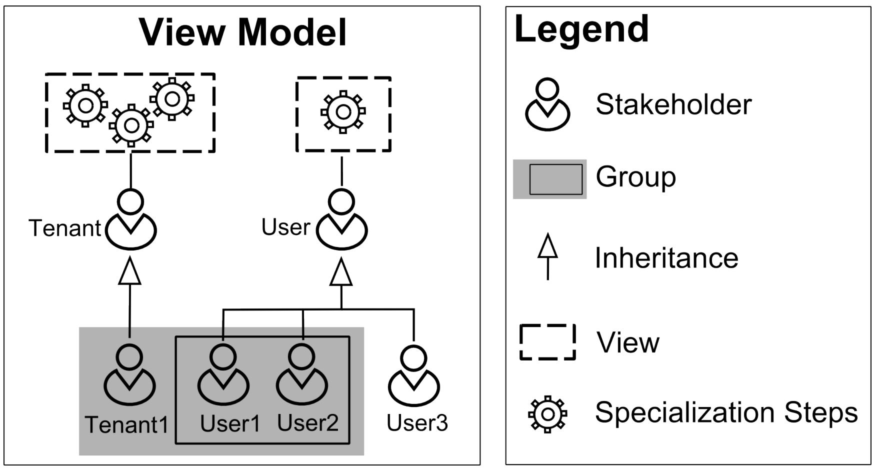


Figure 4.3: The view model [SMM+12]

holders can inherit views from other stakeholders. For example, in Figure 4.3, *Tenant* and *User* are stakeholders that have different views. *Tenant 1* inherits the views from *Tenant* while *User 1* and *User 2* inherit the views from *User*. *Tenant* and *User* are called *Stakeholder Types* that directly refer to views. A stakeholder type is defined according to the applications and corresponds to a stage in the Configuration Workflow Model (see Section 4.1.3). All other stakeholders should inherit from stakeholder type to obtain a view on the specialization steps. For instance, a new stakeholder *User 3* arrives and should inherit from *User* to obtain the view. Therefore, a stakeholder's view depends on its stakeholder type's view.

With the help of stakeholder types, we can use inheritance relation to identify the different users that are in the same configuration stage but from different organizations. For example, two stakeholders *User 1* and *User 2* from different organizations inherit view of the same stakeholder type *User* and have the same view on the EFM.

Besides, *Group* is used to define VM. A group describes the membership relation among stakeholders. It consists of a *Leader* and a set of *Members*. For example, in Figure 4.3 *Tenant 1*, *User 1* and *User 2* compose a group, in which *Tenant 1* is the group leader and *User 1* as well as *User 2* are group members. In a group, the group members' specialization steps are affected by the group leader's specialization steps. On the one hand, the group members can not perform specialization steps until the group leader finishes its specialization steps. On the other hand, the possible specialization steps of the group members depend on not only their stakeholder type, but also the specialized feature model yielded by their group leader.

As mentioned above, multiple stakeholders can be involved in a configuration process of a cloud-based application. Commonly, stakeholders' configuration ordering is determined according to the application. At the run time, each newly arriving stakeholder inherits from the related stakeholder type, and thus obtains a view on the EFM.

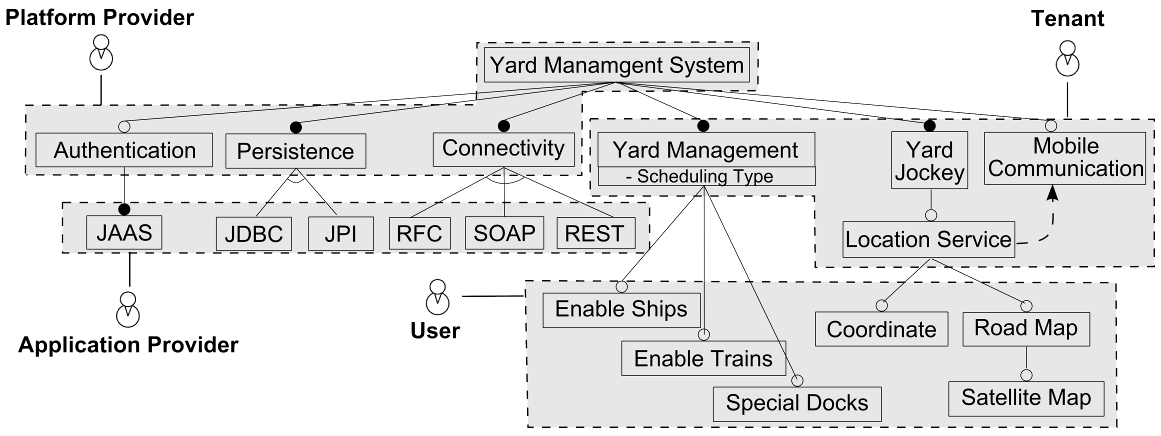


Figure 4.4: Views for stakeholders in the Yard Management System

With the help of the VM, views of stakeholders for the illustrated example can be modeled. As depicted in Figure 4.4, four kinds of stakeholder types are involved and have various views on the EFM of the YMS. In Figure 4.4, *Platform Providers* and *Application Providers* focus on the fundamental platform services and make global pre-configurations valid for all *Tenants* and their *Users*. After platform providers and application providers finish performing specialization steps, tenants and their users can eliminate left-over configuration choices and eventually leading to a configuration.

4.1.3 Configuration Workflow Modeling

In the configuration process of a multi-tenant cloud-based application, different stakeholders are involved and apply specialization steps. Some stakeholders can perform the operations concurrently, but others must perform the operations in a certain order. Commonly, the stakeholder operations order depends on the their stakeholder types. For example, in Section 4.1.2 we display the view model of the YMS, where four stakeholder types are involved. Among the stakeholder types, tenants can perform the specialization steps only if the associated providers complete their operations. Therefore, a configuration process can be composed of a sequence of configuration stages. Each configuration stage is defined according to different stakeholder type with different views.

In order to capture the order within the configuration stages and stakeholders' specialization steps, we define a Configuration Workflow Model (CWM) to model a structured and staged configuration process. The defined CWM can address Requirement 3.

To specify the CWM, we utilize the UML Activity Diagram[1] (due to its widespread use in both academia and practice) as our workflow language (see Section 2.5.2). Activity diagrams are UML behavior diagrams which are intended to model both computational and organizational processes (i.e. workflows) [RJB04]. An activity diagram is constructed from a limited number of shapes, connected with arrows. The details of the elements in activity diagrams can refer to [OMG11]. In this section we introduce elements of activity diagram utilized and the configuration workflow constructed by these elements.

As introduced in Section 2.5, a workflow model comprises a set of nodes (representing start/end nodes, tasks or control connectors) and a set of control edges between them. According to the concept, we model our CWM based on activity diagram as follows:

- **Activity:** A configuration workflow is represented by an activity, which consists of all other elements in the workflow. An activity has a tree-like structure. Each activity node has only one input edge except the *Idle action* which will be introduced below.

- **Initial Node:** An initial node is a starting point for executing a configuration workflow. A configuration workflow must have only one initial node. When a configuration workflow is executed, a feature model as a data object is imported via the initial node.

- **Flow Final Node:** A flow final node is a final node that terminates a flow of a configuration workflow. A configuration workflow may have multiple flows and flow final nodes. When a flow completes, a complete configuration is provided.

- **Activity Final Node:** An activity final node is a final node that stops all flows in a configuration workflow. A configuration workflow must have only one activity final node. When the workflow is completed, all of actions stop and change their state into *completed*.

- **Action:** An action is a single step within an activity and represents a set of specialization steps in the configuration workflow. Each action refers

[1]http://www.omg.org/

to a stakeholder and is performed by the stakeholder. When stakeholder performs specialization steps, the related action takes a feature model as an input and yields a specialized feature model as an output. Thus, the possible specialization steps concerning the action depend on the related stakeholder's view and configuration choices of the input feature model. In addition, the yielded feature model is stored locally. If an action has successors, the duplicate of the feature model is propagated to the successors. In our workflow, we use the action name to symbolize the stakeholder name.

- **Idle Action** During the configuration process, new stakeholders can be dynamically integrated into the workflow. Therefore, the workflow state should not become *completed* immediately, when all actions in the workflow are complete. In our workflow, an idle action is specified to keep the workflow in the state *suspended* (see Section 2.5.1), when all actions in the workflow are complete. A configuration workflow includes only an idle action, which is connected with the activity final node. We define that the idle action can be performed by a third party stakeholder. When the stakeholder performs the idle action, the workflow is complete.

- **Fork Node:** A fork node is used to split a flow into multiple concurrent flows. Through the fork node, each successive action can obtain a specialized feature model from the previous action.

- **Control Flow:** A control flow is an edge that starts an activity node (start/final node, action, join/fork node) after the previous one is finished. It is used to show the order that actions will be performed in the workflow.

According to the above specification, a configuration workflow about the YMS is depicted in Figure 4.5. In Figure 4.5, the workflow includes four kinds of stakeholder types (i.e. *Platform Provider*, *Application Provider*, *Tenant* and *User*). According to Czarnecki et al. a configuration stage can be defined in terms of roles (see Section 2.4). Thus, in our configuration workflow, each stakeholder type reflects a configuration stage. The stakeholders in the same stage have the same view on the EFM, but differ from one another in their organizations, such as *Tenant 1* and *Tenant 2*.

Figure 4.6 depicts a staged configuration flow involved in Figure 4.5. In the flow, each stakeholder performs the specialization steps according to its views on the EFM. The views of the stakeholders are defined in VM. In addition, after a stakeholder finishes its specialization steps, a partial configuration is derived and propagated to its succeeding stakeholder. Therefore, the special-

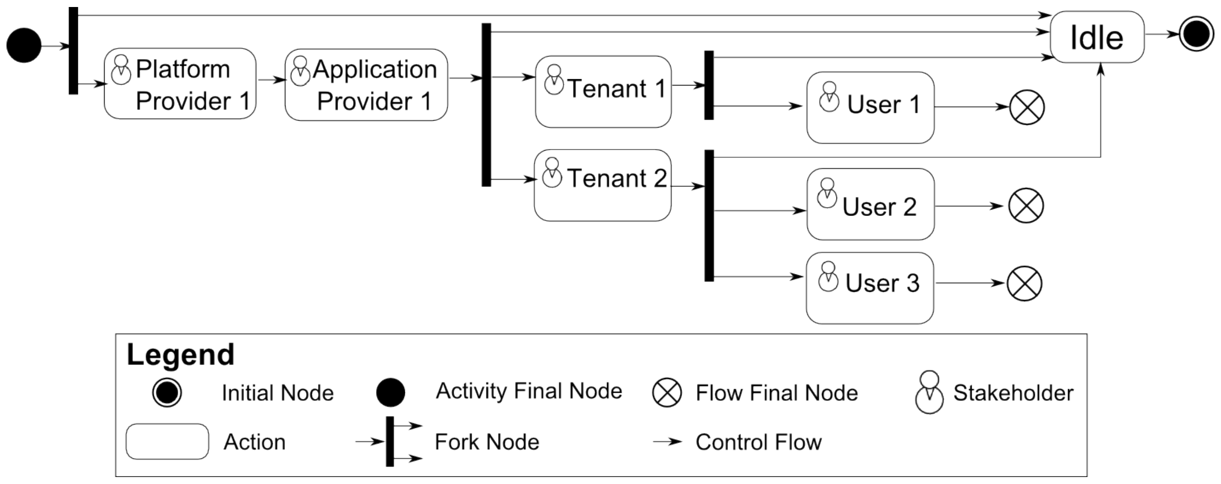


Figure 4.5: A configuration workflow about the Yard Management System

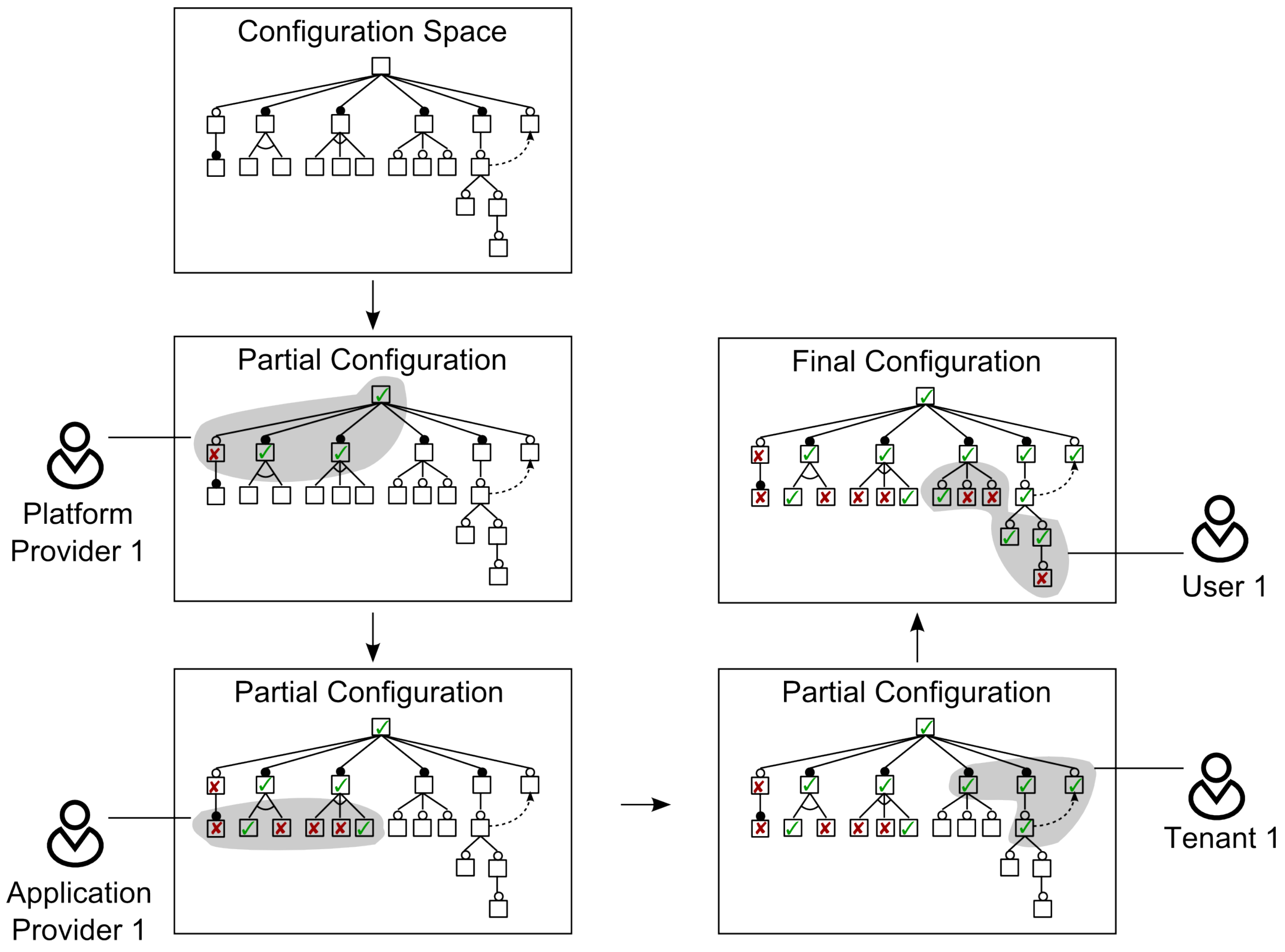


Figure 4.6: Staged configuration concerning Yard Management System

ization steps of preceding stakeholder affects the succeeding stakeholder.

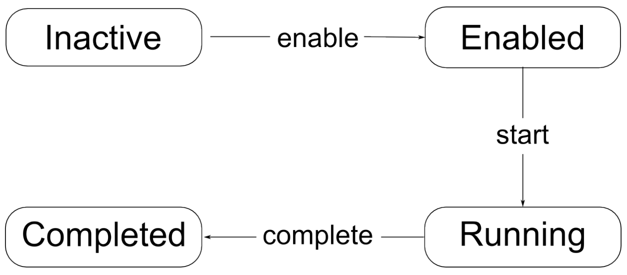


Figure 4.7: State transitions of an action instance

As mentioned in Section 2.5.1, a workflow instance and an action instance have life cycles separately. In our CWM, we define that a workflow instance has the same life cycle as described in Section 2.5.1. In addition, we define that each action instance has four states in its life cycle as depicted in Figure 4.7. When the state of its previous action instance is *completed*, the action instance state changes from *inactive* to *enabled*. For example, when *ApplicationProvider1* is completed, the state of *Tenant 1* changes from *inactive* to *enabled*. If the stakeholder associated with the action instance performs the specialization steps, the action instance state changes to *running*. When an action instance completes, its state changes to *completed*.

In order to describe the control flow constructs of our configuration workflow, we utilize two control flow patterns which are depicted in Figure 4.8. In Figure 4.8, we use event diagrams (as proposed in [Wes12]) to describe the semantics of selected control flow patterns. The *sequence pattern* expresses that an action B becomes enabled after an action A has completed. The *parallel split pattern* allows splitting the control flow into multiple flows which are then concurrently executed. In the *parallel split pattern*, after completing action A, both action B and C become *enabled*. In our configuration workflow, when a workflow instance is cerated and then executed, the action instances, which are connected to the initial node, change their state from *inactive* to *enabled*. Then the workflow instance runs along with the concepts of *control flow patterns*. When a third party stakeholder arrives and performs the *Idle action*, the workflow will be completed.

The basic concepts of our configuration management are represented by the combination of EFM, VM and CWM. At design time, the three models can describe the configuration process for a multi-tenant cloud-based application. At runt time, a workflow instance is created and then executed, but the workflow instance can not support the dynamic integrating of stakeholders. In Section

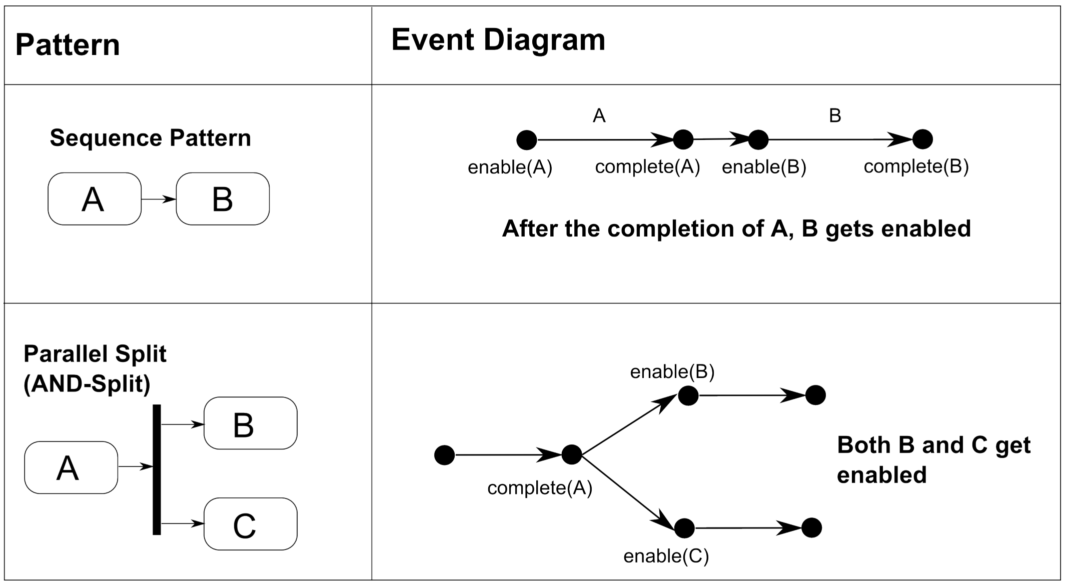


Figure 4.8: Control flow patterns [RW12]

4.2 we will introduce our solution for the above issue.

4.2 Configuration Workflow Adaptations

In Section 4.1.3 we defined the configuration workflow model to specify the configuration process. The model allows the stakeholders, which are known during application design time, to apply specialization steps. But new customers may be added dynamically at run time. Therefore, a workflow adaptation mechanism is required for the configuration workflow to support the dynamic integrating of stakeholders. In this section, we describe our concept that enables the configuration workflow to dynamically change at run time. This concept is used to address Requirement 4.

As introduced in Section 2.5, a workflow is also a directed graph. During the configuration process, newly arriving stakeholders change the workflow model, i.e. the graph, and thus *graph transformation* (see *section Graph Transformation*) is a suitable solution to the dynamic workflow modification.

The idea of graph transformation is to apply a rule to a graph and create a new graph from the original one. The rule defines an original graph (*left-hand side*) and a goal graph (*right-hand side*). It can transform a graph by replacing the occurrence of the left-hand side with the right-hand side. In addition, a rule can include a set of *application conditions* that determine whether or not the transformation is applied. There are two advantages to use graph transformation as the solution to workflow adaptation. On the one hand, it is convenient and simple. We only need to define the rules according to the predefined stakeholder types and the rules are able to determine the inserting positions as well as the configuration orders of arriving stakeholders in the workflow. On the other hand, it is very flexible. In different situations we can define different rules to realize the workflow adaptation and some of the rules are able to be reused. Therefore, we utilize the concepts of *graph transformation* to realize the configuration workflow adaptation. Based on the example of YMS, we display our defined rules in the following parts.

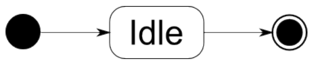


Figure 4.9: The initial configuration workflow

Before we design the rules, we first define the initial configuration workflow, to which our rules are applied. Figure 4.9 depicts the initial configuration workflow that consists of an initial node, an idle action, an activity final node and two control flows (see Section 4.1.3). In the initial configuration workflow, the idle action has the state *enabled*. The goal of the idle action is to keep the workflow in the state *suspended* after all actions in the workflow are performed.

In our example, we define four rules based on the involved stakeholder types (i.e. platform provider, application provider, tenant and user) as depicted in Figure 4.10. Each rule is used to integrate a type of stakeholder into the workflow. A rule consists of a *left-hand side*, a *right-hand side* and a set of *application conditions*. In the rules, A, B, C and D are variables that is used to express the stakeholders. The left-hand side presents the graph to be searched in the given graph. It may occur many times in the given graph. If an occurrence of the left-hand side is found, the application conditions will be checked. In the application conditions we define two relationships *inherits from* and *belongs to*, which are respectively used to represent the *inheritance* and *group* in VM. For example, in rule (d), *C inherits from tenant* depicts that the arriving stakeholder C should inherit from the stakeholder *tenant*. *D belongs to C* depicts that the arriving stakeholder D is a member of a group, in which the exiting stakeholder C is the leader. When the application conditions are satisfied, the rule will be applied. As a result, the occurrence of the left-hand side in the given graph is replaced by the right-hand side.

In Figure 4.10, rule (a) is used to integrate a new platform provider in the workflow. When the left-hand side of rule (a) is found in the given workflow, the application condition will be checked. If the arriving stakeholder is platform provider, rule (a) is applied to the given graph. Rule (b) and (c) are

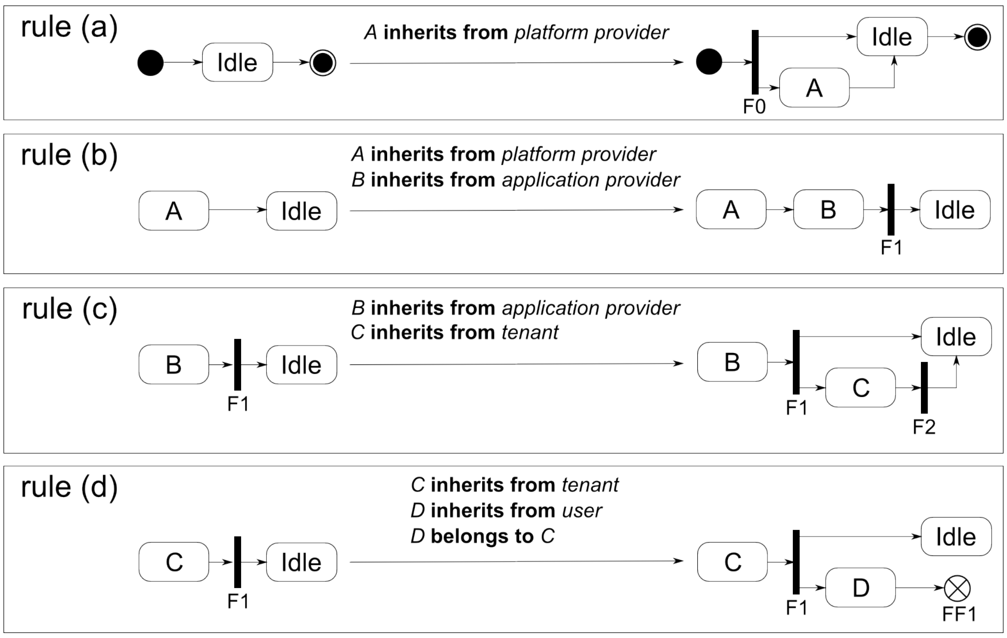


Figure 4.10: Rules of the graph transformation

separately used to integrate a new application provider and a new tenant in the workflow. Both rules have similar structure but differ in application conditions. Rule (d) is used to handle a new user. After rule (d) is applied to the given graph, a new user is added into the workflow. At the end of the flow, a complete configuration will be derived. In our YMS only one platform provider and one application provider are involved, our rule (b) and (c) do not consider the group relationship.

The replacement of the occurrence of the left-hand side by the right-hand side changes the workflow model. As introduced in Section 2.5.4, two approaches can be used to modify workflow model. One approach is *change primitives* that operate on single elements of a workflow mode at a low abstract level. The possible operations of change primitives are *add node*, *delete node*, *add edge* and *delete edge*. The node, which is added or removed, can be an initial node, a final node, a fork node or an action. Another approach is *high level change operations* that combine a set of primitives to enable changes of a workflow. As high level change operations, there are already several predefined adaptation patterns, such as *move process fragment* and *replace process fragment*. Since our defined rules are simple, in our concepts we utilize change primitives to replace the occurrence of the left-hand side with the right-hand side. For each rule, we design a collection of ordered change primitives as depicted in Table 4.1. When a valid rule is found, its change primitives will be performed in a certain order.

Rules	Change Primitives
rule (a)	01: delete edge from Initial Node to Idle 02: add node A 03: add node F0 04: add edge from Initial Node to F0 05: add edge from F0 to Idle 06: add edge from F0 to A 07: add edge from A to Idle
rule (b)	01: delete edge from A to Idle 02: add node B 03: add node F1 04: add edge from A to B 05: add edge from B to F1 06: add edge from F1 to Idle
rule (c)	01: add node C 02: add node F2 03: add edge from F1 to C 04: add edge from C to F2 05: add edge from F2 to Idle
rule (d)	01: add node D 02: add node FF1 03: add edge from F1 to D 04: add edge from D to FF2

Table 4.1: Change primitives for the rules in Figure 4.10

According to the the rules and changes primitives above defined, we display the adaptation flowchart of a configuration workflow in Figure 4.11. When a stakeholder arrives and wants to join in the workflow, it has to enter its type and name as the input parameters, e.g. its type is *User* and name is *User 1*. If necessary, it also needs to enter a name of a existing stakeholder, whom it wants to request services from, e.g. the input name is *Tenant 1*.

After that, the workflow starts to search the valid rules. It first searches the occurrence of left-hand side of a rule. The given workflow may have multiple occurrences of left-hand side of the rule. If its left-hand side occurs in the graph, the workflow checks the application conditions of the rule. If the conditions are fulfilled, the rule is valid and its change primitives will be performed. The workflow iterates the search process until all rules are checked.

During the searching process, if one rule is valid, VM will create a role if the

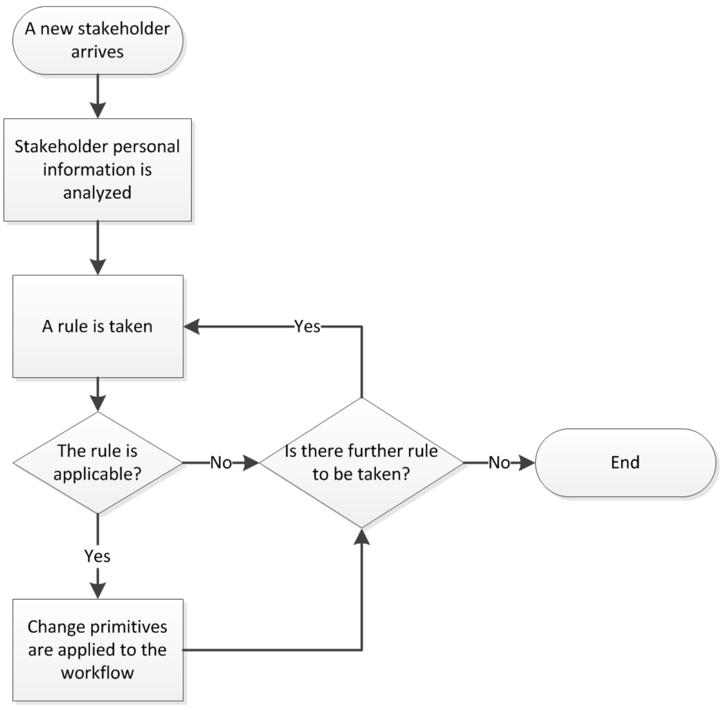

A new stakeholder arrives
Stakeholder personal information is analyzed
A rule is taken
Yes
The rule is applicable?
No
Is there further rule to be taken?
No
End
Yes
Change primitives are applied to the workflow

Figure 4.11: The adaptation flowchart of a configuration workflow

stakeholder is not included. The created role inherits from the stakeholder's type and assigned to the stakeholder. Then, VM will put the arriving stakeholder into a group as a member, in which the demanded stakeholder is the leader.the added actions need to be handled as follows. Besides, the added actions must refer to the arriving stakeholder. In addition, the state of an added action depends on its preceding action. If its preceding action has the state *completed*, the state of the added action is *enabled*. If the preceding action has other state (i.e. *inactive*, *enabled* and *running*), the added action's state is *inactive*. Finally, after all rules are checked, the procedure of the workflow adaptation is terminated.

Figure 4.12 shows an example using the above defined rules and flowchart to realize the graph transformation. In the example, the given workflow includes six stakeholders *Platform Provider 1*, *Application Provider 1*, *Tenant 1*, *Tenant 2*, *User 1* and *User 2*. When a new user *User 3* arrives, it enters its type *User* and its group leader name *Tenant 2*. Then the workflow starts to search the valid rules. There are three occurrences of the left-hand sides of rule (c) and (d) in the workflow: $action_{ApplicationProvider1} \rightarrow action_{Idle}$, $action_{Tenant1} \rightarrow action_{Idle}$ and $action_{Tenant2} \rightarrow action_{Idle}$. Then, the workflow checks the application conditions of rule (c) and (d). Since the arriving stakeholder is a user, the application condition *D inherits from user* in rule (d) is

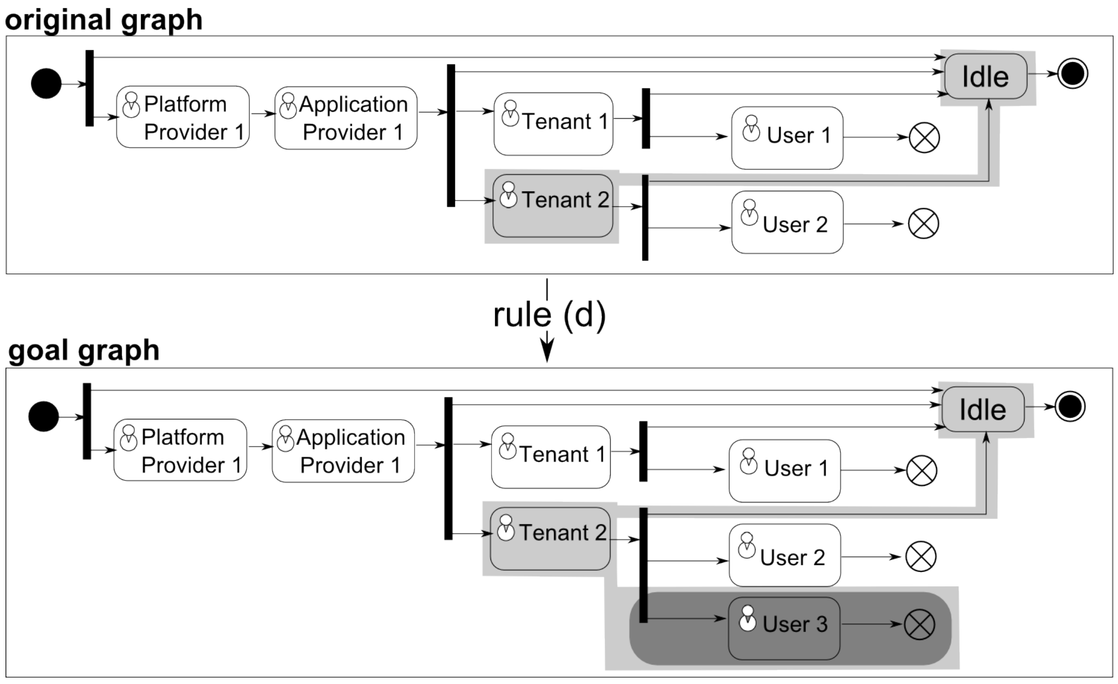


Figure 4.12: Example about using change primitives for User 2

fulfilled whereas the application condition *C inherits from tenant* in rule(c) is not fulfilled. According to the other two conditions in rule (d), the occurrence $action_{Tenant2} \rightarrow action_{Idle}$ is satisfied, which is covered by the gray area in the top graph of Figure 4.12. Therefore, the rule (d) should be applied. After the performance of the change primitives, the occurrence the left-hand side of rule (d) (colored by gray in the top of the graph) is replaced by the right-hand side of rule (d) (colored by gray in the bottom of the graph). In Figure 4.12 the new added nodes and edges are colored by dark gray..

The graph transformation rules with their change primitives compose our concepts of the configuration workflow adaptations. According to different situations, different graph transformation rules have to be defined. The defined rules are not only used at run time but also used to construct the workflow at design time so that they are capable to be utilized by the workflow instance. The three models (EFM, VM and CWM) as well as the concepts of the workflow adaptation enable the configuration workflow to dynamically integrate a newly arriving stakeholder. During the configuration process, a complete configuration is created at the end of a flow. This configuration should be utilized by the cloud-based application. Therefore we need an interface for the configuration management to allow the application to recognize the configuration. In next section we will introduce our concept for the interface based on the YMS example.

4.3 Mapping between Problem Space and Solution Space

Usually, a cloud-based application provides multiple configurable functionality to meet customers' different requirements. A customer can select functionality in accordance with their need and then a configuration is created. According to the configuration, the application supplies the customer with the required functionality. Based on the above proposed configuration workflow, a configuration represented by a specialized EFM is produced. To use the configuration, the cloud-based application must be able to recognize the EFM. Therefore, a mapping is required between our EFM (problem space) and the application functionality (solution space). To different applications, the ways of mappings are different. In this section, based on the YMS example, we introduce our mapping concept that addresses Requirement 5.

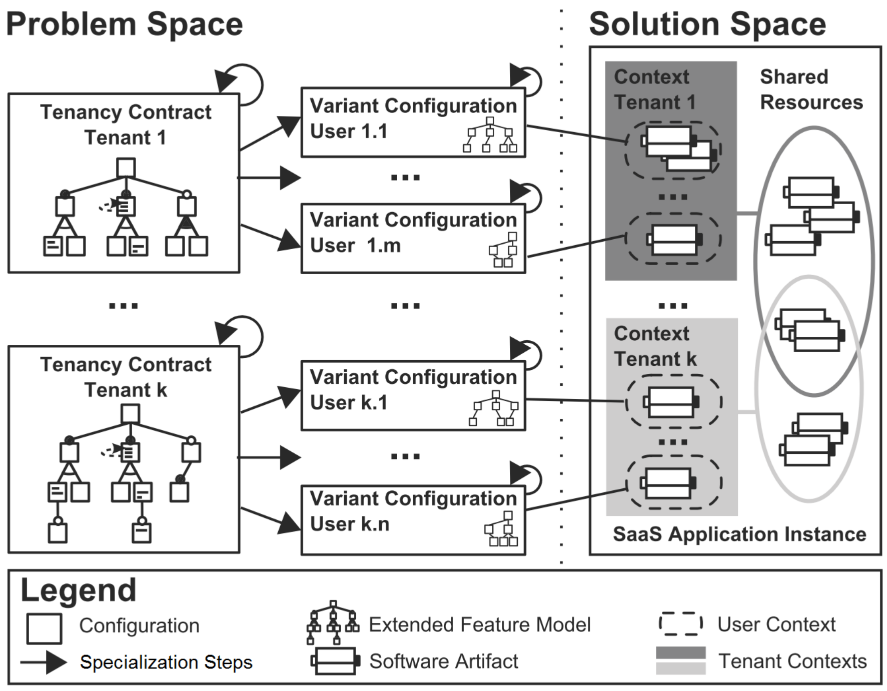

Problem Space
Solution Space
Tenancy Contract Tenant 1
Variant Configuration User 1.1
Variant Configuration User 1.m
Tenancy Contract Tenant k
Variant Configuration User k.1
Variant Configuration User k.n
Context Tenant 1
Shared Resources
Context Tenant k
SaaS Application Instance
Legend
Configuration
Specialization Steps
Extended Feature Model
Software Artifact
User Context
Tenant Contexts

Figure 4.13: Configuration and instantiation of a cloud-based application [SMM+12]

Figure 4.13 shows configuration and instantiation of a cloud-based application. It describes a mapping between *Problem Space* and *Solution Space* (see Section 2.2) for the cloud-based application. In the problem space, *tenancy contracts* define the application functionality as well as functionality properties, which are rented by tenants. A tenancy contract is created by a tenant via specialization steps and builds the basis for deriving multiple *variant configurations* for a tenant's users. In a user's variant configuration, all variability is bound. In the solution space, user variant configurations are instantiated as *user contexts* at runtime. The set of all user contexts describes a virtual *tenant context*. Those tenant contexts are integrated into the same application instance and share resources (e.g. software, hardware, databases). Thus, variant configurations are independent in the problem space whereas become dependent in the solution space.

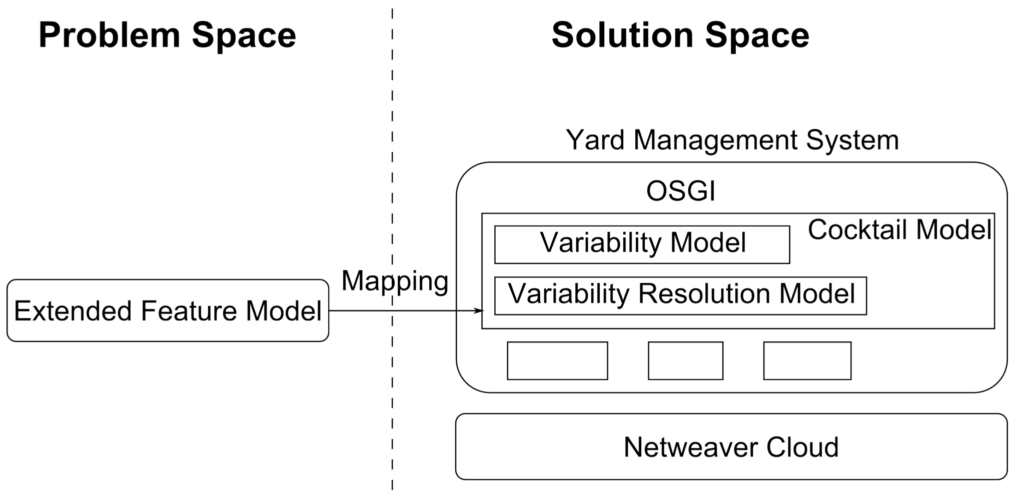


Figure 4.14: Mapping between extended feature model and Yard Management System variability model

Figure 4.14 displays the mapping between our EFM and the YMS models. In our concepts, we use EFM to express the variability in the problem space. Thus, the user's variant configuration is expressed by a specialized EFM. In addition, we utilize a staged configuration, which is realized by CWM, to create the user's variant configuration. The YMS uses OSGI framework and runs on SAP NetWeaver Cloud which uses an OSGi-based JavaEE6 application server. It utilizes an internal *Cocktail Model* to describe the variability. Cocktail Model consists of *Variability Model* and *Variability Resolution Model*. YMS uses variability model to represent the variation points in YMS. Furthermore, it uses variability resolution model as a user's variant configuration to bind all variabilities to specific values. Therefore, we can match our configuration (a fully specialized EFM) with the variability resolution model in YMS to indirectly realize the mapping between the EFM and the application functionality. In the following part we first introduce the variability model as well as the variability resolution model in the YMS, then we show the mapping between our configuration and the variability resolution model.

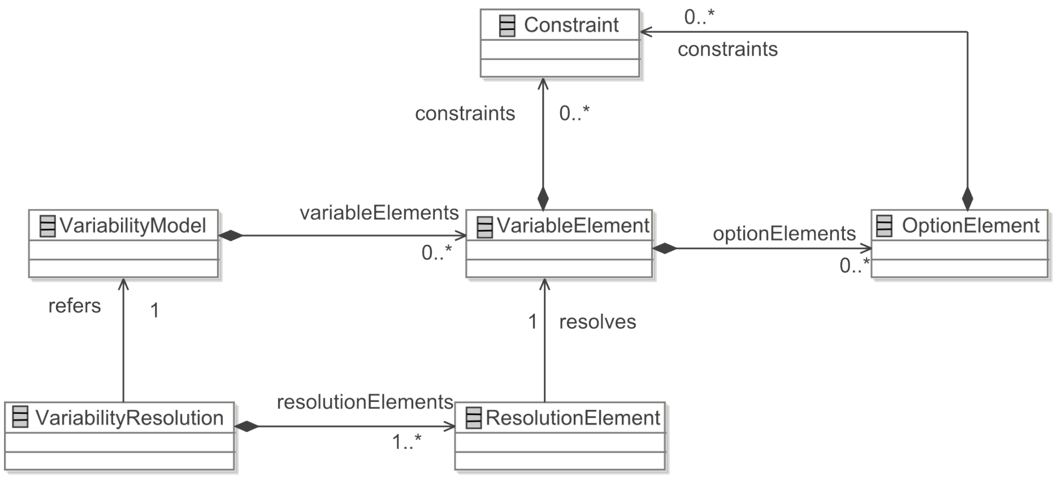


Figure 4.15: Cocktail Model in the Yard Management System

Figure 4.15 depicts the variability model and the variability resolution model in the YMS. The variability model includes a *VariabilityModel*, which is a container of a collection of *VariableElement*. Each *VariableElement* describes a variation point of an application (i.e. a functionality or functionality property) and may contain several *OptionElements*. The *OptionElements* are used to depict the value range of a *VariableElement* if necessary. A *Constraint* is considered as a relationship or a constrain among the *VariableElements* and *OptionElements*. A *VariabilityResolution* can refer to a *VariabilityModel*. It consists of a set of *ResolutionElements*, which can resolve the referred *VariableElements*, e.g. it can determine that functionality is selected or not.

A derived user's variant configuration in our concepts is a fully specialized EFM. In the configuration, all variability is bound. Therefore, in order to map our configuration onto the variability resolution model, we need to match the *features* and *feature attributes* in EFM with the *VariabilityElements* in the variability model of YMS. In addition, we have to match the *feature state* and the *feature attribute value* with the *ResolutionElement*.

After users perform specialization steps, fully specialized EFMs are derived through the configuration workflow and then transformed into variant configurations of YMS, which are represented by variability resolution model. The variant configurations are instantiated as user contexts, which are integrated into the same application instance. According to the user contexts, the application instance provides various services to the users. In next section, we will illustrate the complete process for our configuration management with the

YMS.

4.4 Configuration Process Simulation

In this section, we display the complete configuration management concepts through simulating a configuration process with regard to YMS. This simulation consists of two steps that are applied at design time and run time respectively. In the following part, we introduce the simulation in the above two phases.

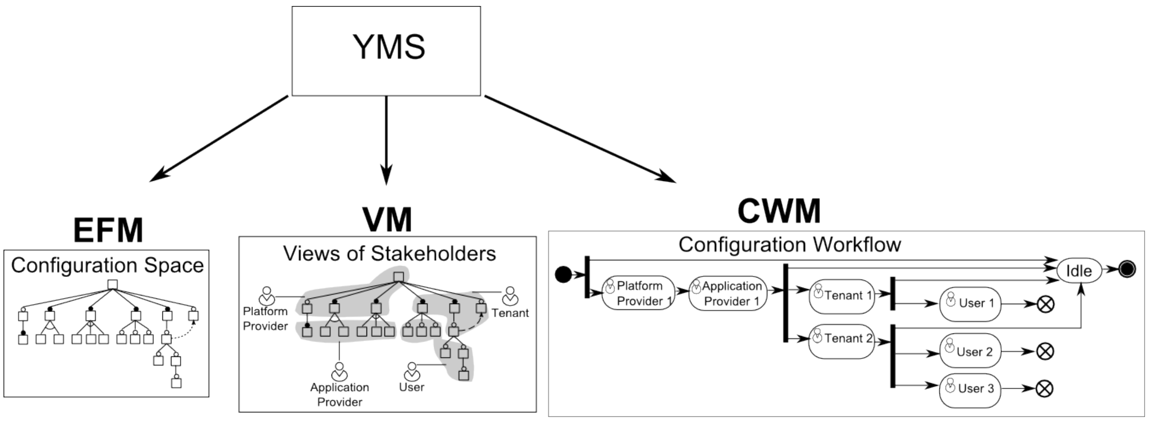


Figure 4.16: Configuration management specification

Figure 4.16 depicts the work that has to be prepared for the configuration process at design time. First, we define the EFM that describes the variability of the YMS. In Section 4.1.1 we have already modeled an EFM for the YMS that is depicted in Figure 4.2. Secondly, in the light of the EFM and YMS we establish the VM, which depicts the allowable specialization steps on the EFM for each involved stakeholder type. A predefined VM for the YMS and the EFM is displayed in Figure 4.3 in Section 4.1.2. Finally, according to the known stakeholders, we design the configuration workflow using the defined graph transformation rules for the YMS in Section 4.2.

The construction of the workflow begins with the initial workflow as depicted in Figure 4.9. Each known stakeholder should be added in the VM and inherit its stakeholder type. In addition, if the stakeholder is a group member of its preceding stakeholder, the group relationship in the VM should be updated. In the simulation, we utilize the workflow described in Figure 4.5 as the constructed workflow at design time. During the design time, an EFM, a VM as well as a CWM are derived. These three models enable stakeholders to perform specialization steps during the configuration process at run time.

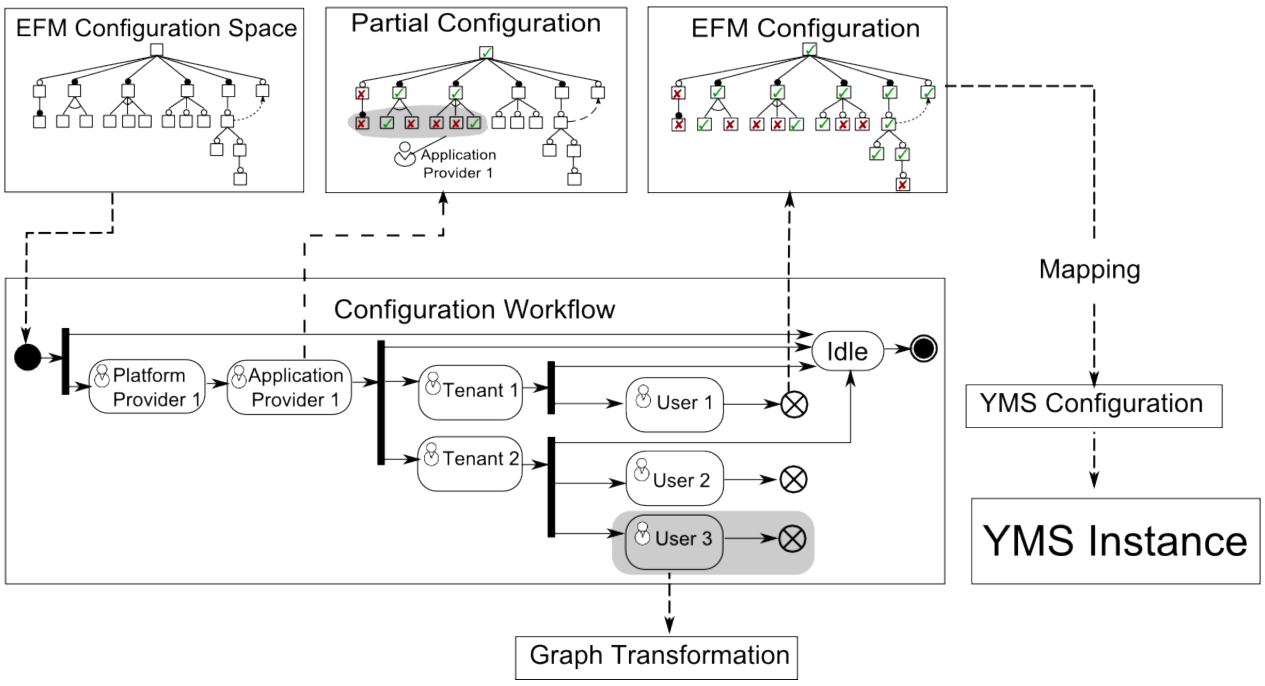


Figure 4.17: Configuration management execution

As shown in Figure 4.17, at run time, an instance is created from the above constructed workflow and then executes. All actions contained in the workflow have the state *inactive*, except that the idle action has the state *enabled*. At the beginning, the EFM is imported into the workflow instance from the initial node. When the workflow instance executes, the actions connecting to the initial node change their state from *inactive* to *enabled* and obtain the imported EFM.

During the configuration process, the stakeholders can perform specialization steps if their referred actions' state is *enabled*. When they perform specialization steps, the state of their referred actions changes from *enabled* to *running*. After they complete the specialization process, partial configurations (i.e. specialized EFMs) are propagated from their preceding stakeholders' actions to their actions. The allowable specialization steps of stakeholders depend on the stakeholders' permissions described in the VM and the specialized EFM from their preceding stakeholder in the workflow. After the stakeholders finish their specialization steps, the derived partial configurations (i.e. specialized EFMs) are stored locally and the related actions change their state from *running* to *completed*. The succeeding actions of those completed actions change their state from *inactive* to *enabled*.

At run time, a new stakeholder can arrive dynamically. For example, in Figure 4.17 a new stakeholder *User 3* arrives. If the stakeholder wants to take part in the workflow, it has to give its type (e.g. the type of *User 3* is *user*), and its preceding stakeholder type and group leader (e.g. the preceding stakeholder type is *tenant*, the group leader is *Tenant 2*), if available. According to the input data, the workflow searches the valid graph transformation rules and apply them. If rules are not found valid, e.g. a tenant arrives when the workflow has no application provider yet, the workflow will not change.

After the valid rules are applied, new actions are added into the workflow. In the workflow of Figure 4.17, a valid rule is found for the arriving stakeholder *User 3*, and thus an action is added into the workflow. An added action should refer to the related stakeholder and initialize its state. If its preceding action has the state *completed*, the state of the added action is *enabled*. If the preceding action has other state (i.e. *inactive*, *enabled* and *running*), the added action's state is *inactive*.

At the end of a flow, an EFM-based configuration is derived. The configuration must be transformed into the valid configuration of the application (e.g. YMS configuration). The transformed configuration are integrated into the application instance. According to the user's variant configuration, the application instance provides the specified services to the user.

During the configuration process, the idle action keeps *enabled* all the time. It waits for the termination activity from a stakeholder. When the configuration process is invalid (e.g. application instance do not run any more), a stakeholder applies a termination activity and the configuration workflow is terminated.

The proposed concepts of the configuration management for cloud-based application enable us to design and execute a configuration workflow. Furthermore, they also allow integrating dynamic stakeholders at run time. In Chapter 5 we introduce our configuration management tooling which implements the proposed concepts.

Chapter 5

Implementation

In this chapter, we introduce our configuration management tooling for cloud-based applications, which implements the concepts depicted in Chapter 4. Our tooling uses Eclipse[1] as the open source platform and software development environment. Eclipse is an open source community dedicated to developing open development platforms and products. By virtue of an highly flexible plug-in mechanism, the Eclipse Platform is easily extensible. In the development of the tooling, we utilize three application frameworks developed by the Eclipse community: Eclipse Modeling Framework (EMF)[2], Eclipse Modeling Framework Text (EMFText)[3] and Java Workflow Tooling (JWT)[4].

Based on EMF, the Extended Feature Model (EFM) and the View Model (VM) are modeled. In the previous work, the EFM and VM are already defined.[5] In addition, two textual editors are separately developed for EFM and VM with the help of EMFText. We integrate the two textual editors into our tooling so that the tooling allows users to define the application variability and stakeholders as well as their views on the application variation points using domain specific languages.

JWT provides us an EMF-based workflow editor, which is shown in Figure 5.1. By extending the meta model of JWT Workflow Editor (JWT WE), we construct the Configuration Workflow Model (CWM) and build our Configuration Workflow Editor (CW editor). This editor is used to show and execute the configuration workflow. Users can also use the editor to specify the workflow to be executed. When the workflow executes, users can perform the staged configuration process. Furthermore, the editor also supports the dynamic inte-

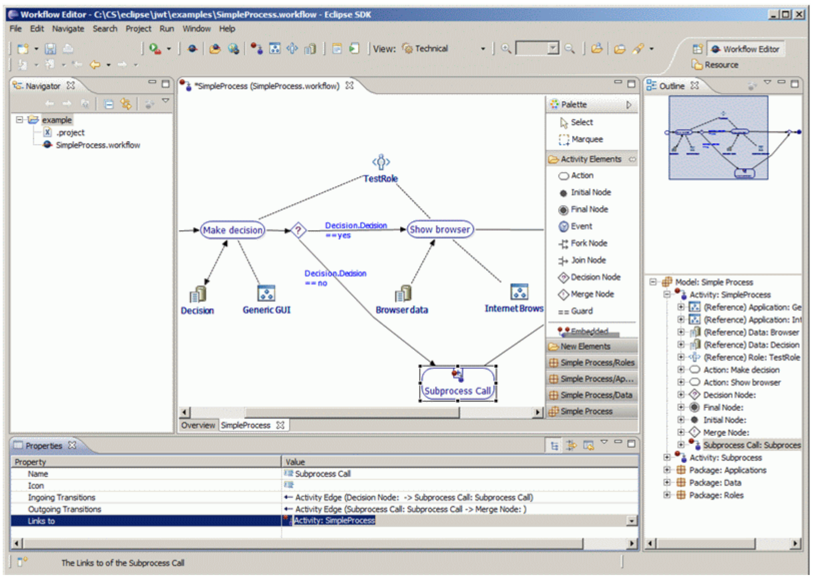
[1]http://www.eclipse.org/

[2]http://www.eclipse.org/modeling/emf/

[3]http://www.emftext.org/index.php/EMFText

[4]http://www.eclipse.org/jwt/

[5]https://github.com/extFM/extFM-Tooling



Property	Value
Name	Subprocess Call
Icon	
Ingoing Transitions	Activity Edge (Decision Node: -> Subprocess Call: Subprocess Call)
Outgoing Transitions	Activity Edge (Subprocess Call: Subprocess Call -> Merge Node:)
Links to	Activity: SimpleProcess

The Links to of the Subprocess Call

Figure 5.1: Java Workflow Tooling Workflow Editor

grating stakeholders by searching and applying the valid graph transformation rules that are predefined. When a configuration is derived during or after the configuration process, the tooling is able to transform the EFM configuration into YMS configuration, which will be integrated into the YMS instance. The CW editor, together with the editors of EFM and VM, compose our configuration management tooling.

In Section 5.1, we specify the predefined models by using EMF. By extending the meta model of JWT WE we construct our CWM. In Section 5.2, we introduce the implementation of the graph transformation rules which are defined in Section 4.2. In Section 5.3, we show the mapping method that is used to map EFM configuration onto YMS configuration. In Section 5.4, we display the configuration workflow tooling and its functionality. In Section 5.5, we evaluate the usability of our proposed concepts by performing the configuration process regarding a cloud-based application.

5.1 Configuration Specification

EMF is a modeling framework and code generation facility that is able to create systems. In EMF these systems are called *core models*. The metamodel [AK03] of all core models is called *Ecore*. Basically, Ecore is a sub-set of UML[Fow97] Class diagrams [MH06]. From Ecore metamodels, EMF provides tools to produce a set of Java classes, together with a set of adapter classes for viewing and command-based editing of the model, and a basic editor. By utilizing EMF, the meta models of EFM, VM and JWT WE are modeled. In this section, we abstractly introduce the above three models that are relevant to our tooling.

5.1.1 Extended Feature Model Specification

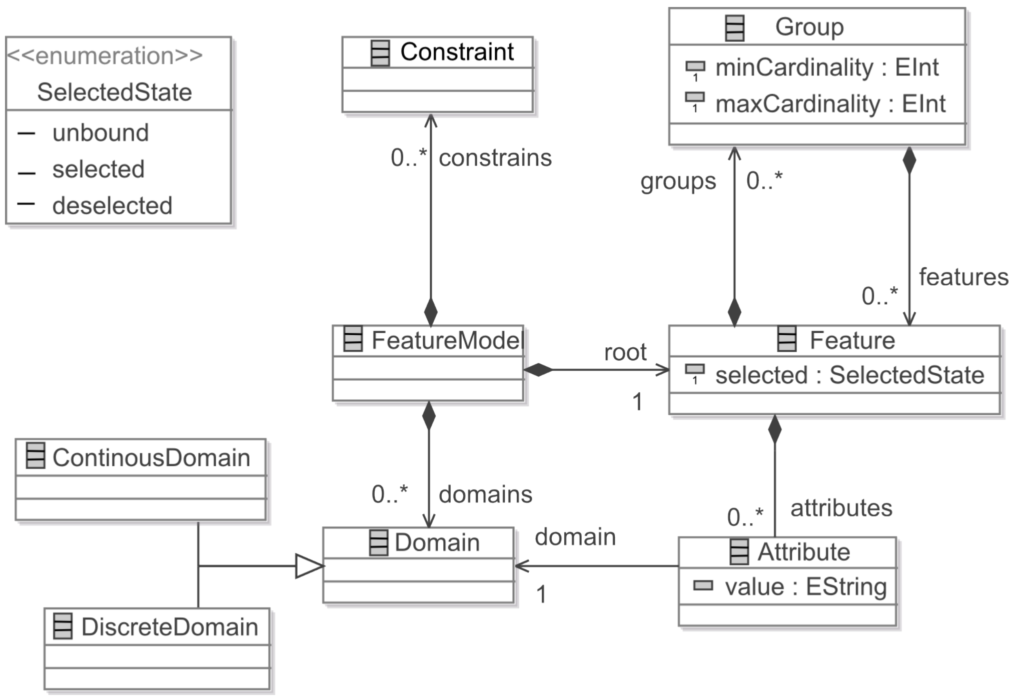


Figure 5.2: Abstract Ecore metamodel of Extended Feature Model

In the EFM, the feature diagrams organize sets of features in a tree-like hierarchical structure. As depicted in Figure 5.2, a *FeatureModel* has a *Feature* as the root of the tree structure. The tree structure is realized by using *Groups*. As a node of the tree, a *Feature* can include a set of *Groups* as its subtree. As a subtree, a *Group* can also contain several *Features* as its nodes. In addition, a *Group* has two attributes *minCardinality* and *maxCardinality* that are used to express the cardinality-based syntax (see Section 4.1.1).

Each *Feature* has a selected state which has three possible values: *unbound*, *selected* and *deselected*. Furthermore, a *Feature* may have several *Attributes*. Each *Attribute* has a value and refers to *Domain*, which is contained in *FeatureModel* and describes the possible specialization steps of the feature attribute. The domain can be a *DiscreteDomain* or a *ContinuousDomain*. Besides, *FeatureModel* also includes a collection of *Constrains* that can specify constrains on valid product configurations.

5.1.2 View Model Specification

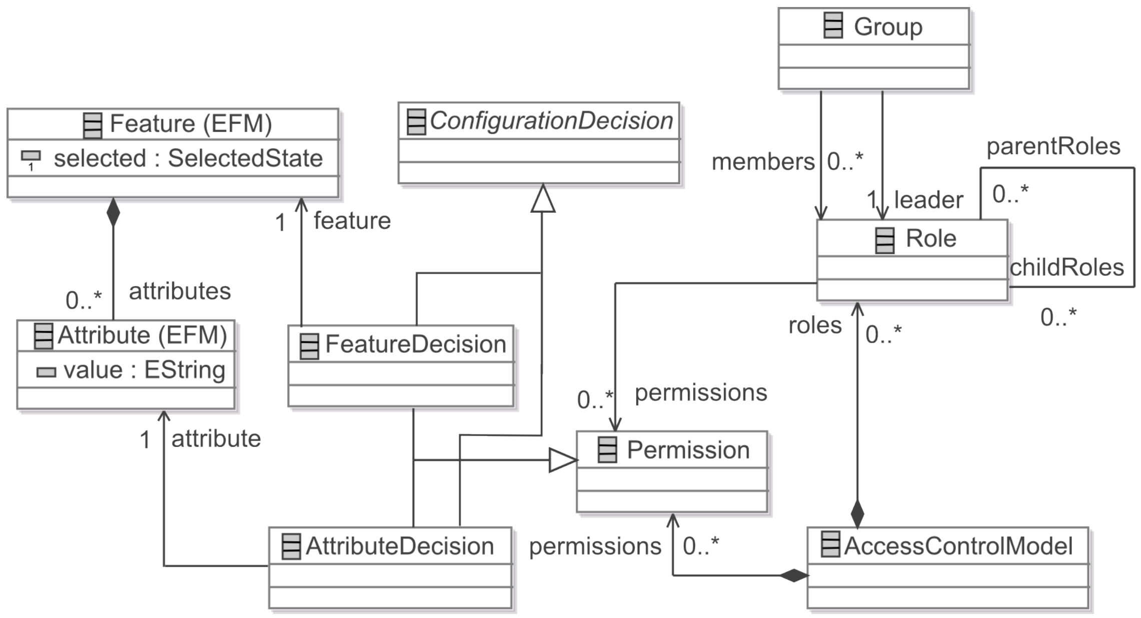


Figure 5.3: Abstract Ecore metamodel of View Model

The VM uses the base model of *Role-based Access Control*, namely $RBAC_0$ (see Section 2.3)). In Figure 5.3, a *AccessControlModel* represents the VM and contains a set of *Roles*, *Permissions* and *Organization*. The *Role* can be used to express the a *stakeholder* in our VM, e.g, a platform provider, an application provider, a tenant or a user. It can refer to a collection of *Permissions* that determine the role's possible specialization steps on an EFM. A *Role* has two types of *ConfigurationDecision*: *FeatureDecision* and *AttributeDecision*, which refers to a *Feature* and an *Attribute* in an EFM respectively. A *FeatureDecision* is used to select or deselect a feature while an *AttributeDecision* is used to set an attribute value. Both kinds of decisions are also considered as *Permissions*, which determine the possible decisions of a role.

A *Role* can have multiple *parent roles* and *child roles*, which realize the *Inheritance* relationship in the VM. The child roles inherits all permissions from their parent roles, and thus, multiple inheritance of a *Role* become reality. In addition, the *Group* represents the membership relation among stakeholders. Each *Group* contains a *leader* and several *members*.

5.1.3 Configuration Workflow Model Specification

JWT project provides design time, development time and runtime workflow tools to develop, deploy and test workflow. As a tool of JWT project, JWT Workflow Editor (WE) is used in our tooling as a basic workflow editor. WE is a visual tool for creating, managing and reviewing process definitions. It is based on EMF and enables users quickly to create workflow process definitions, check and store them for further use. By extending WE, we specify our CWM and develop the configuration management tooling, which consists of EFM editor, VM editor and CW editor. In this section, we introduce the abstract meta model of JWT WE that is relevant to our development, and then display our CWM that extends the JWT WE meta model.

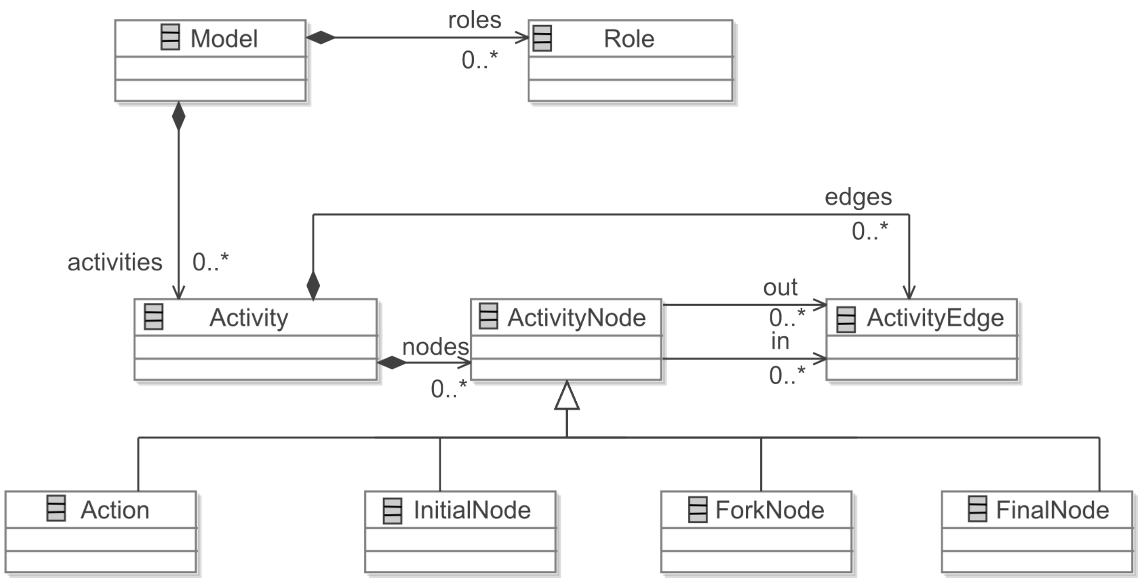


Figure 5.4: Abstract Ecore metamodel of JWT Workflow Editor

Figure 5.4 shows the abstract meta model of JWT WE. The JWT WE meta model contains a *Model*, which consists of a set of *Activities* and *Roles*. All processes modeled with Eclipse JWT are *Activities*. An *Activity* includes all elements in a graphical model. Examples for those elements are *ActivityNodes* and *ActivityEdges*. An *ActivityNode* can be, for example, an *Action*, an *InitialNode*, a *ForkNode* or a *FinalNode*. Each *ActivityEdge* connects two *ActivityNodes* as its source node and target node. *Roles* are defined not only for one process model, but also for all processes. Each *Action* can be performed either automatically or by a specific *Role*. For details on the JWT metamodel interested readers can refer to [BLR08].

We use the JWT WE as a basic activity diagram editor. Based on the basic activity diagram editor, we extend the above meta model to specify our CWM. In the following part, we will depict our extension to the JWT WE meta model. Each extended class is displayed in a figure and colored with gray.

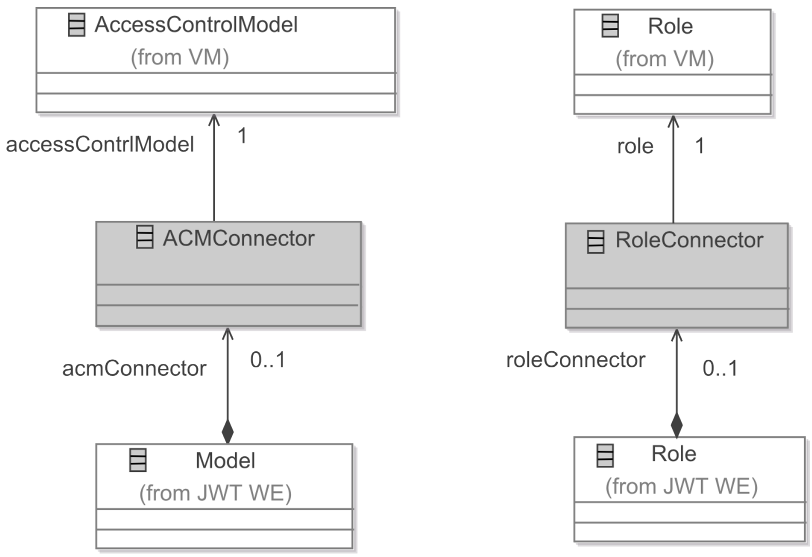


Figure 5.5: Connection between JWT Workflow Editor model and View Model

In order to connect JWT WE model with VM, an *ACMConnector* and a *RoleConnector* are built as depicted in Figure 5.5. The *Model* in JWT WE includes an *ACMConnector* which refers to an *AccessControlModel* in VM. This extension enables the JWT WE to get access to the *AccessControlModel*. *RoleConnector* is used to make a connection with the two *Roles*, which are separately included in JWT WE and VM. With the help of *RoleConnector*, the *Roles* in JWT WE are able to know their inheritance relation, membership relation as well as the views on the related EFM.

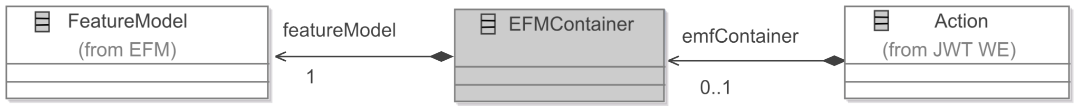


Figure 5.6: Connection between JWT Workflow Editor model and Extended Feature Model

When a workflow executes, a feature model which depicts the configuration space, is imported into the workflow. Once one stakeholder completes its specialization process, the specialized feature model as a partial configuration is stored locally and one copy is propagated to the succeeding stakeholder. In order to realize the storage functionality, an *EFMContainer* is constructed as shown in Figure 5.6. It links the JWT WE with EFM and allows the *Action* in JWT WE to store a specialized feature model.

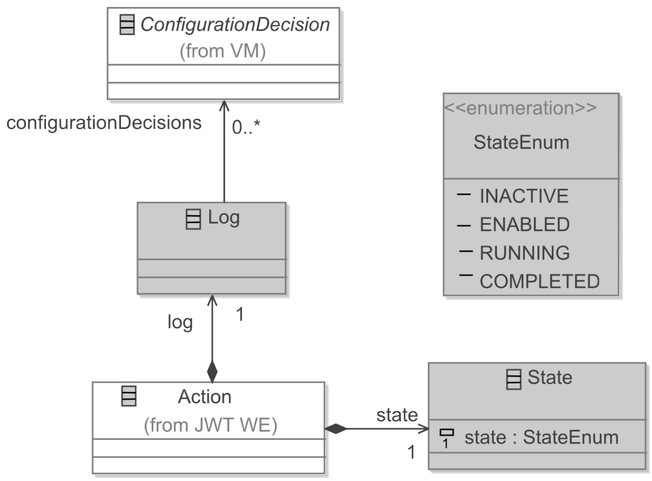


Figure 5.7: Extension of *Action* in JWT Workflow Editor model

When a stakeholder receives a partial configuration from its preceding stakeholder, it can start performing specialization steps. After the stakeholder completes the specialization process, its configuration decisions should be recorded. This record is important if a reconfiguration is requested in the future. This functionality is implemented by *Log* which is displayed in Figure 5.7. Each *Action* has a *Log* which includes a set of *ConfigurationDecisions*. These configuration decisions are the specialization steps, which are performed by the stakeholder of the *Action*.

Besides, we also create a *State* to realize the life cycle of an *Action*. There are four states in total: *inacitve*, *enabled*, *running* and *completed*. Each *Action* contains a *State*. In particular, when a new stakeholder arrives, and a new action is added into the workflow and its state must be initialized. The initial state of the action depends on the state of its preceding action. In addition, an event-based trigger mechanism is implemented which enables the workflow to execute according to the *control flow pattern* (see Section 4.1.3). If a stakeholder completes its operation, its related *Action* changes the state from *running* to *completed*, and notifies its succeeding actions of its new state. After receiving the notification, the succeeding actions change their states from *inactive* to *enabled*, and the referred stakeholders are able to perform specialization steps.

Figure 5.8 shows the complete meta models as well as their connections in the configuration management tooling. We extend JWT WE meta model to construct CWM. With the help of the extension, CWM is capable of utilizing EFM and VM to specify application variability and stakeholder configuration decisions. By using CWM, we can specify a configuration workflow. The workflow can execute according to the control flow pattern. When the workflow executes, the involved stakeholders are able to perform the configuration process. During the process, partial configurations and stakeholders' configuration decisions can be stored locally. Furthermore, the partial configurations are propagated from preceding stakeholders to succeeding stakeholders, and finally, complete configurations are derived. In next section, we will introduce our implementation of graph transformation.

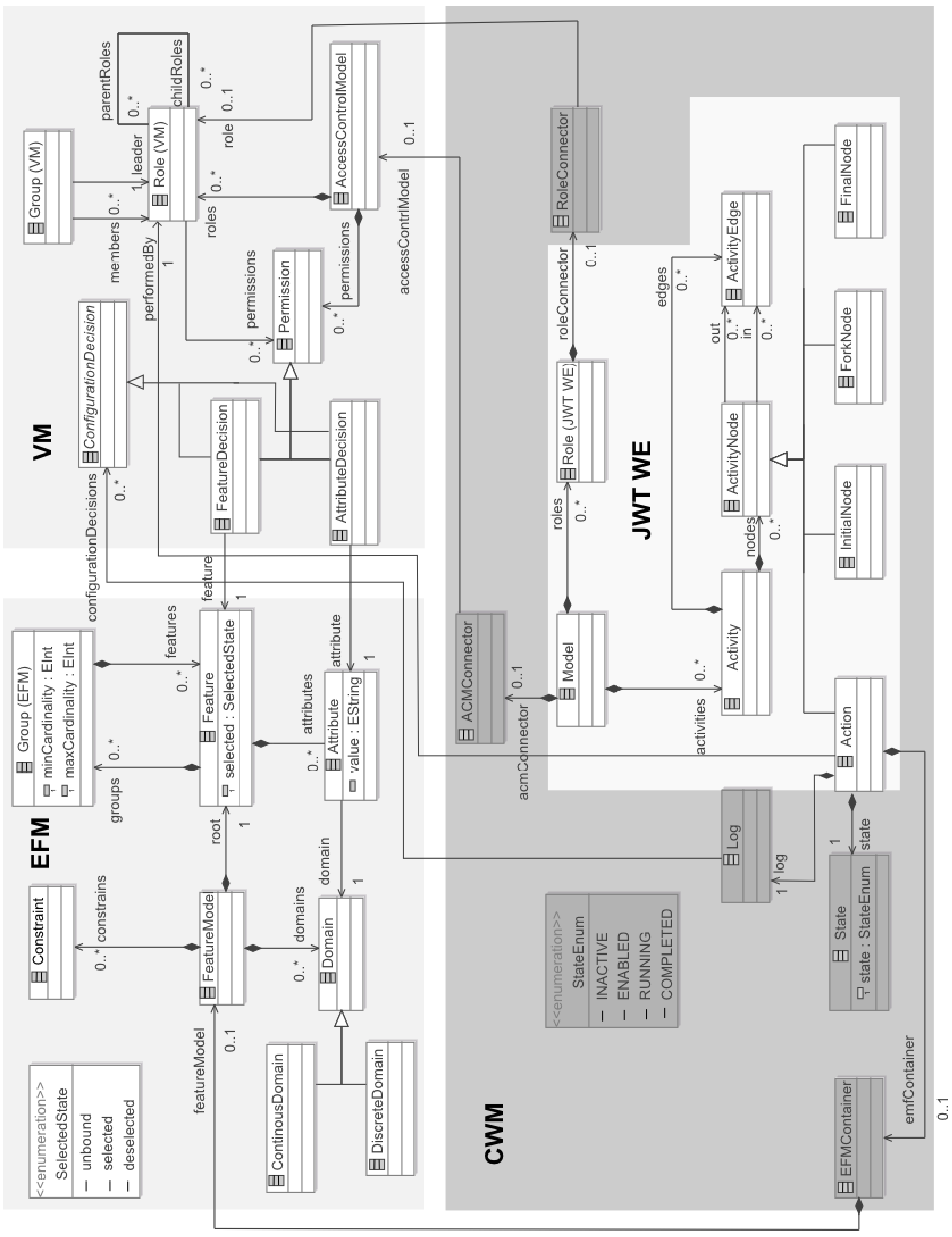


Figure 5.8: Utilized meta models in the configuration management tooling

5.2 Graph Transformation Rules

In order to integrate dynamic stakeholders into the configuration workflow, a workflow adaptation mechanism is needed for dynamic changes. In our concepts, we utilize the graph transformation to realize the workflow adaptation. In this section, we introduce the implementation of the graph transformation and its usage in our tooling. Based on the JWT WE meta model, we use Java as the programing language to realize all the following methods and algorithms.

In Section 4.2 we introduced the concepts of graph transformation. The idea of graph transformation is to apply a rule to a graph and create a new graph from the original one. Therefore, we focus on the specification and implementation of the rules.

In our concept in Chapter 4, we defined four rules for the YMS examples. Each rule is specifically defined for a certain stakeholder type. When a stakeholder arrives, the tooling will check the defined rules one by one. If rules are found valid, then the associated change primitives will be performed to the workflow.

A rule consists of *left-hand side*, *right-hand side* and *application conditions*. Additionally, for each rule a set of change primitives are defined, which are used for directly changing the elements in the workflow. In order to perform a rule, the left-hand side of the rule must occur in the workflow. Moreover, the application conditions must be fulfilled. The right-hand side of a rule is the result that the defined change primitives are applied to the left-hand side. Therefore, we only need to specify the left-hand side, application conditions and the associated change primitives, which are displayed as follows.

Since our workflow has a tree-like structure, the left-hand side can be considered as a subtree that contains a collection of nodes, which are connected with several edges. For example, the left-hand side of rule (c) contains two actions and a fork node. The three nodes are connected with two edges. In order to search the subtree in a workflow, we first need a specification of the subtree. In our solution, we use the root node as the specification to express the subtree. Listing 5.1 depicts this specification.

Listing 5.1: Specification of the left-hand side of rule (c)

```java
// initial the nodes and edges
Action action = processFactory.createAction();
ForkNode forkNode = processFactory.createForkNode();
Action idleAction = processFactory.createAction();
ActivityEdge edge1 = processFactory.
```

```
    createActivityEdge();
ActivityEdge edge2 = processFactory.
    createActivityEdge();

// build the subtree structure
edge1.setSource(action);
edge1.setTarget(forNode);
edge2.setSource(forNode);
edge2.setTarget(idleAction);
```

In Listing 5.1, the node *action* has an output edge, which connects the node *action* with the node *forkNode*. Similar to *edge1*, *edge2* connects the node *forkNode* and *idleAction*. Therefore, the node *action* is the root node and can express the structure of the subtree. To search the subtree, we can check whether nodes contained in the workflow have the same structure as the subtree. The searching method is depicted in Listing 5.2.

Listing 5.2: Method for searching left-hand side of rule (a)

```
searchLeftside_RuleC(Activity activity){
  for(ActivityNode node : activity.getNodes){
    if(hasStructure_RuleC(node)){
      leftSides.add(node);
    }
  }
}

hasStructure_RuleC(node){
  ActivityNode nextNode = JWTUtil.getNextNode(node);
  if(nextNode instanceof ForkNode){
    ForkNode forkNode = nextNode.getOutEdge().
        getTarget();
    if(JWTUtil.getNextNodes(forkNode).contains(
        idleAction)){
      return true;
    }
  }
  return false;
}
```

As a result, the *leftSide* contains all the occurrence of the left-hand side of rule (c). Similar to rule (c), we can also define the searching algorithms for the other rules. After we find the occurrence of left-hand side of a rule, we need to check the application conditions.

There are two types of application conditions. *Inherit from* is used to check the stakeholder type while *belongs to* is used to check the stakeholder membership. They respectively refer to the *role hierarchy* and *group* in VM. When a new stakeholder arrives and requests services, it must show its stakeholder type. Then the VM will create a role which inherits from User. If there are multiple services providers, the stakeholder has to show its requested stakeholder. For example, in Figure 4.12 a stakeholder wants to request services from *Tenant 2*. It should give its stakeholder type *User* and the requested stakeholder *Tenant 2*. The VM will create a role with the name *User 2* which inherits from User. Based on the given stakeholder type and requested stakeholder, the tooling will check whether the conditions of rules are fulfilled. Listing 5.3 describes an abstract method for checking conditions of rule (c).

Listing 5.3: Check application conditions

```
checkConditions_ruleC(leftSides, arrivingStk){
  for(ActivityNode root : leftSides){
    if(inheritFrom(root.getRole(), VM.getRole("
       ApplicationProvider")){
      if(inheritFrom(arrivingStk, VM.getRole("Tenant
         ")){
        return true;
      }
    }
  }
  return false;
}
```

After a rule is found valid, the related change primitives will be performed. The found occurrences of rules' left-hand side provide the exact positions where the change primitives should be performed. We implement the change primitives with the help of the API provider by JWT. Some examples about the change primitives are shown in Listing 5.4.

Listing 5.4: Examples of change primitives

```
public static Action addAction(Activity activity,
   String name) {
  Action action = processFactory.createAction();
  action.setName(name);
  activity.getNodes().add(action);
  return action;
}
```

```

public static ActivityEdge addEdge(Activity activity,
    ActivityNode source, ActivityNode target) {
  ActivityEdge actEdge = processFactory.
     createActivityEdge();
  actEdge.setSource(source);
  actEdge.setTarget(target);
  activity.getEdges().add(actEdge);
  return actEdge;
}
```

By utilizing the graph transformation rules, the tooling is able to change the workflow dynamically when a stakeholder arrives. In next section we will introduce the realization of the mapping between our EFM configuration and YMS configuration.

5.3 Mapping Realization

In Section 4.3 we have introduced the variability model utilized in the YMS. In an EFM configuration all variability is bound. Thus, an EFM configuration is composed of the features with a certain state (i.e. *selected* or *deselected*). In the variability model of YMS, the configuration variation points are represented by $VariabilityElements$ while the configuration decisions are expressed by *ResolutionElements*. Therefore, we match the Feature and Attribute in EFM with the $VariabilityElement$ in the variability model of YMS. In addition, we match the feature state (the attribute *selected* of *Feature*) and the feature attribute value (the attribute *value* of *Attribute*) with the *ResolutionElement*.(see Figure 5.8)

We first create an EFM according to the feature model of YMS depicted in Figure 4.2. In the cerated EFM, each feature or feature attribute is coordinated with a $VariabilityElement$ in the YMS variability model. After the configuration process, a fully specialized EMF is derived and all variability is bound. The feature state is either *selected* or *deselected.* Moreover, in the feature model of YMS, there is only one feature attribute *scheduling type.* The value of the feature attribute is a String type value.

In our mapping methods, we create a *resolutionElement* for each feature contained in the EFM. In the resolutionElement, we set its value as the feature state, and its resolved variability element as the feature id. Finally, the resolutionElement is saved in the variability resolution model. An example for mapping a configuration decision onto resolution element is depicted in Listing

5.5.

Listing 5.5: Algorithm for mapping EMF and VariabilityResolution

```
Feature feature = EFMUtil.findFeature(featureName,
   featureModel);
for(Feature feature : featureModel){

  ResolutionElement resolutionElement = new
     ResolutionElement();

  // to coordinate with the YMS variability element
  String resolves = TransformParser.analyze(feature.
     getId());

  // set variability element
  // that is resolved by the resolution element
  resolutionElement.setResolves(resolves);

  // set the value for the resolved variability
     element
  resolutionElement.setValue(feature.getSelected());

  // set the binding time
  resolutionElement.setBindingTime(BindingTime.
     RUN_TIME);

  variabilityResolution.getResolutionElements().add(
     resolutionElement);
}
```

In the YMS, the *VariabilityModel* and *VariabilityResolution* are respectively saved in the a **.var* file and a **.res* file. When the YMS instance is deployed, the *VariabilityResolution* in the **.res* file is loaded. At run time, when users perform the specialization steps, all their configuration decisions are first saved in **.res* file, and then loaded by the YMS instance. Thus, after we finish the mapping between EFM configuration and YMS configuration, we save the *VariabilityResolution* in the **.res* file and transport the file to the YMS instance.

5.4 Configuration Management Tooling

As introduced above, the configuration management tooling consists of a configuration workflow editor and two editors for EFM and VM. The editors for EFM and VM are developed based on EMF and EMFText. Thus, they allow developers to specify EFM and VM by using domain specific languages. Examples for the EFM and VM modeling by using the editors are shown in Listing 5.6 and Listing 5.7.

Listing 5.6: Example for EFM using EFM editor

```

feature model "Test"
domain <d1> [v1, v2, v3, v4]
domain <d2> [10..20, 30..50]

feature "F_Root" <fr>
  group <g_alt> (0..1) {
  feature "F2" <f2>
  feature "F3" <f3>
}
```

Listing 5.7: Example for VM using VM editor

```
access control on <simpleFM.eft>

permissions {
  select feature f2
  deselect feature f2
  select feature f3
  deselect feature f3
}

role <roleType1> {
  select feature f2
  deselect feature f2
}
role <roleType2> {
  select feature f3
  deselect feature f3
}

role <role1> extends roleType1
role <role2> extends roleType2
```

```

group "group1" <g1> role1 {
   role2
}
```

The Configuration Workflow editor (CW editor) aims to specify the configuration workflow. After the editor imports the EFM and VM models via an *import models* dialog as depicted in reffigimportingmodels, we can use the editor to design our workflow. After we gives the input parameters via a dialog (e.g. the stakeholder type name), the CW editor will create a role for the stakeholder. Then it searches the predefined graph transformation rules. If a rule is found valid, the CW editor will automatically change the original workflow according to the rule. The UI of the CW editor is depicted in Figure 5.10.

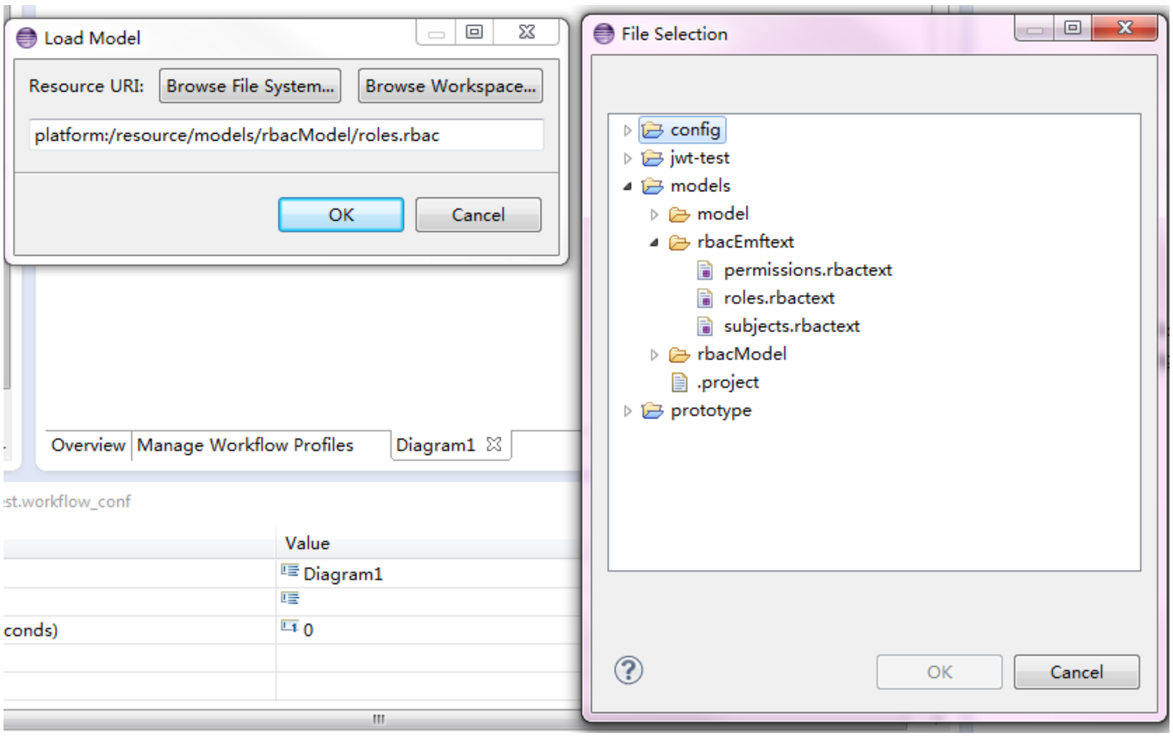

Resource URI: platform:/resource/models/rbacModel/roles.rbac

Figure 5.9: Models importing

In the depicted workflow, there are two stakeholders involved. Both of them refer to an action. The action state is shown in the action name. Before the workflow executes, except the *idle action* all actions' state is *inactive*. The *Idle action* keeps the state *enabled* until we double click the *idle action*. When we execute the workflow (clicking *start* button), the root action (action of *Platform Provider 1*) changes its state from *inactive* to *enabled*.

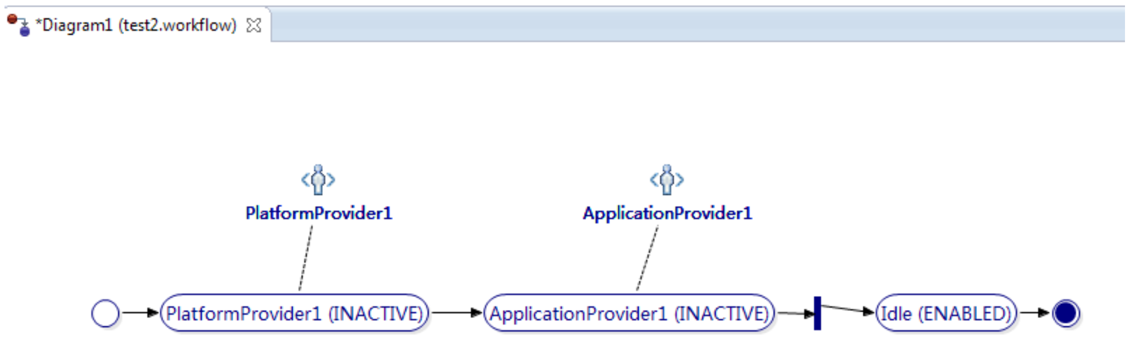
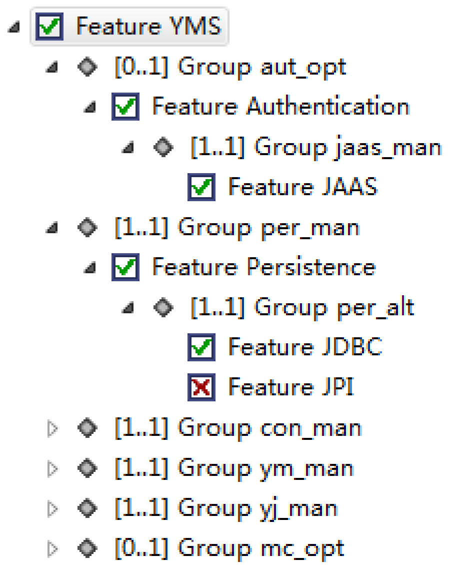


*Diagram1 (test2.workflow)
PlatformProvider1
ApplicationProvider1
PlatformProvider1 (INACTIVE)
ApplicationProvider1 (INACTIVE)
Idle (ENABLED)

Figure 5.10: A workflow example in the configuration workflow editor



- ☑ Feature YMS
  - [0..1] Group aut_opt
    - ☑ Feature Authentication
      - [1..1] Group jaas_man
        - ☑ Feature JAAS
  - [1..1] Group per_man
    - ☑ Feature Persistence
      - [1..1] Group per_alt
        - ☑ Feature JDBC
        - ☒ Feature JPI
  - [1..1] Group con_man
  - [1..1] Group ym_man
  - [1..1] Group yj_man
  - [0..1] Group mc_opt

Figure 5.11: Configuration viewer

We can perform the specialization steps via viewer with tree-like structure, which is shown in Figure 5.11. The viewer can identify the stakeholder and get access to VM to obtain the permissions of the stakeholder. Through the viewer, we are allowed to select and deselect features, or set the feature attribute values according to our permissions which specified in VM. After the specialization process, the related action changes state from *running* to *completed*, and then the succeeding actions are enabled. The partial configuration is stored in the action and a copy is propagated to the succeeding actions. At the end of a configuration flow, a configuration is derived. This configuration is represented by a specialized EFM, which is saved in a **.feature* file. With the **.feature* file as input parameter, we call the mapping method, which is depicted in Section 5.3. After the mapping process, the **.feature* file is transformed to **.res* file. This file will be integrated into the YMS instance which is deployed in remote server.

The configuration workflow tooling implements our proposed concepts. In order to evaluate concepts, we will utilize the tooling to perform a configuration process regarding a cloud-based application in the next section.

5.5 Evaluation

In this section, we will evaluate the usability of the proposed concepts. By using the developed tooling, we apply a configuration process to a cloud-based application. As a case study, we use the YMS prototype running on SAP Netweaver Cloud and apply a configuration process to derive a YMS configuration.

As introduced in Chapter 3, the YMS is used to manage the movement of trucks and trailers in the yard of a manufacturing facility. The current version supports several features, such as dock door scheduling, mobile communication service and location services. The YMS defines multiple variation points that can be tailored to meet customer needs. In order to apply configuration operations on the variation points, we first utilize the EFM editor to specify the feature model regarding the YMS. The feature model is already given in the Figure 4.2. By using the EFM editor, we specify the feature model as depicted in Listing 5.8.

Listing 5.8: Specification of EFM regarding YMS using EFM editor

```
feature model "YMS"
domain <scheduleType> [next, fitting]
//domain <d2> [10..20, 30..50]

feature "YMS" <yms>
  group <aut_opt>(0..1) {
    selected feature "Authentication" <authentication
      >
      group <jaas_man> (1..1){
        feature "JAAS" <jaas>
      }
  }
  group <per_man>(1..1){
    feature "Persistence" <persistence>
```

```
        group <per_alt >(1..1){
          feature "JDBC" <jdbc>
          feature "JPI" <jpi>
      }
    }
... ...

constraint <lsconstraint> ls -> mc
```

In addition, four stakeholder types are involved in the configuration process. These stakeholder types as well as their possible specialization steps are defined in VM. We use the VM editor to define the above information that is depicted in Listing 5.9.

Listing 5.9: Specification of VM regarding YMS using EFM editor

```
access control on <YMS.eft>

permissions {
  select yms,
  select authentication ,
  deselect authentication ,
  ... ...
}
role "platformProvider" <platformProvider> {
  "select yms",
  "select authentication",
  ... ...
}
role "applicationProvider" <applicationProvider> {
  ... ...
}
role "tenant" <tenant> {
  ... ...
}
role "user" <user> {
... ...
}
```

After we finish the specification of EFM and VM, we import the models into the CW editor. When we add a new stakeholder into the workflow, the editor will change the workflow. Figure 5.12 shows a complete workflow.

During the configuration process, if an action has the state *enabled*, we can

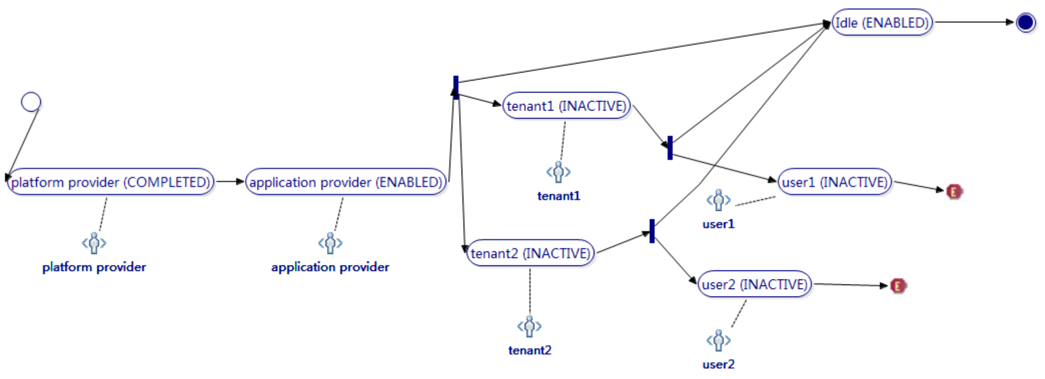


Figure 5.12: A configuration workflow in the configuration workflow editor

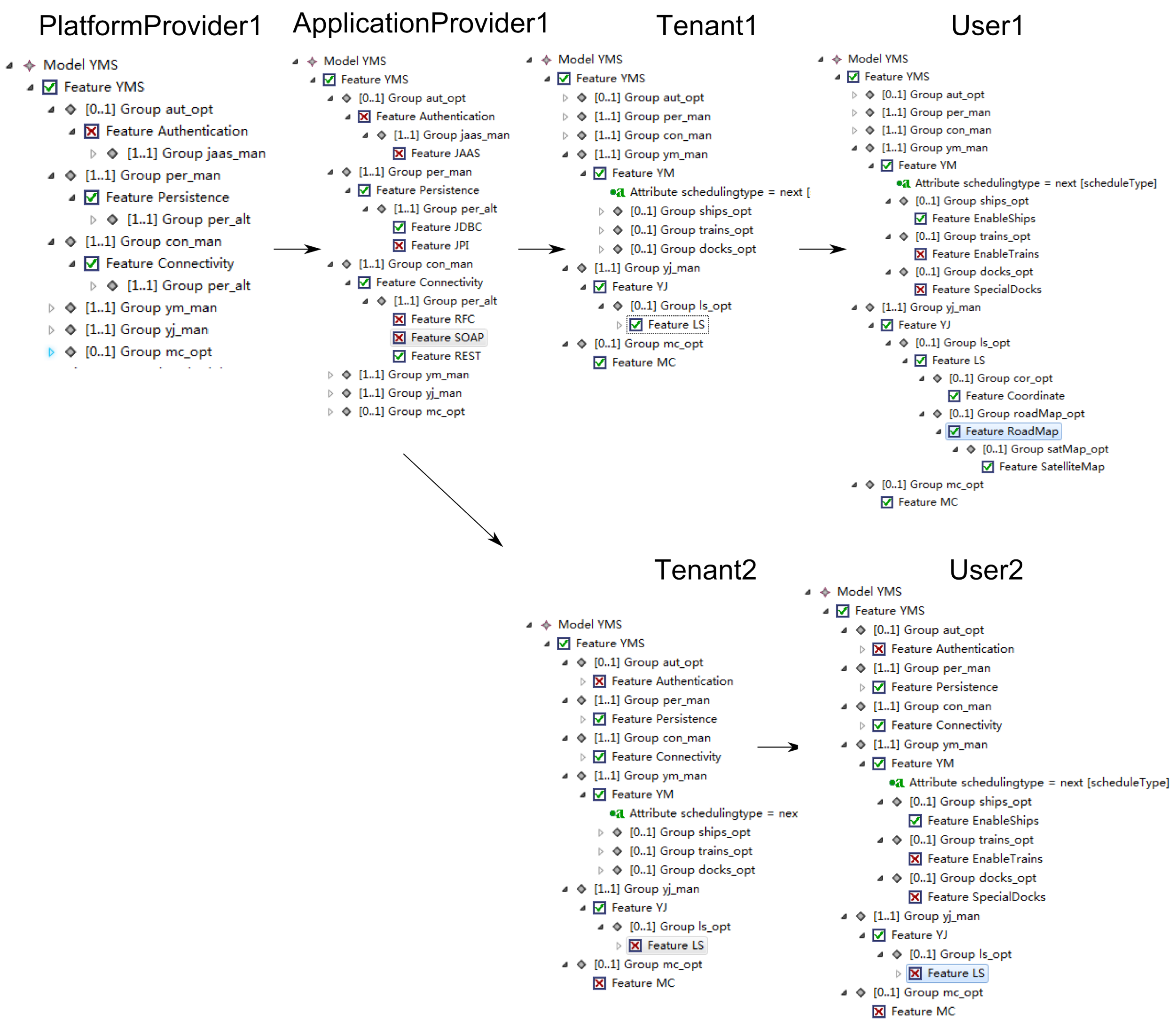
perform the specialization steps to the action via the configuration viewer that is already shown in Figure 5.11. The tooling allows multiple stakeholders to perform the specialization steps simultaneously. For example, in Figure 5.13, *Tenant1* and *Tenant2* obtain the copies of the partial configuration from *ApplicationProvider1*, and perform the specialization steps concurrently. When they complete the operations, they propagate their partial configurations to their users.

After a staged configuration process, a complete configuration is derived at the end of a flow and will be transformed into YMS configuration. Listing 5.10 displays the part of transformed features in the YMS configuration.

Listing 5.10: YMS configuration

```
<ns3:resolutionElement resolves="SchedulingType"
    value="next" bindingTime="runTime"/>
 <ns3:resolutionElement resolves="useMobile" value
    ="true" bindingTime="runTime"/>
<ns3:resolutionElement resolves="shipsEnabled"
    value="true" bindingTime="runTime"/>
<ns3:resolutionElement resolves="useGps" value="
    true" bindingTime="runTime"/>
<ns3:resolutionElement resolves="showGpsText" value
    ="true" bindingTime="runTime"/>
<ns3:resolutionElement resolves="showGpsMap" value
    ="true" bindingTime="runTime"/>
<ns3:resolutionElement resolves="gpsMapIsSatellite"
     value="true" bindingTime="runTime"/>
```

After the configuration is transported to the YMS instance, the requested services will be provided. For example, the above configuration enables the



PlatformProvider1

```
Model YMS
  ✓ Feature YMS
    [0..1] Group aut_opt
      ✗ Feature Authentication
        [1..1] Group jaas_man
    [1..1] Group per_man
      ✓ Feature Persistence
        [1..1] Group per_alt
    [1..1] Group con_man
      ✓ Feature Connectivity
        [1..1] Group per_alt
    [1..1] Group ym_man
    [1..1] Group yj_man
    [0..1] Group mc_opt
```

ApplicationProvider1

```
Model YMS
  ✓ Feature YMS
    [0..1] Group aut_opt
      ✗ Feature Authentication
        [1..1] Group jaas_man
          ✗ Feature JAAS
    [1..1] Group per_man
      ✓ Feature Persistence
        [1..1] Group per_alt
          ✓ Feature JDBC
          ✗ Feature JPI
    [1..1] Group con_man
      ✓ Feature Connectivity
        [1..1] Group per_alt
          ✗ Feature RFC
          ✗ Feature SOAP
          ✓ Feature REST
    [1..1] Group ym_man
    [1..1] Group yj_man
    [0..1] Group mc_opt
```

Tenant1

```
Model YMS
  ✓ Feature YMS
    [0..1] Group aut_opt
    [1..1] Group per_man
    [1..1] Group con_man
    [1..1] Group ym_man
      ✓ Feature YM
        Attribute schedulingtype = next [
        [0..1] Group ships_opt
        [0..1] Group trains_opt
        [0..1] Group docks_opt
    [1..1] Group yj_man
      ✓ Feature YJ
        [0..1] Group ls_opt
          ✓ Feature LS
    [0..1] Group mc_opt
      ✓ Feature MC
```

User1

```
Model YMS
  ✓ Feature YMS
    [0..1] Group aut_opt
    [1..1] Group per_man
    [1..1] Group con_man
    [1..1] Group ym_man
      ✓ Feature YM
        Attribute schedulingtype = next [scheduleType]
        [0..1] Group ships_opt
          ✓ Feature EnableShips
        [0..1] Group trains_opt
          ✗ Feature EnableTrains
        [0..1] Group docks_opt
          ✗ Feature SpecialDocks
    [1..1] Group yj_man
      ✓ Feature YJ
        [0..1] Group ls_opt
          ✓ Feature LS
            [0..1] Group cor_opt
              ✓ Feature Coordinate
            [0..1] Group roadMap_opt
              ✓ Feature RoadMap
                [0..1] Group satMap_opt
                  ✓ Feature SatelliteMap
    [0..1] Group mc_opt
      ✓ Feature MC
```

Tenant2

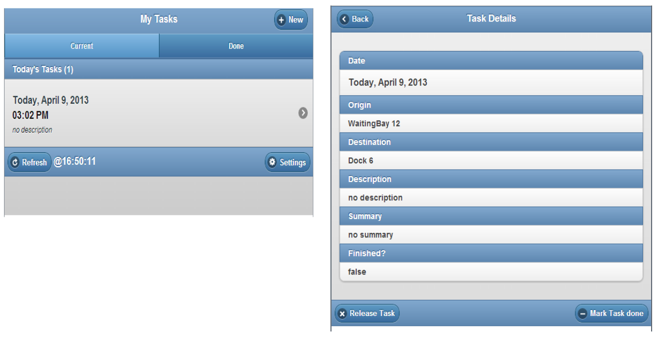
```
Model YMS
  ✓ Feature YMS
    [0..1] Group aut_opt
      ✗ Feature Authentication
    [1..1] Group per_man
      ✓ Feature Persistence
    [1..1] Group con_man
      ✓ Feature Connectivity
    [1..1] Group ym_man
      ✓ Feature YM
        Attribute schedulingtype = nex
        [0..1] Group ships_opt
        [0..1] Group trains_opt
        [0..1] Group docks_opt
    [1..1] Group yj_man
      ✓ Feature YJ
        [0..1] Group ls_opt
          ✗ Feature LS
    [0..1] Group mc_opt
      ✗ Feature MC
```

User2

```
Model YMS
  ✓ Feature YMS
    [0..1] Group aut_opt
      ✗ Feature Authentication
    [1..1] Group per_man
      ✓ Feature Persistence
    [1..1] Group con_man
      ✓ Feature Connectivity
    [1..1] Group ym_man
      ✓ Feature YM
        Attribute schedulingtype = next [scheduleType]
        [0..1] Group ships_opt
          ✓ Feature EnableShips
        [0..1] Group trains_opt
          ✗ Feature EnableTrains
        [0..1] Group docks_opt
          ✗ Feature SpecialDocks
    [1..1] Group yj_man
      ✓ Feature YJ
        [0..1] Group ls_opt
          ✗ Feature LS
    [0..1] Group mc_opt
      ✗ Feature MC
```

Figure 5.13: Multiple stakeholders concurrently perform specialization steps

drivers and jockeys to use mobile devices. In addition, the yard manager can user the satellite map services. The two services are respectively displayed in Figure 5.14 and Figure 5.15.



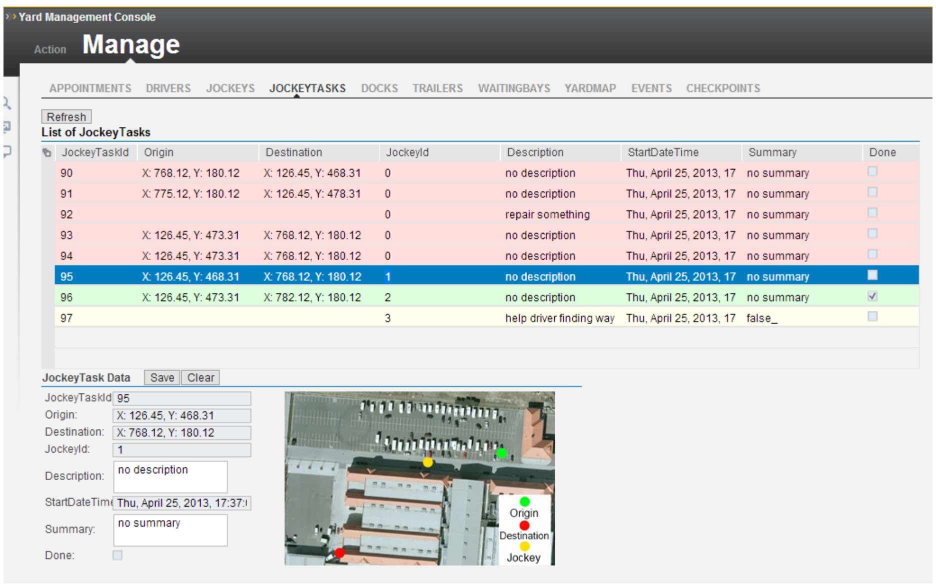
Today's Tasks (1)

Today, April 9, 2013
03:02 PM
no description

@16:50:11

Date	Today, April 9, 2013
Origin	WaitingBay 12
Destination	Dock 6
Description	no description
Summary	no summary
Finished?	false

Figure 5.14: Mobile communication service



List of JockeyTasks

JockeyTaskId	Origin	Destination	JockeyId	Description	StartDateTime	Summary	Done
90	X: 768.12, Y: 180.12	X: 126.45, Y: 468.31	0	no description	Thu, April 25, 2013, 17	no summary	☐
91	X: 775.12, Y: 180.12	X: 126.45, Y: 478.31	0	no description	Thu, April 25, 2013, 17	no summary	☐
92			0	repair something	Thu, April 25, 2013, 17	no summary	☐
93	X: 126.45, Y: 473.31	X: 768.12, Y: 180.12	0	no description	Thu, April 25, 2013, 17	no summary	☐
94	X: 126.45, Y: 473.31	X: 768.12, Y: 180.12	0	no description	Thu, April 25, 2013, 17	no summary	☐
95	X: 126.45, Y: 468.31	X: 768.12, Y: 180.12	1	no description	Thu, April 25, 2013, 17	no summary	☐
96	X: 126.45, Y: 473.31	X: 782.12, Y: 180.12	2	no description	Thu, April 25, 2013, 17	no summary	✓
97			3	help driver finding way	Thu, April 25, 2013, 17	false_	☐

JockeyTask Data

JockeyTaskId	95
Origin:	X: 126.45, Y: 468.31
Destination:	X: 768.12, Y: 180.12
JockeyId:	1
Description:	no description
StartDateTime	Thu, April 25, 2013, 17:37:
Summary:	no summary
Done:	☐

Figure 5.15: Location service with satellite map

Based on the YMS example, we use the configuration management tooling to perform the staged configuration process. The multiple stakeholders apply the specialization steps to the imported EFM configuration space. At the end of each flow, an EMF configuration is derived and transformed into YMS configuration. Finally, the YMS instance will provide the required services to the users according to the integrated configurations. By utilizing the developed tooling, we have accomplished the the configuration management for a multi-tenant cloud-based application, and up to now the usability of the proposed concepts is finished with demonstration.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we focus on managing the variability of the multi-tenant cloud-based applications. As a case study, we analyzed the YMS and identified the requirements of the configuration management for cloud-based applications. Based on the previous work [SMM$^+$12], we presented concepts of configuration management that is able to manage and create tenant configurations for cloud-based applications.

In the concepts, the Extended Feature Model (EFM) is used to model the commonality and variability of a cloud-based application. By using EFM, a configuration space regarding the application is constructed. In order to limit stakeholders' possible specialization steps in the configuration space, we defined the View Model (VM) that provides a view concept. In order to model a configuration workflow, we defined the Configuration Workflow Model (CWM). The CWM aims to construct a configuration workflow that enables multiple stakeholders to apply configuration process concurrently. After the related stakeholders finish the staged configuration, a complete configuration per user is created and all variability is bound. EFM, VM and CWM compose our configuration management model structure.

Additionally, in order to allow the configuration management to integrate dynamic stakeholders, we also presented the workflow adaptation concept that utilizes the graph transformation rules to perform the dynamic changes of a workflow. With the help of this concept, our configuration management is able to manage the configurations of predefined or new arriving stakeholders.

In order to use the derived EFM configuration, the cloud-based application must be able to recognize the EFM. Therefore, we provided a mapping solution based on the YMS. By mapping EFM configuration onto the YMS configuration, we integrate the configuration into the YMS instance, which will provide the users with demanded services.

To implement the concepts, we extend the JWT Workflow Editor (JWT WE) to build the CWM and connect the CWM with EFM and VM. By utilizing the developed configuration management tooling, we can visually specify a configuration workflow and execute the workflow. During the execution of the workflow, a staged configuration process is performed. Each involved stakeholder will apply specialization steps according to their permissions that are defined in VM. When a new stakeholder intents to join the workflow, the tooling will search the valid graph transformation rules and apply their associated change primitives. In addition, the tooling is able to create YMS configurations according to the derived EFM configurations at the end of the workflow. When YMS configurations are integrated into the YMS instance, the YMS instance will provide the demanded services for the related stakeholders. Therefore, the developed tooling proofed the usability of our proposed concept.

6.2 Future Work

In this section, some future work will be listed as follows:

1. **Reconfiguration**
   When a stakeholder‘s objective changes, a reconfiguration is probably required. For example, a tenant decides to rent different services, then its configuration needs to be reconfigured. Our proposed CWM can be used to specify a staged configuration process. In a staged configuration process, the down-stream configurations are affected by the up-stream configuration. Therefore, if a stakeholder changes its configuration, its subsequent stakeholders' configurations are affected by the reconfiguration, and probably need to be changed as well.

2. **Verification**
   In order to ensure the configuration process consistency, a process verification is required. The process of verification can be used to verify the soundness, completeness and termination of the overall configuration process. With the process of verification, users can also keep track of their configurations to complete error-correction and error-avoidance.

3. **Removal of Stakeholders**
   During the configuration process, some stakeholders finish their operations and will leave the workflow. Thus, the functionality of removing

stakeholder from the workflow is needed. Our proposed graph transformation rules can be used to fulfill the removal of workflow elements, such as an action, or a subworkflow. When a stakeholder is removed from the workflow, it may probably affect other stakeholders. For example, if an application provider leaves the workflow, its related tenants will also leave or refer to other providers. Therefore, the definition of the graph transformation rules about removing stakeholders should take the stakeholders' objectives into consideration.

4. **Design and Realization of Graph Transformation Rules**
   Based on the YMS example, we defined four graph transformation rules. According to different applications, various rules should be respectively defined. Therefore, a standard modeling tool can help developers to efficiently design the rules. Furthermore, in our concepts, we check the whole workflow to search the occurrences of the left-hand side in a rule. This algorithm is inefficient and leads to redundant searching work. For this reason, the methods about graph matching in pattern recognition can be considered as solutions of searching the rules' left-hand side [CFSV04].

The above future work is expected to focus on the extension possibilities and improvements in our feature-based configuration management of applications in the cloud.

Bibliography

[AEH$^+$99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer programming*, 34(1):1–54, 1999.

[AK03] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41, 2003.

[BBRC06] Don Batory, David Benavides, and Antonio Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Communications of the ACM*, 49(12):45–47, 2006.

[BD07] Danilo Beuche and Mark Dalgarno. Software product line engineering with feature models. *Overload Journal*, 78:5–8, 2007.

[BLR08] Bernhard Bauer, Florian Lautenbacher, and Stephan Roser. Agilpro - agile processes in the context of erp - agilpro metamodel description. `\T1\textquotelefthttp://wiki.eclipse.org/images/2/2f/AgilPro_MetamodelDescription.pdf`, 2008.

[BRCT05] David Benavides, Antonio Ruiz-Cortes, and Pablo Trinidad. Using constraint programming to reason on feature models. In *The Seventeenth International Conference on Software Engineering and Knowledge Engineering, SEKE*, volume 2005, pages 677–682, 2005.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Advanced Information Systems Engineering*, pages 491–503. Springer, 2005.

[BZP$^+$10] C-P Bezemer, Andy Zaidman, Bart Platzbeecker, Toine Hurkmans, and A t Hart. Enabling multi-tenancy: An industrial experience report. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–8. IEEE, 2010.

[CFSV04] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence*, 18(03):265–298, 2004.

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In *Software Product Lines*, pages 266–283. Springer, 2004.

[CHE05a] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[CHE05b] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[CK05] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints: A progress report. In *International Workshop on Software Factories*, 2005.

[Con11a] Indenica Consortium. Description of feasible case studies. `http://www.indenica.eu/fileadmin/INDENICA/user_upload/d51-casestud.pdf`, 2011. Indenica Project Deliverable D5.1.

[Con11b] Indenica Consortium. View-based design time and runtime architecture for tailoring vpss. `http://www.indenica.eu/fileadmin/INDENICA/user_upload/d31-viewbasedarch.pdf`, 2011. Indenica Project Deliverable D3.1.

[Con12] Indenica Consortium. Implementation of a family of service platforms and applications. `http://www.indenica.eu/fileadmin/INDENICA/user_upload/D531_ImplementationPlatforms.pdf`, 2012. Indenica Project Deliverable D5.3.1.

[FKC92] DF Ferrailio, DR Kuhn, and R Chandramouli. Role based access control. In *15th National Computer Security Conference*, 1992.

[Fow97] Martin Fowler. *UML Distilled: Applying the Standard Object Modelling Language.* Addison-Wesley, 1997.

[HDT95] MY Hu, SA Demurjian, and TC Ting. User-role based security in the adam object-oriented design and analyses environment. *Database Security VIII: Status and Prospects. North-Holland*, 1995.

[Hec06] Reiko Heckel. Graph transformation in a nutshell. *Electronic notes in theoretical computer science*, 148(1):187–198, 2006.

[HHS$^+$11] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Dirk Deridder, and Ebrahim Khalil Abbasi. Supporting multiple perspectives in feature-based configuration. *Software & Systems Modeling*, pages 1–23, 2011.

[HTHS07] Michael Hompel, Michael Ten Hompel, and Thorsten Schmidt. *Warehouse management: automation and organisation of warehouse and order picking systems.* Springer, 2007.

[KCH$^+$90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

[KU00] Czarnecki Krzysztof and Eisenecker Ulrich. Generative programming: Methods, tools, and applications, 2000.

[MCO07] Marcilio Mendonca, Donald Cowan, and Toacy Oliveira. A process-centric approach for coordinating product configuration decisions. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 283a–283a. IEEE, 2007.

[MG11] Peter Mell and Timothy Grance. The nist definition of cloud computing (draft). *NIST special publication*, 800:145, 2011.

[MH06] R. Miles and K. Hamilton. *Learning UML 2.0.* O'Reilly Media, Inc., 2006.

[MMLP09] Ralph Mietzner, Andreas Metzger, Frank Leymann, and Klaus Pohl. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 18–25. IEEE Computer Society, 2009.

[NO94] Matunda Nyanchama and Sylvia L Osborn. Access rights administration in role-based security systems. In *Proceedings of the IFIP WG11*, volume 3, pages 37–56. Citeseer, 1994.

[NWG+09] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*, pages 124–131. IEEE, 2009.

[OAWtH10] Chun Ouyang, Michael Adams, Moe Thandar Wynn, and Arthur HM ter Hofstede. Workflow management. In *Handbook on Business Process Management 1*, pages 387–418. Springer, 2010.

[OMG11] OMG OMG. Unified modeling language (omg uml), 2011.

[PBVDL05] Klaus Pohl, Gunter Bockle, and Frank Van Der Linden. *Software product line engineering*, volume 10. Springer, 2005.

[RA11] Stefan T Ruehl and Urs Andelfinger. Applying software product lines to create customizable software-as-a-service applications. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, page 16. ACM, 2011.

[RBSP02] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with uml multiplicities. In *6th Conference on Integrated Design & Process Technology (IDPT 2002), Pasadena, California, USA*, 2002.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The.* Pearson Higher Education, 2004.

[RS10] Marko Rosenmüller and Norbert Siegmund. Automating the configuration of multi software product lines. *Proceedings of VaMoS*, 10:123–130, 2010.

[RW12] Manfred Reichert and Barbara Weber. *Enabling flexibility in process-aware information systems.* Springer, 2012.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[SCG+12] Julia Schroeter, Sebastian Cech, Sebastian Götz, Claas Wilke, and Uwe Aßmann. Towards modeling a variable architecture for multi-tenant saas-applications. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, pages 111–120. ACM, 2012.

[Sch06] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[SMLF09] Borja Sotomayor, Rubén S Montero, Ignacio M Llorente, and Ian Foster. Virtual infrastructure management in private and hybrid clouds. *Internet Computing, IEEE*, 13(5):14–22, 2009.

[SMM$^+$12] Julia Schroeter, Peter Mucha, Marcel Muth, Kay Jugel, and Malte Lochau. Dynamic configuration management of cloud-based applications. In *Proceedings of the 16th International Software Product Line Conference-Volume 2*, pages 171–178. ACM, 2012.

[VBB11] William Voorsluys, James Broberg, and Rajkumar Buyya. Introduction to cloud computing. *Cloud Computing: Principles and Paradigms, Wiley Press, New York*, pages 3–41, 2011.

[VDAVH04] Wil Van Der Aalst and Kees Max Van Hee. *Workflow management: models, methods, and systems.* The MIT press, 2004.

[Wei09] Neal Weinberg. Cloud computing: Hot technology for 2009. `http://www.networkworld.com/supp/2009/outlook/hottech/010509-nine-hot-techs-cloud-computing.html`, 2009.

[Wes12] Mathias Weske. *Business process management.* Springer, 2012.

[WRRM08] Barbara Weber, Manfred Reichert, and Stefanie Rinderle-Ma. Change patterns and change support features–enhancing flexibility in process-aware information systems. *Data & knowledge engineering*, 66(3):438–466, 2008.

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, 30.04.2013