FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Model Driven Systems Configuration

Improving the Efficiency & Quality of Engineering Process Assembly based on Variability Modeling

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Kristof Meixner, BSc.
Matrikelnummer 09725208

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Stefan Biffl
Mitwirkung: Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Richard Mordinyi
Dipl.-Ing. Dr.techn. Dietmar Winkler

Wien, 19. Februar 2018

Kristof Meixner

Stefan Biffl

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Model Driven Systems Configuration

Improving the Efficiency & Quality of Engineering Process Assembly based on Variability Modeling

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Kristof Meixner, BSc.
Registration Number 09725208

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Stefan Biffl
Assistance: Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Richard Mordinyi
Dipl.-Ing. Dr.techn. Dietmar Winkler

Vienna, 19th February, 2018

Kristof Meixner

Stefan Biffl

Erklärung zur Verfassung der Arbeit

Kristof Meixner, BSc.
Sauerbrunnerstraße 2a, 7033 Pöttsching

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. Februar 2018

Kristof Meixner

Acknowledgements

I want to thank my advisors Stefan Bifflˆ, Richard Mordinyi and Dietmar Winkler, for their valuable guidance and advice, for keeping up the pace, and their invaluable support, especially at the end of the work. I want to thank Richard Mordinyi in particular for the initial idea of the work and its continuous evolution. I also want to thank Rick Rabiser for some goods ideas and references for the thesis. Furthermore, I want to thank $C_8H_{10}N_4O_2$ for keeping me awake and $C_6H_{12}O_6$ for keeping me happy.

Finally, I want to express my very profound gratitude to my mother, my father, my loving sister, my daring brother, and especially to Julia Obdrzalek for their endless support over time and over the course of some of the most challenging times in life. This accomplishment would not have been possible without every single one of them. Thank you!

Kurzfassung

Große Planungsprojekte im Anlagenbau finden in einer multidisziplinären Umgebung statt, in der Ingenieure aus mehreren Disziplinen an einem gemeinsamen Ziel arbeiten. Arbeitsprozesse in einer solchen Umgebung werden aufgrund limitierter Datenaustauschmöglichkeiten der Softwarewerkzeuge, die tief in der jeweiligen Disziplin verankert sind, durch diese bestimmt. Plattformen für Werkzeugintegration, wie der Engineering Service Bus, integrieren Werkzeuge und Arbeitsprozesse nahtlos. Solche Plattformen müssen aber an die jeweiligen Werkzeuge und Arbeitsprozesse der Kunden angepasst werden. Heutzutage wird diese Anpassung durch Anwendungsintegratoren in manueller Arbeit durchgeführt, die oft mühsam und fehleranfällig ist. Besonders die Anpassung der Arbeitsprozesse in der Anwendung, deren Konfiguration und die Verbindung mit Softwareservices ist aufwendig und beschwerlich.

Diese Arbeit beabsichtigt die Frage zu beantworten, inwiefern der Anpassungsprozess durch einen weiterentwickelten Ansatz verbessert werden kann. Dafür, wird untersucht wie Varianten von Arbeitsprozessen auf Varianten von Softwareservices abgebildet und mit einer automatisch unterstützten Methode konfiguriert werden können. Dazu wird erforscht, wie Konzepte der Variabilitätsmodellierung für Arbeitsprozesse und Softwareservices adaptiert werden können, um Varianten beider Gruppen aufeinander abbilden zu können.

Im Forschungsteil werden zuerst ähnliche Arbeiten, auf denen aufgebaut werden kann angesehen. Danach, wird Variabilität in den Arbeitsprozessen von Industriepartnern und den Services des Engineering Service Bus erforscht. Basierend darauf, wird ein Ansatz vorgeschlagen, um Varianten von Arbeitsprozessen auf solche von Softwareservices abzubilden. Schließlich, wird der Ansatz anhand eines Beispiels eines Industriepartners mit einem Prototypen evaluiert.

Die Resultate der Arbeit sind, ein Ansatz um Variabilität anhand der Business Process Model & Notation Sprache und Feature Modeling darzustellen, sowie der Vorschlag einer Methode um Varianten aufeinander abbilden zu können. Die Evaluierung zeigt, dass der Ansatz machbar ist und den Traditionellen in Komplexität und Aufwand signifikant reduziert. Zusätzlich, werden die Möglichkeiten für qualitätsgesicherte Maßnahmen erhöht. Der Ansatz wurde an einem einfachen Beispiel getestet und zeigte die Überlegenheit gegenüber dem traditionellen Ansatz. Der Autor geht davon aus, dass die Methode noch vorteilhafter für größere Beispiele ist, was in einer zukünftigen Arbeit zu überprüfen ist.

Abstract

Large-scale projects in *Production Systems Engineering* occur in multidisciplinary environments were engineers of different domains work together in a combined effort. Due to their limited integration and connectivity, specialized engineering tools, deeply-rooted in these domains, determine the *Engineering Processes*. *Tool Integration Platforms*, like the *Engineering Service Bus*, seamlessly integrate processes and tools. Nevertheless, *Tool Integration Platforms* need to be tailored to implement the customer's specific processes and tools. Today, application integrators manually perform the customization process, which is tedious and often error-prone. The customization of *Engineering Processes* and the configuration of their connection to software services is particularly costly and cumbersome.

This work aims at answering to what extent the customization process for *Engineering Processes* can be improved using a more sophisticated approach than the manual one. Therefore, it investigates how variants of *Engineering Processes* can be mapped to service variants and configured adequately by a (semi) automated method. Beforehand, we need to examine how concepts of *Variability Modeling* can model *Engineering Processes* and software systems to map variants of either domain?

The research approach, first, investigates related work. Second, variabilities in *Engineering Processes* of industry partners and the services of the *Engineering Service Bus* is examined. Afterwards, variability models for *Engineering Processes* and *Engineering Service Bus* services are developed. Based on this, we propose an approach for mapping *Engineering Process* variants to service variants. Finally, the solution approach is evaluated based on a real-world example of an industry partner and a prototype.

As results, the thesis proposes an approach to define variability models based on the Business Process & Model Notification language and Feature Modeling. The main result is a method to map process templates to Feature Models. The evaluation shows that the solution is feasible and that the manual approach was reduced in effort and complexity significantly. Moreover, the proposed solution increased the number of quality assurance mechanisms. The solution approach was evaluated on a small sample and showed its superiority compared to the manual approach. The author expects the solution approach to work even better on more extensive examples, which has to be proved in future work.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Large-scale projects in the context of *production systems engineering (PSE)*, such as planning and constructing power plants or cold rolling mills, are set in a multidisciplinary environment. In such project settings engineers from different domains, like mechanical, electrical and software engineering are assembled to a joint team, to perform their development effort.

Each engineering discipline has its individual view on the model of the production system under planning. Nevertheless, the various models have commonalities which, from an engineering perspective, can be seen as 'common concepts' [42] and act as interfaces between several disciplines. Plant planners, for example, design entire building blocks or factory floors and concentrate on statics and structural properties. Mechanical engineers then define and place construction units like generators, working cells or conveyor belts on the previously created floor plan. Electrical engineers, next, specify the control hardware of the construction units and map the wiring between them. Finally, software engineers program the control logic and software of the control hardware, as well as customize and implement the software used by factory workers to control the production system. Although this seems like a sequential process, working tasks are started and processed in parallel by the engineers involved [74], once the requirement specification is roughly defined. Consequently, the results of their work need to be joined and merged at a later point.

To perform the engineering tasks, distributed over multiple disciplines, engineers use specific tools to define and manipulate their model of the system. These very specialized tools usually only contribute to a single discipline and produce proprietary artifacts, which means they can not be read and used by other tools. Furthermore, such tools are most of the time isolated from each other regarding data interchange and connectivity.

This tool isolation is owing in large part to tool providers focusing mainly on a particular engineering discipline and its unique requirements.

Some tool providers offer tool suites [8] that support an assorted set of engineering disciplines from an industry sector. However, these suites usually do not entirely meet the needs of a multidisciplinary environment [20]. Therefore, often custom solutions are 'patched together' with tools applied in ways they were not meant to be [8], which frequently results in erroneous engineering data. An example of such a custom solution would be the utilization of spreadsheets to manipulate *Comma Separated Value (CSV)* data exports from one engineering tool, and their later import to another engineering tool. If at a single point of this chain the character encoding of the files changes, the target tool might not able to read the data any more or even worse, loses valuable information. Due to these limitations of tools and tool suites, project managers often utilize tools that seem most suitable for their project requirements and combine them in a 'best of breed' toolset, which under normal conditions undoubtedly raises similar problems.

These issues lead to a highly heterogeneous tool environment, which forces engineers to utilize processes that are heavily determined by the tools used [11]. Such processes induce, for instance, that various artifacts need to be exchanged and synchronized on a regular basis between the engineers of the involved disciplines to establish model consistency on a project level [75]. Achieving sufficient consistency requires disciplined cooperation among numerous participants and detailed coordination of the work during each engineering phase. The very scale of such projects makes it hard to ensure such objectives without the support of additional tools.

Biffl and Schatten [3] in their work described the concept of an '*Engineering Service Bus (EngSB)*' to enable the development of an integrated engineering environment. Their idea derives from a software architecture concept called *Enterprise Service Bus (ESB)*, which illustrates a structure that provides a platform based communication system for distributed software services with the aim to allow an information interchange between several participating applications. The *EngSB* follows this approach and adapts it for *PSE* to allow an information interchange among engineering tools and disciplines to foster seamless cooperation in large-scale engineering projects. To achieve this goal, an open platform named *Open Engineering Service Bus (OESB)* was implemented, that provides features to a) store different engineering models and their data in a versioned fashion and propagates change between them; b) integrate engineering tools coherently by offering extendable data endpoints; and c) deploy executable process models that link process tasks to software services. Furthermore, the platform allows a flexible usage and extension of its components and functionality through the application of the *OSGi (OSGi)* specification. The *OSGi* specification is a standard specified by the *OSGi* alliance[1], which aims at the modularization of software components and the interoperability of services to increase productivity.

[1] OSGI Alliance – `https://www.osgi.org/`

1.2 Problem Statement

In the prior section, we briefly discussed the present issues of multidisciplinary engineering in large-scale industrial projects, leading to heterogeneous tool environments with a lack of data integration and unsuitable engineering workflows. We also introduced the *EngSB* as tool integration platform, which helps to implement an integrated engineering environment to enable better cooperation between engineering disciplines.

However, to serve different industry sectors and customers of different domains, the *EngSB* and its features require tailoring in a *Customization and Configuration Process (CCP)* according to individual needs. Customization, in this context, means the adaptation of the platform to the customer's engineering culture and tool landscape in a combined effort with relevant stakeholders of the customer.

Engineering Integration Ecosystem

The implementation of an integrated engineering application and the *CCP* of this application is embedded in a broader software ecosystem, which involves several software components and stakeholders. The concepts of the ecosystem are explained, using Figure 1.1, in the paragraphs below. The *CCP* is indicated in the figure by the arrow between the platform and the instance, and extended for better understanding, at the lower part of the picture.

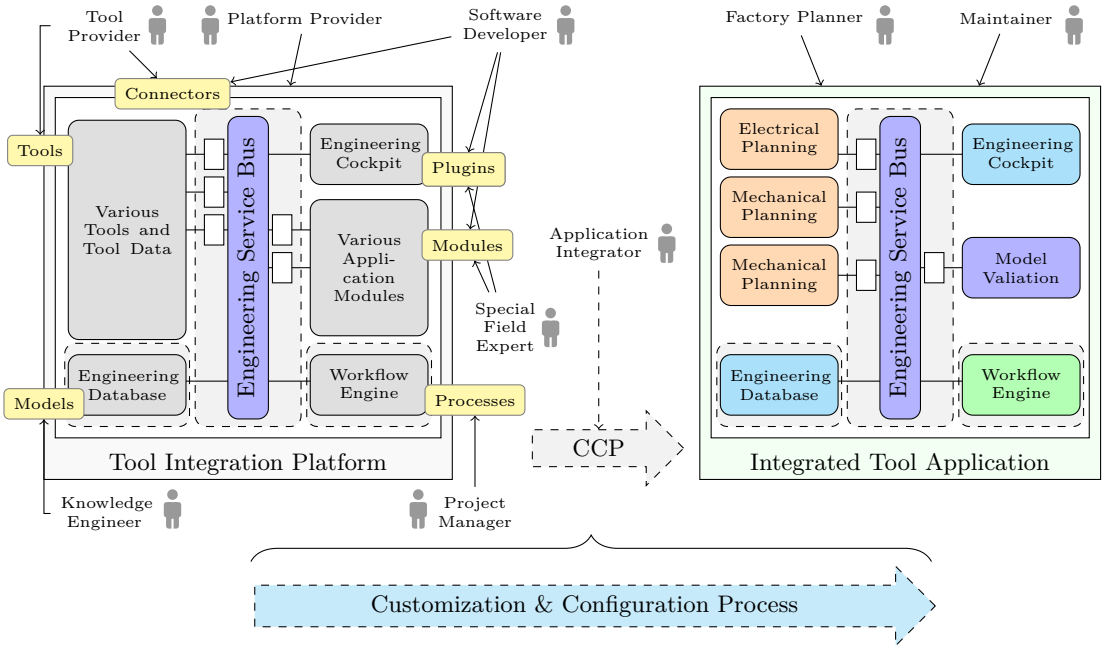

Figure 1.1: Stakeholders and components in the *EngSB* approach

On the left-hand side of the picture the stakeholders, the tool integration platform and

available components are displayed, that are involved in the implementation, customization, and configuration of the integrated engineering environment. For instance, *tool providers* are stakeholders that provide *(engineering) tools*. *Tool providers* but also the *platform provider* offer *tool connectors*, which, for example, can be used to read data from engineering tools.

On the right-hand side of the figure the integrated engineering application, resulting from the customization, and the stakeholders involved in the engineering and maintenance of the production systems over its life cycle are depicted. The factory operator runs the facility and, with the help of the factory maintainer, keeps up its status.

However, some stakeholders are concerned with both phases, like the *project manager* and the *knowledge engineer*, that provide essential information for the *CCP*. For example, during the tool integration, the *tool connectors* are adapted by *software developers* on behalf of *project managers* that define the requirements of the tool integration platform. The most relevant stakeholders related to the *CCP* are discussed shortly in the following paragraphs.

When an engineering company decides to implement an integrated engineering environment in cooperation with a platform provider, a *project manager*, experienced their typical projects, in combination with an *application integrator* from the platform provider, are mandated to set up the artifacts for the integration infrastructure. These artifacts include specifications of tool requirements, process descriptions of existing engineering workflows as well as engineering data models, that are defined by knowledge engineers. The *application integrator* with the knowledge of the engineering company's needs then initiates the *CCP*.

On the one hand, the *project manager*, as mentioned in Section 1.1, aims at obtaining an integration solution, that best supports the needs of the company's engineering projects. Project managers, therefore, demand a well-arranged software toolchain, which can be easily accessed and managed for their purposes. Also, the engineering models used in such systems should be presented with reasonable complexity, to support comprehensibility. An example of breaking down the initial complexity would be to enable definable views for shared large data models.

On the other hand, the *application integrator* must care which of the needs can be satisfied with an integrated solution and which exceed the possibilities. Moreover, application integrators need the *CCP* to be effective and efficient, to realize the integrated engineering solution with as minimal effort s possible. Therefore, they need a set of tools to configure the system solutions in a manner that is functional for them.

The third important stakeholder involved in the *CCP* is the *application maintainer*. After the installation of the integrated engineering application by the platform provider, the *application maintainer* is responsible for the correct operation during runtime. Furthermore, the *application maintainer* updates the platform and installs additional features, which might be needed for project management or due to changes in the used engineering toolset. This stakeholder aims to maintain the application and the tool

environment with the most effective and efficient configuration and analysis tools to avoid unnecessary downtime or more severe issues like data loss.

Customization and Configuration Process

We previously mentioned that the *EngSB* requires being tailored to the specific needs of the engineering company to be implemented successfully. While the *EngSB* supports this, the *CCP* still remains a complex task in practice. Figure 1.2, shows the sequence of activities that needs to be performed in the *CCP* of the *EngSB*. Based on this the complexity of the process and the arising issues are explained later on.

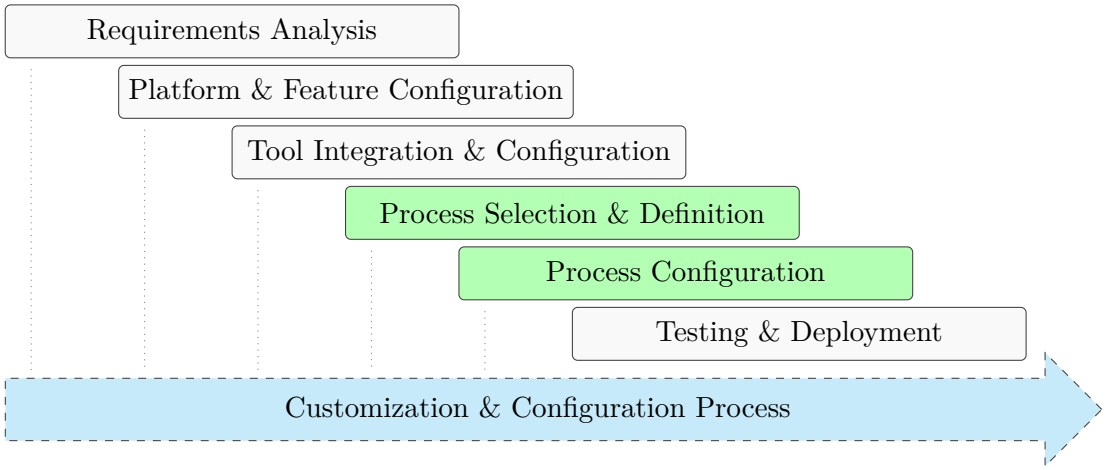

Figure 1.2: Steps of building a customized platform

First, in the *Requirements Analysis* activity, the demands of the customer towards an integrated engineering application are collected and described with the help of the project manager. A list of the tools, used by the client's engineers to perform their tasks, needs to be collected. Another goal in the same stage is to illustrate the workflows of the engineers and individual disciplines. Apart from the primary engineering tools and workflows, the use of alternative ones within a company is not often visible and rarely documented, which means the analysis should aim at revealing them. In this step, the workflows and the application of tools should the thoroughly investigated to optimize existing and developed novel discipline crossing processes.

In the *Platform & Feature Configuration* activity of Figure 1.2, the application integrator selects the components from the platform skeleton and additional modules, that fit the requirements of the customer, and configures along with the platform. If not already existing in the platform repository, software components that provide the demanded functionality, need to be implemented.

The *Tool Integration & Configuration* step is the next phase in the *CCP*. Tool connectors, which are applicable in the particular customization, are configured according to the configuration of the engineering tools. Otherwise, it needs the implementation of tool

connectors for the particular tool or tool group, which often requires additional interfaces or software from tool and plugin providers. The same holds true for models of the tool data formats, which possibly have to be developed by knowledge engineers and implemented by software developers in a model language known to the platform. Additionally, it requires the formalization of transformations between the models and those used internally by the tool integration platform.

The following phase in Figure 1.2 is the *Process Selection & Definition* activity On the one hand, several processes used in engineering companies are very similar, especially when they work in the same domain. Such processes most likely were already deployed to the platform repositories as models in a formal notation and can be selected and used with little customization or even off the shelf. On the other hand, optimized or novel process descriptions developed in the *Requirements Analysis* are used as blueprints and formalized to models in this phase.

The process models, selected or defined in the prior phase, need to be configured in the *Process Configuration* phase. Therefore, the wiring of the single process activities to the exact services exposed by the components that result from the selection in the *Platform & Feature Configuration* phase. Depending on the process modeling language this is either done in a separate file or in the process model itself.

Finally, in the *Testing & Deployment* phase the partial contributions of the prior activities must be collected and tested, for example, with integration test. Once these tests complete with a satisfying result, the artifacts need to be assembled to a deployable application to derive an integrated engineering application that works as expected from existing components. The created application, again, needs to be extensively tested before its release to the customer.

Challenges of Customization

The gathering of requirements in the *Requirements Analysis* phase, depicted in Figure 1.2, is usually done in a narrative or semi-structured way which often results in loose descriptions of customer needs. Hence, these requirements might be interpreted in different ways, which leads to problems during the customization process or even worse in the running system. Moreover, it is challenging to ensure, that existing platform components meet customer needs, due to the absence of appropriate mechanisms to automatically match requirements and capabilities.

Furthermore, the selection of the software components, in the *Platform & Feature Configuration* and *Tool Integration & Configuration* phases, is associated with high efforts since their feature and capability representation rarely exists explicitly. Beyond that, missing dependency management often leads to the effect, that dependency conflicts in many cases occur not until the system is up and running.

Two particular complicated parts of the *CCP* are the *Process Selection & Definition* as well as the *Process Configuration* step.

First, the application integrator must verify, that for all processes the corresponding sub-processes are selected. Second, the application integrator must assure, that all process activities are wired to services and, also, that these services are deployed with the application. These problems occur despite the fact that the *EngSB* can interpret process definitions in different model notations, which eases the configuration of processes.

So one part of the complexity of the *CCP* arises from the integration of the engineering tools and their often, proprietary models and export formats. The other part of the complexity results from the tedious and most of the time manual configuration of the system, its components, and processes. This complexity causes the effect that the configuration of such software systems is time-consuming, often error-prone and inefficient which leads to incorrect results [6]. Besides the issues mentioned above, the quality of the resulting software systems is difficult to ensure [10]. This shortage of quality is because of missing metrics for integration solutions, but also due to the lack of automatically generated and executed end-to-end tests.

But in today's software engineering pluggable components, specially adapted processes, or even customized adaptations of the software itself are more common than ever [9]. Thus, platform providers face the increasing demand of supplying additional and extended functionality, which allows a flexible configuration and takes a broad spectrum of customer requirements into account [56, 71]. Also, it is emphasized, that customizations should be implemented with minimal effort to prevent high costs of development [11].

The issues but also the demands referenced raise the need for approaches to (semi-) automatically generate and test parts of customized system solution instances, as well as to assemble them to a deployable integrated engineering application. In the context of systems and tool integration [4, 41, 6] propose model-driven approaches, to automatically generate configurations for system integration solutions.

A mainly model-driven approach that gained momentum in research in the recent years is *Variability Modeling (VM)*. *VM* is the task of identifying commonalities and variabilities in a *Software Product Line (SPL)* and modeling them in formal notation. *SPLs* (or *Software Product Families (SPFs)*) allow related software products to be based on a platform that shares common features while enabling the implementation of additional product-specific functionality. For a *SPL* to be successful in the long run, *VM* is exceptionally important [9]. Various approaches of *VM* exist today, which considering the use cases can be applied to different requirements [18].

The *EngSB* approach is already meant to provide a platform that can be extended and customized to different needs, and is, therefore, well prepared for the adaptation to an SPL. So the constructs of *VM* and *SPL* seem promising to realize approaches to tackle the challenge of generating parts of customized integrated engineering applications. Optimized business processes can be a competitive advantage [14] for companies and are thus of higher interest to be improved. The same holds true for optimized engineering processes that save engineering effort, but also ensure a higher quality of engineering results.

As the configuration of processes is an excellent example of the overall customization process and also highly valuable, this thesis concentrates on the topic of *automatically generating quality assured and configured process variants for integrated engineering solutions* (see component 'Workflow Engine' and 'Processes' in Figure 1.1). The thesis understands configured engineering process variants as processes, that correctly refer to the services, which are deployed with the integrated engineering application.

1.3 Aim of the Work

The goal of this thesis is the development of an approach that enables the generation of customized and configured process definitions from base processes and platform services for integration systems configuration and customization in heterogeneous engineering tool environments. Additionally, a prototype with tool support will be implemented for this thesis to prove the feasibility of the approach. Moreover, the performance of the prototype will be analyzed and evaluated to show its suitability in the field and to disclose benefits for the process of customization and configuration.

This approach should equally address the scientific communities of *SPL* and *Business Process Management (BPM)*, as well as *Model Driven Software Engineering (MDSE)* to point out a possible advantage to link these research topics in an applied field. Furthermore, the approach should serve industrial partners as a basis to implement the findings in practice for their requirements to enhance their actual process of customizing tool integration platforms for their clients.

To achieve this goal, concepts from *VM* and *BPM* alike, will be collected and evaluated, to select the most promising for the approach. The evaluation will include their suitability for the use case, but also their tool support and their backup by the scientific communities.

Based on this analysis, a theoretical solution approach will be carved out from the requirements of the use case and the selected concepts. This method is then implemented in the prototype, tested, based on an example from industrial partners, and evaluated employing the defined metrics and its feasibility for the *CCP* in practice.

The main expected results are the following:

- A description and explanation of the proposed approach
- A set of metrics and key performance indicators to measure the performance of the approach in contrast to a manual *CCP* in an integration solution
- An implementation of a prototype with tool support for the *CCP* based on the proposed approach
- A collection of data that result from a manual as well as prototype-based configuration and customization

- An evaluation of the approach and prototype according to the defined key performance indicators

1.4 Structure of the Work

The remainder of this work is structured as follows. Chapter 2 is divided into two parts. The first part gives an introduction to *VM* and discusses related work and state-of-the-art approaches of *VM* in software engineering and *BPM*. Furthermore, existing approaches that attempt to generate deployable software instances from models automatically are presented. The second part identifies applications and tools from industry and research that support the research topic and explains how they contributed to the approach resulting from this thesis. Chapter 3 defines the research issues of this thesis, based on the work done in the CDL-Flex laboratory and questions that emerged during the research there, as well as insights acquired from the literature research. Additionally, it roughly sketches the criteria that were used to evaluate the solution.

The methodological approach to develop a solution considering the research challenges is outlined in Chapter 4, which also details on the evaluation strategy and its key performance indicators. A motivational scenario that serves as an operational example for the thesis is selected from an industrial use case and illustrated in Chapter 5. Chapter 6 thoroughly describes the developed solution, which utilizes *Feature Modeling (FM)* from *SPLs* and abstract *Business Process Model and Notation (BPMN)* models in order to derive business processes that are linked to software features and can be deployed to a *Business Process Management System (BMNS)*. Furthermore, the implemented prototype of the solution is explained in detail. Chapter 7 evaluates the performance of the solution employing the implemented prototype according to the raised research issues and the defined evaluation indicators.

The results of the solution approach and the evaluation of the prototype are summarized and discussed in Chapter 8. Finally, Chapter 9 gives a conclusion based on the findings of the discussion and presents an outlook for future research.

CHAPTER 2

Related Work

The previous chapter introduced the topic of application and tool integration in the engineering domain and discussed the challenges of customization and configuration of individual customer instances.

This chapter discusses related work relevant for this thesis and presents state-of-the-art methods and models that are exploited to develop the solution approach. In this way, the research issues relevant to the work can be carved out in Chapter 3. Section 2.1 discusses the current state of engineering tool integration and a solution approach that was proposed and evaluated in several industrial cases. Section 2.2 describes several methods of process modeling that act as solution candidates to represent the engineering processes present in the *production systems engineering (PSE)* domain. Finally, Section 2.3 outlines the two most common concepts of variability modeling and their suitability for the following proposed solution approach.

2.1 Engineering Tool Integration

As described in Section 1.1, today's industrial software systems need to provide solutions for engineering tasks that are distributed over multiple disciplines and often span teams over different locations. These software systems and the utilized engineering tools are embedded in a software ecosystem, which is highly complex and tends to grow over time to serve different project needs. Data interchange and the connectivity between tools and software components are crucial to support multidisciplinary engineering processes.

2.1.1 Point-to-Point Tool Integration

The issues of engineering tool data exchange are nowadays, still often tackled with *point-to-point (PTP)* integration solutions of the tools used and their data pools [20]. Such solutions grow over time, due to the growing complexity of companies or because of

step-by-step digitalization of the engineering work. Figure 2.1 illustrates an example of such a tool integration solution. Each engineering domain involved uses specific tools that persist data into their particular data pool or create individual artifacts. Furthermore, each data pool has to be connected via its particular interfaces to the other data pools, to enable data exchange.

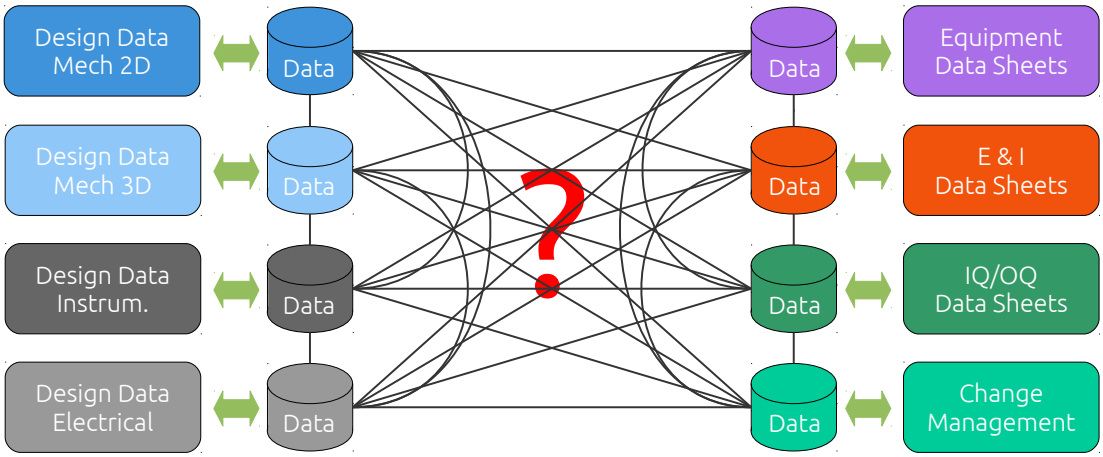

Figure 2.1: Tools, their data pools and *PTP* integration (based on [20] & [26])

While providing a certain level of integration, *PTP* solutions bring up further problems. For instance, with the introduction of a new tool to the environment, the number of connections needed increases exponentially [57]. With exponential growth, additional connections mean high development efforts for the implementation of tool connectors and imply the adaptation and maintenance of components, which were finished a long time ago. This necessity often results in an unmanageable infrastructure and, due to time pressure, in a poorly documented architecture. The task is even more complicated due to various proprietary software tool interfaces and costly due to license fees for these interfaces. Furthermore, the connections in a *PTP* network imply the risk of single points of failure. If one of the connections fails, the whole infrastructure might render unusable. Moreover, *PTP* data exchange prevents sufficient consistency management, as changes need to be propagated to the different data pools. *PTP* integration tends to become highly complicated and brittle over time.

These issues have the following implications on *PSE*. Project managers that aim at obtaining an integration solution, that best supports the needs of their specific project, as mentioned in Section 1.1, have a limited overview of the heterogeneous data sources, which makes accessibility difficult. This restricted overview, in turn, hinders, for example, proper risk management or test automation on a project level, because often not all of the data sources can be utilized. Additionally, system integrators aim at ensuring information exchange along defined toolchains. However, the manual and proprietary configuration of *PTP* solutions at a low technical level are time-consuming and often error-prone. Overall *PTP* integration makes round-trip engineering, as mentioned in

[20] increasingly complicated and might lead, for example, to deadlocks when engineers perform commits in parallel.

2.1.2 Integration with the Engineering Service Bus

In contrary to the *PTP* approach, [3] proposed the concept of the '*Engineering Service Bus (EngSB)*', which allows flexible and efficient integration of engineering tools in a heterogeneous engineering environment. In [5, 7], the authors extended the model to *PSE* due to the need in such environments to cut set-up efforts of tool landscapes and reduce the complexity of projects. To separate between the requirements during the *design time phase* and the *run-time phase* of projects, they distinguish between the *EngSB*, which is used for planning and design, and the 'Control Service Bus', which is used to monitor and control the built factory. Even though these two concepts together build the so-called *Automation Service Bus (ASB)*, the *ASB* is a customization of the *EngSB* for the automation domain which considers domain-specific software tools. As this thesis focuses on the design-time phase of the application, it will stick to the term *EngSB*.

The *EngSB* utilizes ideas from the *Enterprise Service Bus (ESB)* [16] and applies them on *PSE*. An *ESB* provides the means to implement a scalable, decoupled, and distributed network communication infrastructure for integrating enterprise applications on a high level. The *ESB*, therefore, uses concepts like web services, messaging, and routing to enable and control the connectivity and interchange of software services within and among companies.

However, several shortcomings of the *ESB* approach were identified [3, 5], that make it infeasible for the application in combined software and engineering environments. For instance, the *ESB* concept usually requires integrated applications to be always available. However, this is not the case in *PSE*, where off-shore offline teams need to create and adapt plans in engineering tool while they are on the production site. In the same scenario, the *ESB* approach has drawbacks, because it is designed to connect heavy-weight business applications instead of light-weight applications that might run on a laptop computer. Furthermore, the *ESB* was built with the intention to connect applications with a well-supported data exchange interfaces, which is not the case in many engineering applications.

The *EngSB* approach was conceptualized with these issues in mind and to enable, besides others, a) stable engineering processes, b) the flexible and efficient configuration of integration solutions, and c) a mix of backend and frontend tools, for example by utilizing notifications on artifact changes of frontend applications.

Figure 2.2 shows the architecture of the *EngSB* and its basic building blocks. Components in the figure on a gray background run within the *EngSB* container. The core of the *EngSB* is a message or event broker that receives messages from different components and eventually transforms those messages to normalized messages. Afterwards, these messages and sent and propagated to other components, that register for the messages or topics. A message broker also enables an improved asynchronous communication between

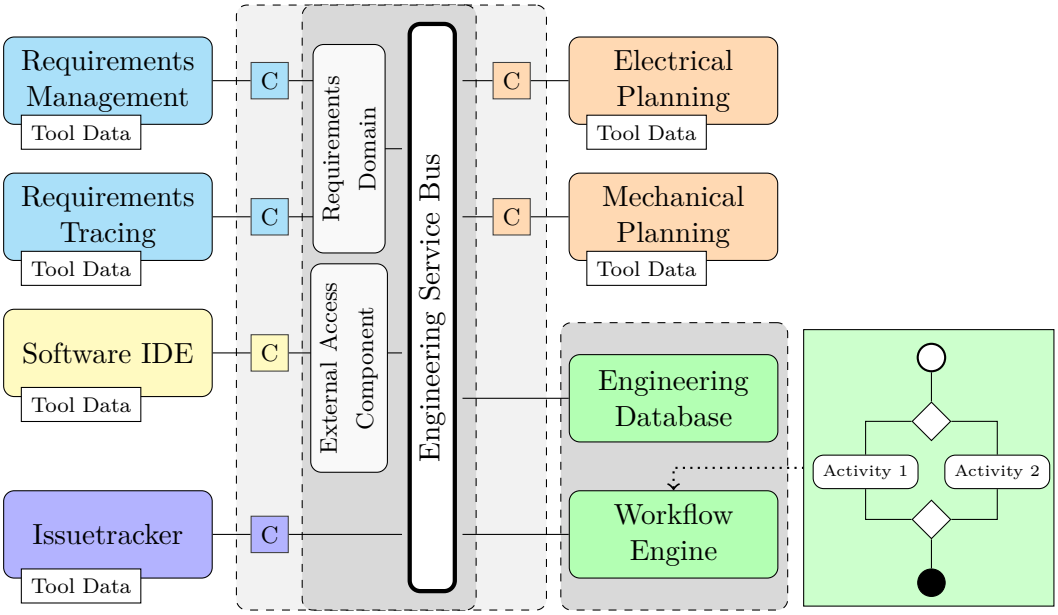

Figure 2.2: *EngSB* with connectors and workflow engine (based on [3])

endpoints, which is the basis for the mentioned support for off-shore teams, and the implementation of well-known enterprise integration patterns [28].

One way of integrating tools to the *EngSB* is via *tool connectors*, that bridge between the tool and the *Application Programming Interface (API)* of the *EngSB*. Connectors are an integration instrument, well known from the *ESB* approach. Two forms of tool connectors can be distinguished. The first one is data integration, were tool connectors read the specific data model of the tool from their export interfaces and translate them to the data model internally used by the *EngSB*. Data coming from the *EngSB* is transformed in the tool connectors to the vendor specific data model and then imported to the tool. Functional integration is the second form of data integration, supported by the *EngSB*. In this case, engineering tool vendors provide *APIs* that are implemented in a tool connector and exchange the data models via these interfaces. The usage of *APIs* assumes cooperation with the vendor, who also defines the level of integration by the functionality of the *API*. In Figure 2.2, the *Electrical Planning* tool, for example, connects via a specially implemented connector to the *EngSB*.

The other way of tool integration, which was introduced in the *EngSB* approach, are tool connectors that, instead of connecting to the bus directly, use *tool domains* as intermediates. A tool domain is a component in the *EngSB* that acts as a standardized facade and defines common functions and unified data models for tools from the same type. Therefore, tool domains enhance stability in the engineering tool environment, as different tools from the same domain conform to a stable shared model. Furthermore, it empowers the interchangeability of engineering tools, which makes the utilization

of different off-the-shelf tools possible. Additionally a tool domain provides a) data mapping from the tool data format to the domain data model, b) data enhancement, for instance by enriching the tool data with statistical data for management use, and c) functional enhancement, like logging features to control data access. In the figure above the *Requirements Management* and *Requirements Tracing* tools connect to the tool domain *Requirements Domain*. This tool domain uses the attributes of requirements as the common model and implements functionality for their manipulation.

Another component that can be utilized to connect to the *EngSB* is the *External Access Component*. This component exposes the *EngSB API* to webservices like REST [22], which can be used by external tools and applications. An example depicted in Figure 2.2 is an integrated software development environment like Eclipse[1], that acts as a passive component only listening to events of the *EngSB*.

Two core components of the *EngSB*, displayed in Figure 2.2, are the *Engineering Database* and the *Workflow Engine*. The *Engineering Database* [72] is a store, which holds the data of the engineering tools in specific models and provides features like data propagation between models and querying of the underlying data. The store versions the data, which means, it saves every stage of the data and can reproduce this stage at an arbitrary point in time. Finally, the *Workflow Engine* is a component that enables the usage of process description languages like *Business Process Model and Notation (BPMN)* or Drools[2] to implement long-running engineering processes over several tools and services.

2.1.3 Engineering Service Bus Implementations

As mentioned in Section 1.1 one implementation, that derived from the *EngSB* approach, is the *Open Engineering Service Bus (OESB)*[3] which is based on the *OSGi (OSGi)* container Apache Karaf[4]. The *OESB* supports the concepts introduced by the *EngSB* approach and provides an engineering model description based on Java classes.

Based on the *OESB*, the *AutomationML Hub (AML.hub)* [75, 76] was developed, which utilizes *AutomationML (AML)* as description language for the engineering tool data models. *AML* is a standardized 'Industrie 4.0' data exchange format based on XML, that utilizes selected other markup languages to enable the modeling of engineering data from different engineering perspectives [29, 30, 31]. Going back to the *PSE* example from Section 1.1, for instance plant planners can describe the overall plant topology in the CAEX format, while mechanical engineers create models of the plant units in Collada and software engineers implement functionality with PLCOpen. Using this format, engineers and industrial partners can exchange a single *AML* file, which references all of the mentioned parts.

[1] Eclipse Foundation – `http://www.eclipse.org/`
[2] JBoss Drools – `http://drools.org/`
[3] Open Engineering Service B us – `http://openengsb.org/`
[4] Apache Karaf – `http://karaf.apache.org/`

2.2 Process Modeling

Most companies have established specific processes to perform their daily work. The term business process subsumes these processes. However, different kinds of processes, like sales, software development or the mentioned engineering processes exist. A *business process* consists of a set of related activities and control flows between these activities with a predefined order and defined start and end points that realize a common business goal [15, 14, 2, 17].

An example of a business process would be the approval of an insurance claim, that might consist of the following tasks. First, a customer submits a claim to the insurance company. Second, an employee checks the claim for validity, for example, if the person submitting the request is insured or if the specific policy covers the claim. In an further activity, the employee might request additional evidence for the claim. Finally, after receiving additional information, a person responsible for the approval of the company decides whether the claim is justified and the customer gets a compensation.

Either business processes existed implicitly and evolved as a part of the company's culture, or they were defined and documented at a particular point in time. For instance, if a company plans the implementation of a quality management system, like the ISO:9000 or ISO:9001 standards, the careful planning, documentation and establishment of standardized processes is needed [32, 33]. *Process modeling* is a model-based approach to explicitly document and visualize business processes to make them comprehensible for non-experts [59]. On top, formal notations and languages define grammars that help to verify the correctness of process models and make them interchangeable. Some of these notations even support business processes to be executed in workflow engines, which is of growing interest among researchers and practitioners [17].

Today various notations and languages exist that are utilized for modeling and exchanging business processes [40]. To evaluate different workflow engines [69] identified some patterns, that represent requirements for a comprehensive workflow language. The next sections outline four different, commonly used methods to define and model business processes, which support these workflow patterns.

2.2.1 Petri nets

Petri nets are directed bipartite graphs that describe a discrete event system and can be used to model business processes [68]. Petri nets provide a graphical notation and are based on a mathematically defined foundation. An example of a Petri net can be seen in Figure 2.3.

The graphical notation defines nodes that are distinguished into *places*, represented by circles, and *transitions*, visualized by rectangles or bars. Places can hold a specific number of *tokens*, represented by dots, in the places. The nodes are connected through *arcs*, with the restriction that arcs must not connect nodes of the same type. Furthermore, arcs can be weighted with a number, which represents the tokens needed to activate an incoming

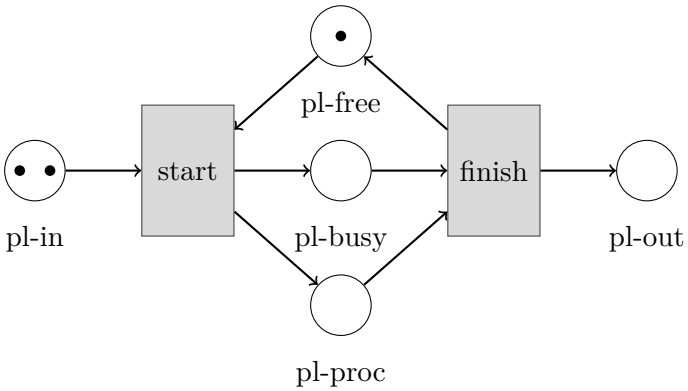

Figure 2.3: Petri net example in ready-to-fire state (based on [68])

arc respectively that are generated by an outgoing arc. A transition fires, if all incoming places hold enough tokens. For example, in the figure *pl-in* and *pl-free* both contain at least one token, which fires transition *start* and generates a token in place *pl-busy* and another one in place *pl-proc*.

A drawback of Petri nets is that even small examples of processes get very complicated and are hard to understand for non-experts. Furthermore, there is currently no standardized exchange format or scalable engine that allows the execution of processes modeled as Petri nets.

2.2.2 Event-driven Process Chains

Event-driven Process Chains (EPCs), developed at University Saarbrücken by August-Wilhelm Scheer in cooperation with SAP AG, provide a flowchart-like, graphical notation to define and re-engineer processes [60]. *EPCs* basically consist of events, functions, connectors, and control flows arranged in an ordered graph to reflect a workflow. Figure 2.4 shows a simple example of an *EPC*.

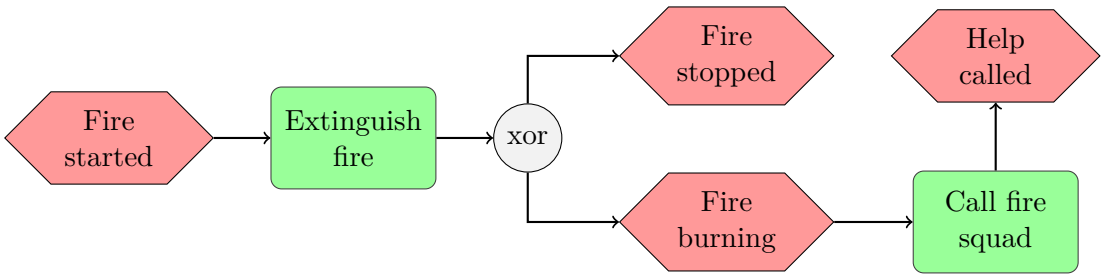

Figure 2.4: Event-driven Process Chain example

Events, represented by hexagons, either trigger functions, or are results of functions which trigger further functions and thus 'drive the process chain'. The beginning and

the end of an *EPC* is always marked by an event. *Functions*, displayed as rounded rectangles, are elements that perform specific actions or transform input data to output data. The events and functions are linked together with *control flows* to create an ordered graph. Through connectors, the logical relation and flow or the *EPC* can be controlled. *Connectors*, represented by circles, split and join the flow where necessary and enable the three logic operations AND, OR, and XOR. Extended *EPCs* provide additional symbols for data, organizational units, and systems, which provide further functionality, which can be used to define the needed resources further.

Compared to Petri nets *EPCs* provide a richer graphical notation and also take additional resources into account, which makes them easier to understand. Furthermore, they provide a more intuitive approach due to the usage of natural language in the description of the nodes. However, *EPCs* is neither open nor standardized as they are part of a licensed model of the ARIS Toolset which is heavily bound to SAP.

2.2.3 Activity Diagrams

UML Activity Diagrams are a behavioral diagram type of the *Unified Modeling Language (UML)* [47], which is a general purpose modeling language established by the *Object Management Group (OMG). UML* defines various diagrams types, that are partitioned into structural and behavioral diagrams, with the intention to describe different aspects of software systems. Therefore, *UML* provides the notation as well as the semantics for the diagrams. Furthermore, it specifies the *Object Constraint Language (OCL)* [46], that can be used to formulate declarative rules for the diagrams, like, for example, that an element must not be its parent. *UML* itself is a strongly formalized industry standard, based on the *Meta Object Facility (MOF)* [49], which defines a meta-data architecture for modeling languages including a meta-meta-model as well as the open and vendor-independent data exchange format *XML Metadata Interchange (XMI)* [48]. The usage of *XMI* enables all models, described in a *MOF* compliant notation, to be exchanged with the same mechanisms and semantics. Figure 2.5 displays an example of an activity diagram.

The main building blocks of an activity diagram are *activities*, represented by rounded rectangles in the graphical notation, for example 'Order pizza' in the figure. Activities are connected via arrows to build an ordered graph. This flow can be controlled with the help of *decisions*, displayed as diamonds, but also *splits* and *joins*, depicted as bars, which indicate a parallel execution of the activities. An activity diagram always starts with one or more *start nodes*, presented by black circles, and ends with one or more *end nodes*, depicted as circles with a centered dot.

With the standardization of the execution semantics for a foundational subset of *UML* [52] and a corresponding action language [51], it is also possible to execute activity diagrams. However, these approaches are in a very early stage, and to the knowledge of the author no common available engines exist, that can reliably execute such diagrams.

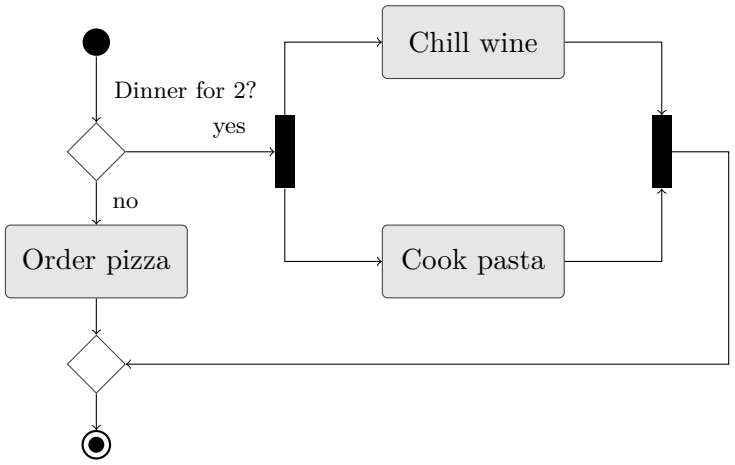

Figure 2.5: Activity diagram example

2.2.4 Business Process Model and Notation

The *Business Process Model and Notation (BPMN)* is a standardized notation for modeling business processes, with the goal to ease the understandability of process models for stakeholders of different domains, on the one hand, as well as to enable the exchange of process models between participants and systems, on the other hand, [44, 17]. *BPMN* was first introduced by the *Business Process Management Initiative (BPMI)*, which merged with the *OMG* in 2005. Until version 2.0 *BPMN* provided a graphical notation only, which made the use of other languages like *Business Process Execution Language (BPEL)* or *XML Process Definition Language (XPDL)* necessary to exchange and execute the process models. *BPEL* is an XML-based orchestration language for web services standardized by *Organization for the Advancement of Structured Information Standards (OASIS)*, which allows to exchange and, with the help of engines, execute business process models. *XPDL* is an XML-based standard for business process exchange defined by the *Workflow Management Coalition (WfMC)*. Since version 2.0, *BPMN* provides a formalized meta-model and an independent electronic exchange format.

BPMN 2.0 provides four diagram types to express different views on business processes, namely process, collaboration, choreography, and conversation diagrams. *Conversation diagrams* describe high-level interactions between different stakeholders leaving out, for example, control flows and process steps. *Choreography diagrams* go into more detail and describe the message exchange between the participants including gateways and decision nodes to control the sequence of steps. Due to the representation of business processes in these two diagram types on a very high-level and their inability to be executed in standard engines, they are not in the focus of a detailed discussion. The collaboration and process diagrams, however, will be described in the following paragraphs. To support the description, Figure 2.6 depicts a simple example of a collaboration and process diagram and Figure 2.7 shows the most important elements of the *BPMN* notation.

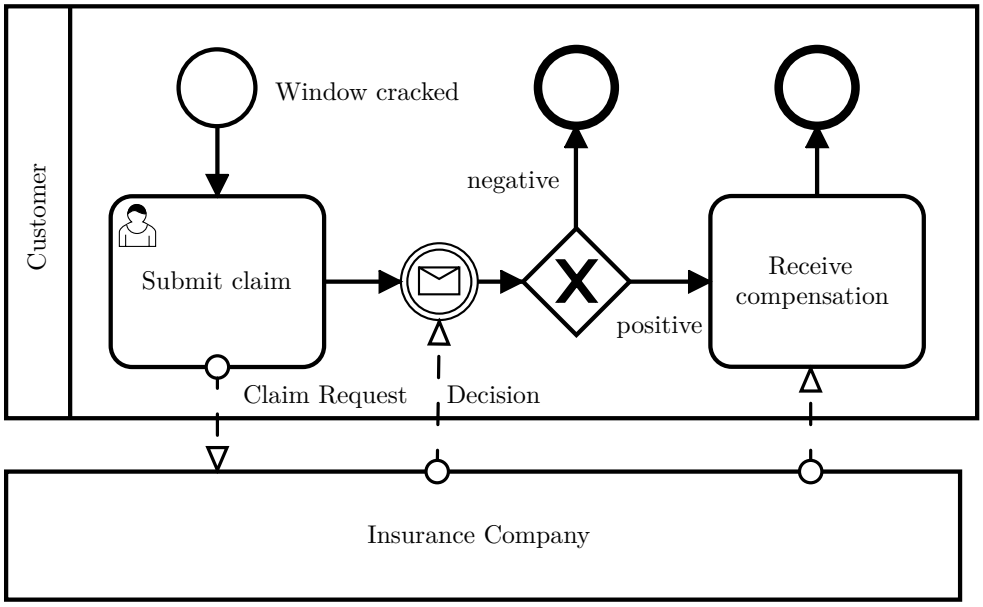

Figure 2.6: *BPMN* collaboration and process diagram example

The basic building blocks of *process diagrams* are flow objects which include events, activities, and gateways, that are connected via control flows. *Events* can be divided in start, intermediate and end events, as shown in the upper right corner of Figure 2.7. *Start events* occur at the beginning of a specific process or sub-process, whereas a process can have a single but also multiple start events. Some start events can catch results, which means they react to a trigger from another process. In the example (see Figure 2.6) the *'Window cracked'* event indicates the start of the process. *Intermediate events* appear anywhere within the process between a start and an end event, changing the flow of the process but without starting or terminating the process. In Figure 2.6, for instance, an intermediate message event is caught in the customer process, that acts as a basis for the decision in the next gateway. *End events* indicate the end of a process, thus terminate the respective process instance and can create a result and throw it, for example to a start event. A process might have a single or multiple end events. The example contains two end events that are either reached when the claim is rejected or when it is accepted and compensated. From Figure 2.7 one can see that different event types exist which can be used to indicate specific behavior in the process model. For instance, *timer events* represent events that fire either after a certain period or on a specific date. In the insurance claim example, a timer intermediate event could be added to express, for example, that after three weeks the customer calls the insurance company if they received the claim request. *Compensation events* can be used to cancel process steps or roll back a state that was done in a process step. An example would be a reservation of a table in a restaurant that is later canceled by a compensation event because the customer got ill.

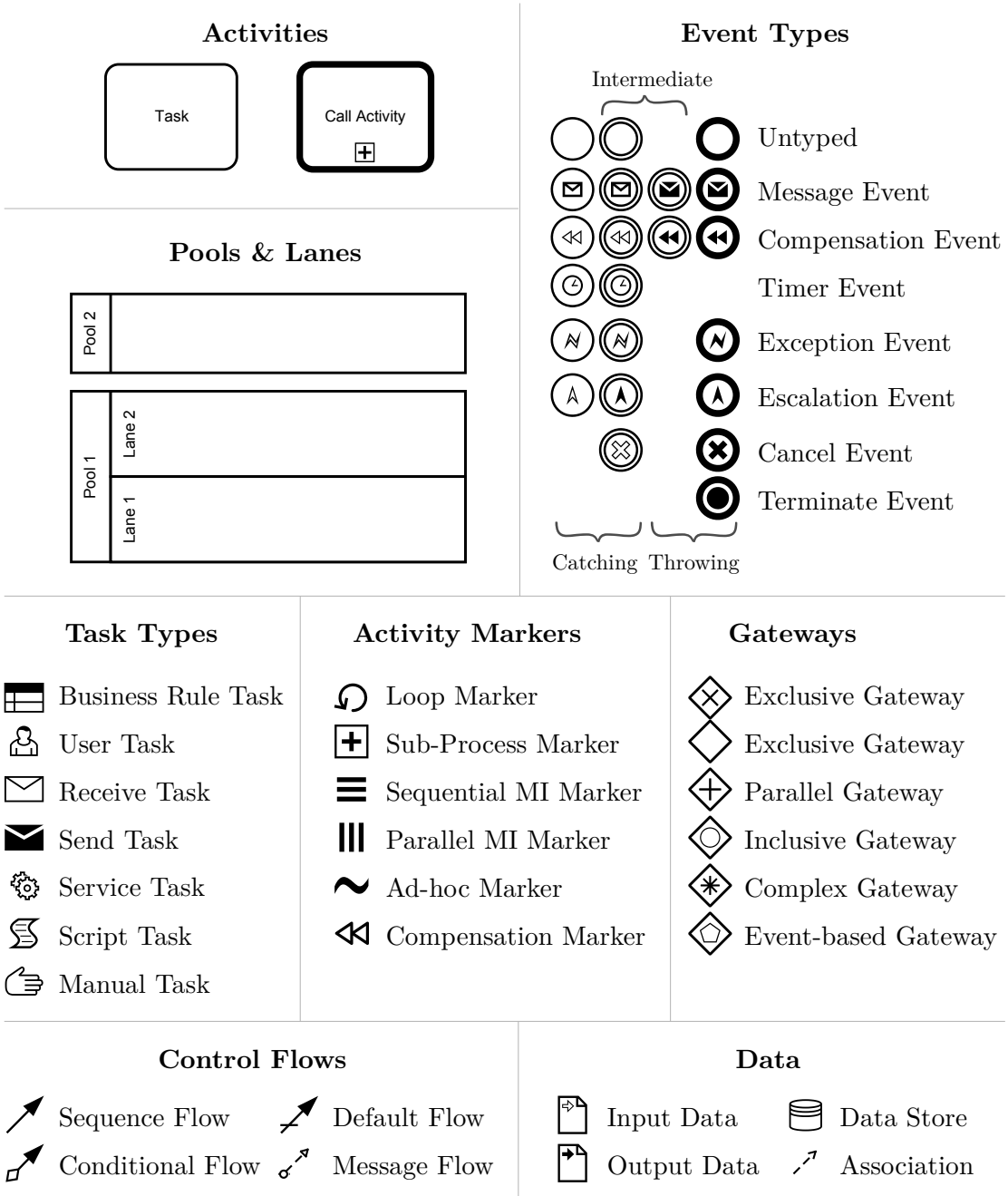

Figure 2.7: *BPMN* 2.0 symbols (based on [44, 17])

Activity in the *BPMN* context is a 'generic term for work that companies perform in a process' [44]. The process diagram distinguishes *tasks*, which are atomic activities and cannot be split into further steps, from *sub-processes*, which are compound activities that

contain several other process steps. Tasks and sub-processes are depicted as rounded rectangles in the diagram, whereas sub-processes have an additional sub-process marker, illustrated as plus sign in a rectangle (see Figure 2.7 upper left corner). Globally defined tasks and sub-processes, which can be re-used in the process diagram, are named *call activities* and marked by a stronger border. Furthermore, activities can have *task types* and *activity markers* assigned. Task types indicate the nature of the activity to be performed, which means who or what executes an action. Figure 2.7 shows, for example, the *user task* and the *service task*, which means that the action is either performed by a user in the system or by a software service. Activity markers specify the execution behavior of the activities, which means how an action is done. Figure 2.7 shows, for example, the parallel MI marker, which indicates that a task is performed in parallel by multiple instances that are spawned, or the compensation marker, that is used for tasks that are rolled back when a compensation event is thrown in the process. To provide an example, Figure 2.6 defines a task *'Submit claim'* that needs to be performed by a person.

Gateways, the third type of flow objects in a process diagram, are used to control the sequence flow. Therefore, an initiating gateway splits the sequence flow into two branches which join later to one sequence flow in a corresponding gateway. This semantics also means, that at a gateway a decision is taken, that determines the triggered branches in the process. In a *BPMN* diagram, gateways are illustrated as diamond shapes with additional markers, which specify the rule which outgoing branches are activated. Such a gateway can be seen in the Figure 2.6, when, depending on whether the decision of the insurance company is negative or positive, the process either terminates or the customer receives a compensation. The most important gateways are *exclusive gateways* and *parallel gateways*, marked by an X or + respectively, as shown in Figure 2.7. At an exclusive gateway splitting the sequence flow, exactly one of the outgoing branches is triggered. At an exclusive gateway joining the sequence flow, the gateway awaits precisely one incoming flow to be activated, before it triggers the outgoing branch. In contrast to this, a parallel gateway triggers all outgoing branches simultaneously on a split and needs all incoming branches to be activated on a merge, to trigger the outgoing branch. *Inclusive gateways* activate one or more branches on a split, which means at the corresponding merge gateway all activate incoming branches need to have completed before the outgoing flow is triggered. *Event-based gateways* are always followed by catching events or receive tasks, such as a timer event. Depending on which of the linked events or tasks happen first, the corresponding branch is activated. *Complex gateways* are defined as gateways that define decision behavior not covered by other gateways.

As mentioned above, flow objects in a process diagram are connected with *control flows*. Control flows are usually standard *sequence flows*, depicted as black tipped arrows that range from the source to the target flow objects (see Figure 2.7 lower left corner), that define the order of activities. Furthermore, *BPMN* defines *conditional flows* and *default flows*. Conditional flows, marked with a small diamond shape on the arrow line, are activated when a certain condition is evaluated to be true. This enables, for instance,

the usage of parallel gateways in combination with additional constraints. Default flows, marked with a slash on the arrow line, are used to label the default branch that is activated if no other condition holds true, for example, in scenarios with complex gateway.

Further elements in the process diagram are *data artifacts* (see Figure 2.7 lower right corner), that are linked to objects in the process diagram. These artifacts can either be single data objects, data collections or datastores, that are associated with the flow objects or connecting objects with dashed arrows with an open arrowhead. In contrast, *data input* and *data output* artifacts act as input for an entire process, respectively result from an entire process.

While process diagrams, with their rich possibilities, can express a detailed description of a single stakeholder's process, collaboration diagrams are needed to illustrate the interactions between different participants. The building blocks of *collaboration diagrams* are pools, processes and message flows. *Pools* represent participants in collaboration and partition the activities of the different stakeholders. A participant can be an organization or a role but also a system like a software application. Pools in the diagram are depicted as rectangles as shown in Figure 2.7, and can either be realized as collapsed pools, representing a black box, or as expanded pools, which then contain a process that shows details of the pool. *Lanes* in a pool are used to further organize and devise pools and contain several activities of the process described in the specific pool. *Message flows* are used to show the information flow between participants across system boundaries and can be attached to pools, activities and message events. In Figure 2.6 the insurance company's process is represented in a collapsed pool, while the customer's process is detailed in an expanded pool. Finally, the interactions between the two participants are realized as message flows that specify the information flow.

Business processes defined in *BPMN* can be executed in various commercial and open-source workflow management systems [1], like Bizagi[5] or Camunda BPM[6]. A drawback of *BPMN* is, however, the ambiguity of the different diagram types and their overlapping scopes as well as the various variants a single process can be modeled with different concepts of *BPMN*.

2.3 Variability Modeling

In Section 1.2 the concept of *Variability Modeling (VM)* was introduced as essential part of *Software Product Lines (SPLs)* and *Software Product Line Engineering (SPLE)*. *SPLE* originates from *Product Line Engineering (PLE)* in industry, which, in contrast to mass production, allows the derivation of a variety of customized products ('variants') from a shared platform [25]. An example is today's automotive manufacturing, where customers can configure their favorable variant from a basic type of vehicle by selecting different variations. The referred shared platform is built from core assets common to all variants,

[5]Bizagi Workflow Engine – `https://www.bizagi.com/`
[6]Camunda Workflow Engine – `https://camunda.org/`

which is then reused in the production. Groups of products that share specific variations are bundled to a product line or product family. *VM* is the task of modeling and managing the commonalities and variabilities of product lines. In *SPLE* commonalities are software features, which are needed across several software products and hence build a platform from which various combinations of software can be constructed and configured for specific customer needs or a particular market segment.

The introduction of *SPLs* can induce remarkable benefits for software engineering companies [65], but also raises the complexity and thus the effort of development and management of software. Bosch [9] and Hallsteinsen et al. [25] carved out some issues of present software, like the lack of variability, that need to be addressed to make *SPLE* successful in the long run. Therefore, several methods and models were developed to enable appropriate *VM* in software engineering.

Czarnecki et al. [18] stated that the most relevant approaches in *VM* are *Feature Modeling (FM)* and *Decision Modeling (DM)*, which more and more converge towards each other over time. *FM*, introduced by Kang et al. [34], concentrates on the commonalities and variability of features that software products provide. *DM* was introduced by McCabe et al. [39] and aims at providing models that act as a basis for the decisions that need to be taken when configuring a product. Most present techniques in *Feature- and Decision Modeling* were influenced by these works. Nevertheless, also other approaches exist like, for example, the *Common Variability Language (CVL)* [45] which was specified by the *OMG* as a domain-independent language for defining and resolving variability. However, as other techniques are marginally used and supported by tools or, as *CVL*, are not entirely specified, this chapter will focus on approaches of *Feature- and Decision Modeling*.

Before *Feature- and Decision Modeling* are explained in the next two sections, a few essential criteria are described, that serve as a basis for the selection of the specific approach for this thesis. A presumption of this thesis is, that for the *Customization and Configuration Process (CCP)* process and, furthermore, the linking of software components to engineering processes, software from existing use cases is utilized. This software is continuously evolving, and new features are added on a regular basis, which then either belong to the commonalities or the variabilities of the *SPL*. The first criterion is that the selected approach should enable the developer of the variability model to relatively easy extract parts of the model from existing components or their source code. Second, as the software components advance, the efforts to adapt the variability model to reflect the changes should be as low as possible. Third, the selected approach should be formal enough, to enable the generation of a machine-readable file from the variability model to further process it. Finally, existing tool support for the approach is desirable, to create and manipulate the variability model in a simple way.

2.3.1 Feature Modeling

In 1990 Kang et al. [34] published their work *Feature-Oriented Domain Analysis (FODA)* which influenced most of today's *FM* approaches [18, 35]. In their technical report, they

discuss a method to reveal and model commonalities in software systems related amongst each other, by utilizing *the process of identifying, collecting, organizing, and representing the relevant information in a domain based on the study of existing systems.* The goal of the approach is the *identification of prominent or distinctive features of software systems in a domain.* A domain, by their definition, is *a set of current and future applications which share a set of common capabilities and data.* Beyond that, the authors specified features as *user-visible aspects or characteristics of the domain.*

Based on prior research, the authors identified three core tasks necessary for the process of the domain analysis: a) a *context analysis* that defines the limits of the domain to be analyzed; b) the *domain modeling* that outlines the issues of the domain covered by the software applications; c) an *architecture modeling* that depicts the solution architecture to the issues of the domain. A major output of the domain modeling phase is the *feature model* that results from a *feature analysis.* The feature analysis aims at gathering and structuring the capabilities of the applications of a domain relevant for end-users in a concise model. Therefore, it classifies the features in commonalities (mandatory features) and variabilities, which can be differentiated into alternative and optional features.

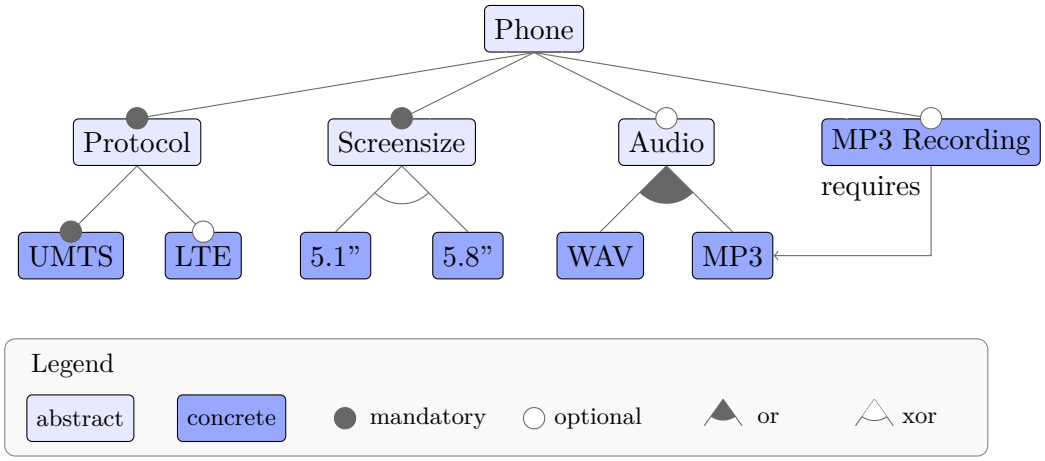

Figure 2.8: Feature model in tree notation - adapted from [18]

An example of such a feature model is shown in Figure 2.8, which is slightly adapted from [18] and takes several of the enhancements from the original *FODA* approach into account. The figure exemplifies a feature model of a product line of mobile phones. The graphical notation depicts features as rectangles with rounded corners and connects them with lines to sub-features to build a feature tree. Features in a feature model can either be defined *abstract* or *concrete.* Abstract features group sub-features regarding aggregation and generalization. Aggregation resembles a *consists-of* relationship as in ‘a car consists of an engine and four wheels’ (as well as other parts). Generalization equals a *is-a* relationship as in ‘a snow tire is a special type of tire’. The notation does not distinguish between abstract and concrete features symbolically, however, in the figure they are shown in slightly different colors. A phone in the feature model has the mandatory

abstract features *Protocol* and *Screensize*, which are marked with a black circle on top. The concrete protocol feature *UMTS* is mandatory and thus always included in a phone. Additionally, the optional protocol feature *LTE* can be selected. Optional features in the feature diagram are indicated by a bordered circle on top of the feature. Considering the screen size, either a *5.1″* or *5.8″* inch screen can be chosen, which corresponds to an exclusive disjunction. Such a logical XOR is displayed by a bordered arc that spans over the connections of the features. For the optional abstract feature *Audio*, one or more of the concrete options *WAV* and *MP3* can be selected. These OR variants are represented by a black arc that spans over the connections of the features. However, if the optional concrete feature *MP3 Recording* is desired, then it also requires the feature *MP3* to be present for the variant of the phone. A requirement connection is indicated by an arrow from a feature to the required feature. Often such constraints are kept in an additional section of the diagram to avoid clutter.

With the described notation, the aspects of a *SPL* can be concisely visualized in a tree format. Nevertheless, feature models of real-world software tend to get large and, furthermore, grow in size and complexity over time. This complexity makes tool support for handling and managing those models inevitable.

FeatureIDE

The *FeatureIDE* is an extensible, open-source software tool, based on the Eclipse IDE, that supports the concept of *Feature-Oriented Software Development (FOSD)* respectively *Feature-Oriented Programming (FOP)* [36, 67]. *FOP* is an approach that treats features as the primary building blocks of software respectively products of software development and allows a compositional combination of these features to applications [55]. This composition is achieved through selection and parameterization of the features in order to create a software system. The framework supports several *FOP* languages and paradigms, like Antenna[7], an annotation-based preprocessor for Java, or FeatureHouse[8], which is a language-independent approach, that derives concrete classes via superimposition or three-way-merge of source files written in different programming languages.

Another aspect of the *FeatureIDE*, especially relevant for this thesis, is the option to create feature models in a graphical editor. Similar to the feature model shown in Figure 2.8, the user can construct the features as well as define their hierarchy. Furthermore, constraints can be formulated in a simple language that supports a selected set of logical operations. The editor calculates the validity of the model based on the feature attributes and the expressed constraints and displays possible issues. If, for example, several contradictory constraints were specified, that prevent a feature to be selected, the editor will mark the feature as well as the involved constraints and display an error message. The resulting models are stored in an XML-based format, which makes them machine-readable and easy to exchange. A drawback of the format is, that no schema definition of the underlying XML exists, which makes it harder to interpret by other tools.

[7] Feature Preprocessor – `http://antenna.sourceforge.net/`
[8] Java Feature Extension – `http://www.fosd.de/fh`

In addition to the export, the *FeatureIDE* provides the possibility to create 'product' configurations based on a specific feature model in the, so called, *ConfigurationEditor* [53]. This editor predetermines a configuration process for product derivation and guides the user through the necessary steps to create a product variant. The application validates the configurations on the fly, according to their correctness related to the feature model, its dependencies, and constraints. This just-in-time validation means that the editor immediately responds to invalid configurations and communicates them to the user.

To summarize, *FM* provides methods to reflect the commonalities and variabilities of a *SPL* in a concise tree-based model, which is capable of representing the dependencies and constraints amongst the features. Considering the criteria mentioned above, the following conclusions can be made. Although not explicitly formalized in the work of Kang et al. [34], a formal specification for the feature model can be derived. For existing software, the feature model needs to be generated in an initial analysis. Depending on the complexity of the software and the system knowledge, the modeler has, the effort for this analysis can vary sharply. The feature model needs to be readjusted in parallel with the changes introduced to the corresponding software. The effort for this task, as mentioned before, also depends on the overall complexity of the software, however, if consequently done it should be within calculable limits. With the *FeatureIDE* a tool is provided, that adequately supports the creation, adaptation, and management of the feature models and its possible configurations of a *SPL*. Furthermore, the *FeatureIDE* provides an XML exchange format, which enables other tools to handle the feature models.

2.3.2 Decision Modeling

The basic concepts of *DM* were introduced in 1993 by McCabe et al. [39] in the Synthesis methodology. The methodology enables project engineers to understand and model *similarities and variations* in software systems, to later *exploit those similarities to eliminate redundant work*. Therefore, the approach provides the means for building and maintaining a decision basis, which is used by project engineers to satisfy their customer's needs by merely *answering the questions that are left open because of variations* during software derivation and configuration.

The approach defines a process, that is based the following core principals, that are utilized in different of its steps: a) the *formalization of a domain* as product families, that significantly share common aspects, but vary in parts among instances; b) *system building* which, with the application of this approach, should be reduced to the selection and configuration of decisions, that represent variations points in products; c) and the *reuse of software* by selection or adaptation of components and their configuration.

This primary process distinguishes between *Application Engineering* and *Domain Engineering* as its foundational sub-processes. *Application Engineering* is defined, by the authors, as the standardized task of producing and delivering software to customers and corresponds to the before mentioned *system building*. This includes a description of the requirements and of the system to be built, as well as the selected engineering decisions,

that result from the structured resolution of a variation questionnaire. This questionnaire is backed by a decision model, which itself is built in the *Domain Engineering* phase, for the specific system or product family. From the decision model and the selected engineering decisions, the work products for the final product are generated and configured. *Domain Engineering* corresponds to the *formalization of the domain* and is the iterative activity of managing and implementing work products of a product family. This process also includes the *Domain Specification* which, as output, produces a *Decision Model* that supports the *Application Engineering* task. The *Decision Model* represents the requirements and engineering decisions, as well as the logical relationships among them, that need to be considered to construct an application. In this way, the model determines the variety of instances in the specific system domain by *reusing software components*.

Name	Description	Range	Cardinality	Relevancy/Constraints
LTE	Is LTE supported?	true | false		
Screensize	How big is the screen?	5.1" | 5.8"		
Audio	Which audio types are supported?	WAV | MP3	1:2	
Recording	Is MP3 recording supported?	true | false		requires Audio.MP3

Table 2.1: Decision model as table (based on and adapted from [18] and [61])

The definition of the initial decision model, to represent variability in software, in the Synthesis methodology acts as a basis for most of the *DM* approaches. An example of a decision model in table notation, which uses the proposed adaptations by Schmid and John [61], is shown in Table 2.1. The adaptations make the decision model more comprehensive, regarding the contained information, and notation independent, by defining decision variables which are referenced at the variation points using decision evaluation primitives. The example in Table 2.1 represents the same variabilities of an exemplary phone product family as the feature model in Figure 2.8. A decision has a *Name*, which acts as a unique identifier, and a *Description*, that is formulated as a question to be used in the questionnaire. Furthermore, each decision defines a *Range* of values that it can hold. These values can be enumerations such as `true`/`false` or object references, like *Audio.MP3*, but also ranges of values, like `1`...`5`. The *Cardinality* indicates how many of the values a decision can hold. Finally, the *Relevancy* and the *Constraints* define whether a decision is relevant in the current configuration and which logical relationships the decision has to other decisions. In the example, the constraint *requires Audio.MP3* indicates, similar to the exemplary feature model, that the decision *Recording* needs the decision *Audio.MP3* to be selected.

The building and management of a decision model, especially of the logical relationships and constraints among the decisions, but also the automated creation of a variation

questionnaire from the model, as well as the derivation of the work products, requires sophisticated tool support to make the approach usable.

DOPLER Meta-Tool

In Dhungana et al. [19] the authors present the *Decision-Oriented Product Line Engineering for effective Reuse (DOPLER)* meta-tool, that supports a flexible and extensible variability modeling approach based on *DM*. In their work, they conduct a case study with several industry partners, to prove the applicability of their tool and the underlying concepts in different industrial areas. The implemented tool, which is based on the Eclipse platform, enables a guided product derivation and automated variant configuration in a specific domain.

The decision model, used for product derivation, is defined in DoplerVML, a variability modeling language proposed by in the working group of the authors. DoplerVML allows modeling the problem space as decision model as well as modeling the solution space with the help of, so called, *asset models*, which are collections of functionally and structurally grouped *assets*. *Assets* are models of artifacts, specified for a particular domain, which provide a solution fragment for the creation of a product instance. An example would be a *service* that covers functionally for the calculation of specific problems and provides an URL as attribute where it can be reached. The modeling approach provides extended flexibility, as, depending on the requirements of the domain, the assets can be precisely defined in the granularity needed. For example, today's software architectures range from classical client-server solutions to microservice architectures and even serverless computing, which makes it impossible to fit the solution fragments in a single rigid model. The asset model and the decision model of the domain are linked together according to the core meta-model of DoplerVML, which allows multiple decisions to be included in specific assets. In this way, product instances can be derived from the assets, which are selected and configured via a guided questionnaire.

The *DOPLER* meta-tool is feasible of providing the necessary features to create and manage a variability model, that can be used to link software variants to engineering processes. However, the *DOPLER* meta-tool is neither publicly available nor open-source. Nevertheless, according to the authors, it is possible to retrieve a research license for tool evaluation and development.

To sum up *DM*, the approach provides concepts to model variabilities of product lines utilizing decisions that need to be made to derive valid product variants and their configurations. In contrast to *FM*, which also models the commonalities, *DM* concentrates on the variabilities between the product instances. The most common notation for a decision model is table based and was introduced in the Synthesis methodology [39]. However, the notation and the underlying model was adapted and extended over time in other works [61, 19]. In regards to the criteria defined previously, it can be noticed that the concepts of *DM* are well formalized in various works. Likewise to the *FM* approach, the initial effort to derive a decision model from existing software depends on

the complexity. The same holds true for the adaptations of the software described in the model, but it can be stated that an additional change to the decision model results in a relatively small effort. Compared to the feature model the efforts of model creation and maintenance are quite similar. A defined exchange format is not known to the author, however, is does not seem very complicated to develop such a format from the notations available.

CHAPTER 3

Research Issues

Chapter 1 framed this thesis in the context of *production systems engineering (PSE)* and their heterogeneous tool environments. The need for tool integration platforms, due to the issues arising in such complex environments, was motivated in Chapter 1 and discussed in detail in Chapter 2. Moreover, in Chapter 1 the *Engineering Service Bus (EngSB)* approach, which allows to seamlessly integrate engineering tools and processes for improved cooperation among engineering disciplines, was briefly introduced and further presented in the related work (see Section 2.1).

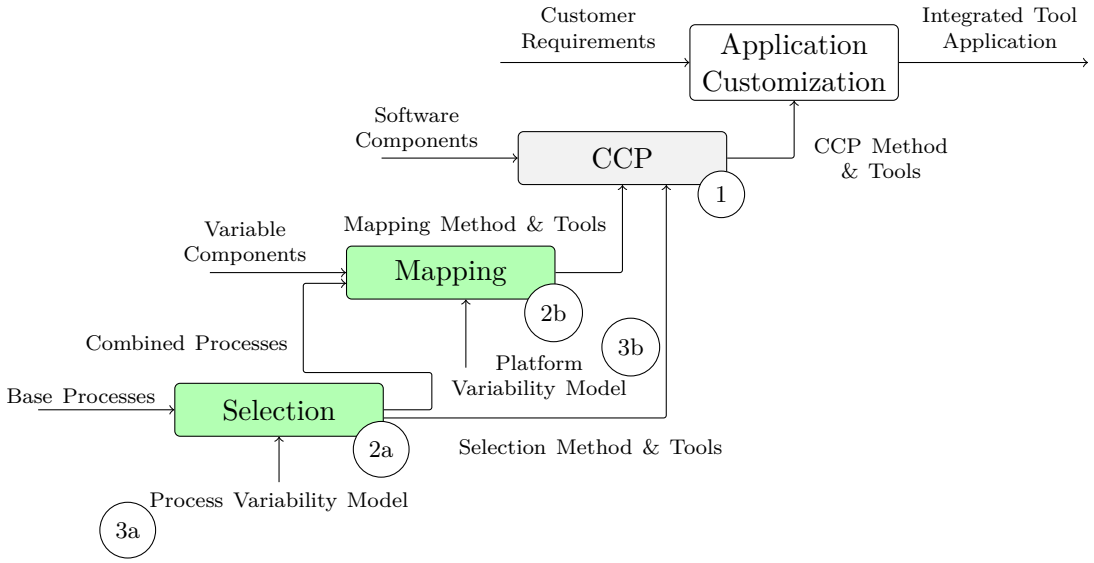

Figure 3.1: IDEF0 of the science contributions of the thesis

As stated in the introduction, the *EngSB* needs tailoring to the client's requirements in a *Customization and Configuration Process (CCP)* to be successfully implemented in

engineering companies. The specific problems of the *CCP* were outlined, which result from the tedious, most of the time manual, configuration and the missing quality assurance due to a lack of automation. A primary aim of the thesis is to provide a better quality of the resulting customized applications and save efforts at the same time. Therefore, the need for *Model Driven Software Engineering (MDSE)* approaches to support the *CCP* by generating parts of customized components or configuration, was motivated.

Besides the *EngSB* tool integration approach, in Chapter 2 related work was presented, that covers methods and notations to model processes, primarily in the business area. These findings can similarly be used to express workflows from the engineering domain. Furthermore, the two most common concepts used to handle variability in software systems were described in the chapter.

Due to missing approaches that handle the variability of software components and processes at the same time, the thesis concentrates on the selection of engineering processes and their mapping to software components. These two activities of the *CCP* result in process variants that are linked to component or service variants and can be executed in workflow engines that run on the tool integration platform.

From the focus of the work and the issues of the *CCP* raised in Section 1.2, but also the approaches that were discussed in the related word, the research questions and evaluation criteria for the proposed solution are defined in this chapter.

Figure 3.1 shows those tasks of the *CCP* that are within the focus of the thesis and the contributions the thesis provides in an IDEF0 diagram. An IDEF0 diagram is composed of a set of functional blocks that represent actions as well as data and object flows depicted by arrows. In this case, the relevant tasks that need to be fulfilled to create a customized integrated tool application instance and the flow of information between them are displayed. On the left side of a functional block inputs, like customer requirements, for the tasks enter. On the right side of a functional block outputs from the specific task exit. These outputs can then be used in other functional blocks. For example, the processes selected and combined exit the selection task and are used as input in the mapping task. Mechanisms or resources, like methods or models, enter a functional block from the bottom and provide the means to fulfill a task. For instance, the selection task takes the *base processes* as input and transforms them with the means of a *process variability model* to a set of *combined processes* which serve as input for the next task. The following research issues will be motivated according to this image in combination with Figure 1.1 from Chapter 1.

3.1 Research Issues

Based on related work of Chapter 2 and the problem description of Section 1.2 we identified a set of research issues, which are discussed in the sections below.

RI-1: Improvements of Tool Integration Platform Customizations

At the moment, the *CCP* needs to be performed by experts - the application integrators, because not only an intense knowledge of the available platform components and their capabilities is required, but also profound insights in the engineering processes that are adapted and implemented. Moreover, the experts need to know, how a specific set of components matches a possible set of activities in the engineering processes considering their dependencies amongst each other. The complexity of the *CCP* and the current manual approach often result in a high error rate within the process and a low initial quality of the resulting application. These difficulties lead to increased efforts of expensive application integrators, as they need to repeat tasks of the process in case of faults and are thus blocked for other work. This complexity also represents a steep entry barrier for new application integrators, especially when the platform continuously evolves.

The effects of the mentioned issues worsen when the risk of an expert shortage is considered. If no resources are available to perform customization tasks, the process is delayed or even worse suspended. This can be the case, if, for example, engineers need to step in for application integrators to acquire the knowledge of the process. The probability of such events should not be underestimated. Therefore, the major question that needs to be raised is:

> *To what extent can the traditional CCP for tool integration platforms and the activities of* engineering process selection*,* engineering process variant to software component mapping *and* engineering process configuration *be improved to enable the generation of fully configured process instances?*

RI-1 aims at the improvement of a) the efficiency of the *CCP* due to a lower level of complexity and effort; and b) due to a higher level of automation; as well as c) the effectiveness of quality assurance by applying possible checks during the process.

From the experience of *Continuous Integration and Deployment (CID)*, the automation of a build process can mean a significant improvement in quality and in saving working effort [21, 24]. *CID* saves time and working effort because the process runs without the constant attendance of a user, and the user is only asked for input to certain steps. *CID* also improves quality because checks can be run during the process that ensures the quality and correctness, or respond to issues that occur. An increased level of automation and additional entry points for quality assurance checks, go hand in hand with a faster customization process, less manual efforts of experts and makes errors visible earlier. Furthermore, a decreased level of complexity fosters the transparency of the process, which lowers the entry barrier for non-experts.

An essential aspect for the improved *CCP* is the acceptance amongst stakeholders involved. Therefore, tools supporting the tasks of the process should be investigated. As it is likely, that only parts of the activities can be backed by tools, it should be examined what other mechanisms exist to help engineers. So a question in this context is: *Which already*

existing tools support the approach of modeling and linking process variants, and which tools are needed on top of this toolset?

From the answers gathered to *RI-1* it should be possible to propose a solution for an improved *CCP* and implement a prototype of the method that can be evaluated considering its performance. The contributions of *RI-1* should lead to an overall improvement of the *CCP* as indicated by the node *1* in Figure 1.1 and 3.1.

RI-2: Mapping of Engineering Process variants to Software variants

The activity of process mapping will be described in detail in Section 6.4, however, the general approach works as follows. An application integrator takes a process description and compares it to process definitions from former existing projects. These process definitions either exist as a formal description or more often are encapsulated in some orchestration component in the code itself. The process definition is then copied and adapted to the needs of a specific customer. When the newly adapted process definition fits the use case, the application integrator has either candidate for possible software components in mind or needs to find such components and map them to the process tasks. Afterwards, the application integrator can link the process tasks and the software components either by writing the calls to the components into the process definition or the orchestration class. It can be imagined, that these activities are as well tedious an error-prone. Therefore, the aspects of the following question need to be investigated:

> *How can variants of engineering processes be semi-automatically selected, mapped to software component variants and configured with service calls?*

The question investigates how the three single steps of the process selection activity can be changed in a way that allows them to be better structured or even formalized to support the application integrator and, also, how tool support can be provided that executes parts of these steps automatically.

To do that, another question needs to be answered: *Which parts of the improved process can be automated and which tasks still need to be manually done?* This means it needs to be investigated which steps need human knowledge or intervention and which parts can be run automatically with additional quality checks that assure their correctness.

The prototype for the solution approach should consider the findings of the raised questions, to automate the process. From several runs of the prototype, data should be collected, contribute to *RI-1* using the evaluation criteria, developed below in Section 3.2. Furthermore, the answers to this research question should contribute to the improvement of the *CCP*. In the overview figure (Figure 1.1) and the IDEF0 diagram (Figure 3.1, the improved engineering process selection is shown as contribution *2a*, and the mapping of the selected processes to reasonable software components is shown as contribution *2b*.

RI-3: Variability Modeling for Engineering Processes and Services

To allow the process selection, the creation of process instances, and, in the further course, the mapping of these instances to specific software components, a representation, is needed, that models variations of the concepts common to both, the processes and the components. Therefore, the question that can be raised is:

> *How can concepts of Variability Modeling (VM) be utilized for modeling engineering processes and software systems to allow mapping between process variants and software variants?*

In Section 1.2, *VM* were assumed promising for the generation of customized parts for integrated engineering applications and especially the creation of configured process variants. Section 2.3 introduced common approaches of *VM*, that enable the representation of commonalities and variabilities in software. It should, therefore, be proved that some models and methods from *VM* can be exploited to develop an improved *CCP*, which can support stakeholders to select process variants and link them to services in the application. The research question aims at the identification of concepts from *VM* and their investigation according to the needs of the use case. Moreover, the question aims at the creation of a process, that is superior to the actual *CCP*, which is manually done.

Before the investigation of which concepts of *VM* can be utilized, and the question above can be answered, it needs to be clear where in the platform variability points can be found. For example, during the *CCP* several components can be selected to fulfill customer requirements, that provide very similar features and often just differ in how they do something. For instance, the data of the engineering tools need to be saved as a model in a model repository, so there might be a *ModelRepository* service. However, often models reference files that also need to be stored in combination. So there might be a *UnifiedRepositroy* service that saves the models in a store optimized for models and the files in a different store optimized for files. So before helpful concepts of *VM* can be identified, the following question needs to be addressed: *Which components, that are used to assemble an integrated engineering application, can vary between application instances and which distinct aspects does a component have?*.

After the components, that provide points of variability, are identified, concepts from the field of *VM* can be investigated, whether they can be utilized for the type of variability and how the variability can be modeled with this concept. When the best fitting models and methods of *VM* were selected, there might remain a gap between their intended use and how they can be applied for the approach to be proposed. For example, the grammar or the semantics of a concept might need to be slightly changed or updated, to serve the goal of the thesis approach. To reply to this issue, the following must be asked: *To what extent do models and methods identified need to be adapted to support an improved CCP?*.

From the answers to the question and its subquestions, it should be possible to propose methods that allow creating models of the process base as well as the software base. These

models should the be used to allow the selection of process variants and their mapping to software component variants. In other words, the first part of the the research question should be a *Process Variability Model* (*3a* in the supporting figures) that contributes to the *Selection* (*2a*). The second contribution of the research question should be a *Platform Variability Model* (*3b* in the figures) that supports the *Mapping* (*2b*).

3.2 Evaluation Criteria

To respond to the research issues mentioned above and *RI-1* in particular, and to evaluate the proposed method through the implemented prototype, *Key Performance Indicators (KPIs)* for the *CCP* must be searched and defined. These *KPIs* must be equally applicable for both, the traditional and the improved process.

Before indicators can be defined two variants of customization that differ must be distinguished. The first type is the customization of the platform to a 'standardized' integration application, that can be used by several companies of the same engineering sector or group for well-known engineering projects. The second sort is the customization for one specific customer with special requirements, that uses the integrated application for the kind of projects typical in the customer's field or frequently changing engineering tools. The first task is done once and can be used by multiple clients; the second is done once for a single client. So the cost of customization in the first case decreases with every additional client, while it stays the same in the latter approach. The scope in this work is set to the 'one customer' cost model, as it would be hard to include and predict the marginal cost saving of a new customer of the 'standardized' application.

KPI-1 - Complexity

The *CCP* is a complicated process, that might take several months of work and needs multiple iterations. Undoubtedly, a process that is less complex is also less error-prone and creates results that be better understand. Because of the existing complexity and the aim to reduce it, the measurement of complexity on top of the *CCP* is a valuable magnitude. However, the measurement of process complexity is a relatively new field of research and a difficult task to conduct [14].

In the literature, different metrics for the complexity of processes exist, which are mostly developed for business processes in business process management. As the *CCP* itself can be seen as a business process in the broader sense, this thesis utilizes several of the proposed methods. One indicator that is emphasized as an indicator for process complexity is the perceived *psychological or cognitive complexity* [15, 12]. However, as this is exceptionally hard to measure and can only be achieved with the additional help of surveys, its measurement is out of the scope of this thesis. Two complexity measurement approaches that seem more promising and can be applied on the *CCP* are the *activity complexity* and the *control flow complexity* [15, 13]. From these approaches, the used *KPI* are derived.

KI-1.1 is defined as the *number of manual activities in the CCP*, while *KI-1.2* is defined as the *number of manual activities in the CCP in relation to the number of automated activities*. The last indicator can also be seen as the *degree of automation* of the process. As no formal or strict description for the traditional approach exists, it is also difficult to apply these measures. However, the approximate sequence of activities in the traditional approach will be outlined in this work and used to apply these measures. *KI-1.1* is expected to be lower in the improved process compared to the traditional approach. *KI-1.2* is expected to be far lower in the improved process compared to the traditional approach, as the process should mainly be an automated process.

For future research, the investigation of *control flow complexity* as an indicator for the improved *CCP* is recommended.

KPI-2 - Effort

In the traditional *CCP*, an application integrator or release manager manually performs the tasks that are associated with the configuration, while software developers implement additional software components and business process experts design the engineering processes. As the assignments of the application integrators are manually done, they are time-consuming which blocks away from the integrators from other tasks in the meantime. So an essential magnitude of the *CCP* is the effort it takes to generate a customized integrated application. However, the effort must be evaluated for at least the following two different scenarios.

An important indicator, for the performance of the method, is defined as the *effort in working hours, invested in the CCP for a customized tool integration platform* with a set of stable requirements [*KI-2.1*]. Still, the effort for the set up of the prototype needs to be taken into account to the overall effort to make these two approaches comparable. Because the initial effort is taken into account, the indicator needs to be measured for each of the two approaches and two subsequent customizations respectively to show the possible cost-saving effect on the second customization.

The second indicator concerns the additional functionality and features realized over time, which enables further application variants to be created. Such developments mean, the application integrator needs to include these components into the set of solution elements for the *CCP*. Therefore, the *effort in steps for a customization after a marginal change of the solution set* is defined as *KPI* [*KI-2.2*].

KPI-3 - Quality

As already noted, the quality of the resulting approach is hard to ensure. Especially *RI-1* covers the question how the quality can be increased. However, one issue mentioned are missing tests that cover the customized components and their interaction correctly.

From this question raised, two further indicators are defined, that is applied to the traditional as well as the improved process to evaluate their performance. The first

one is the *number of activities in the CCP, where quality and test mechanisms are implemented out-of-the-box* [*KI-3.1*]. The second is defined as the *number of quality and test mechanisms that can easily be implemented and automatically be run* in the process [*KI-3.2*].

It is expected, that both values are lower for the traditional approach and higher for the improved *CCP*. This result means that the platform provider can run more test and quality assurance measures and, furthermore, that they can also be automated.

CHAPTER 4

Methodology

This chapter will outline the research approach and the methodology applied in the present work.

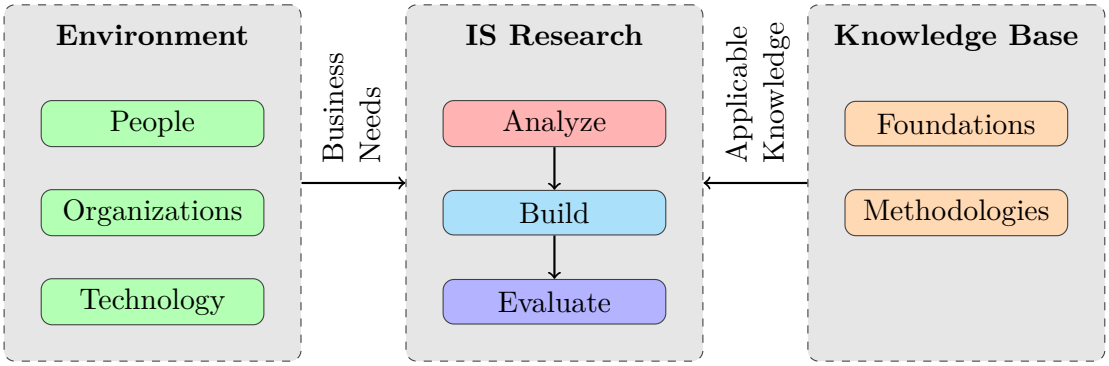

Figure 4.1: Methodology framework (based on [27])

This thesis follows the constructive *Design Research* approach proposed by March and Smith [38] and Hevner et al. [27]. The two approaches and their concepts will be discussed concisely, and the activities carried out in this work will be associated with the presented concepts.

According to March and Smith [38] two types of research approaches for information systems can be distinguished. *Natural Research*, on the one hand, is often utilized in traditional research, like physics or biology, and consists of the two basic activities *Theorize* and *Justify*. This means researchers in their work create theories or hypotheses, that are then justified or invalidated. *Design Research*, on the other hand, is a constructive approach seeking an applied solution for a problem. *Design Research* consists of the two basic activities *Build*, which creates the artifacts *Constructs*, *Models*, *Methods* as well as *Instantiations*, and *Evaluate*. While *Building* is defined as the process of designing

and implementing the before mentioned artifacts, *Evaluating* is defined as the process of testing whether the artifact is applicable for the intended use.

In addition to the original approach, Hevner et al. [27] presented a framework taking the influential forces on the research process into account. Their framework seeks to provide the means to understand and successfully execute *Information Systems Research* by combining *Design Research* and *Behavioral Research*, which to them is fundamental for effective research. They, therefore, introduce a set of guidelines that aim at helping researchers to understand the requirements to conduct *Design Research* properly.

To explain how *Design Research* was utilized in this thesis, Figure 4.1 shows a simplified and slightly adapted version of the *Information Systems Research Framework* [27]. On the left side of the figure, the research *Environment* with its stakeholders is depicted, which demands research relevance and exposes real-world problems that lead to research topic and questions. In the case of this work, industrial partners motivated the need for an effective customization and configuration process in the context of engineering tool integration platforms. Furthermore, they provided insights into engineering processes and needs of engineering stakeholders. On the right side, the *Knowledge Base* is shown, that provides foundations, like theories, instruments, and constructs, as well as methodologies to conduct the research activities of building and evaluating artifacts. Furthermore, the scientific communities that mind for the knowledge base are mandated to test the research produced. In the central section of Figure 4.1 the research process itself is depicted. For the thesis, the figure was extended by the *Analyze* activity. This phase includes the activities performed before the research artifacts could be designed and implemented in the build phase. This means, for example, the task to investigate whether the research topic was relevant both for the industrial partners as well as the addressed scientific communities.

In the following paragraphs the three research activities *Analyze*, *Build* and *Evaluate* are underpinned with the tasks executed in this thesis.

Analyze activity

In Chapter 1 we present a motivational example and a brief problem description, which was adopted from relevant industrial partners and own experience in the CDL-Flex research laboratory. After a more precise description of the problem and the research topic, the research issues are defined in Chapter 3. Moreover, through a requirements analysis with industrial partners, further needs are weaved into the research questions. To detect concepts and methods, a careful literature study is conducted, of relevant research areas, which seem promising to contribute to a solution. The results are presented in Chapter 2 with a particular focus on the fields of *Variability Modeling (VM)* and *Business Process Management (BPM)*. Additionally, the literature study thoroughly investigates the relevance of the research topic in the addressed scientific communities. A selection of concepts is then matched to the research topic. Furthermore, a tool study is carried out and discussed in Chapter 2 to collect and evaluate tools that can be exploited for an

applied solution prototype. Also in this activity, the evaluation criteria are formulated, and key performance indicators are defined, to enable an adequate evaluation.

To be able to evaluate the approach proposed by the thesis, a set of metrics for configuration efforts needs to be developed. Moser et al. [43] defined performance indicators that measure the effort for an integration solution. Broy [10] specified further aspects and metrics that indicate and measure the quality of large software systems. As a starting point, the aforementioned metrics can be considered. The specific set of metrics to be used is part of the investigation in the thesis. When the set of metrics is defined, it should be applied to the resulting prototype to determine the efforts of the (semi)automated customization of the platform.

Build activity

In the build phase, that follows the analysis phase, first a specific use case is formulated and explained in the Chapter 5, that serves as underlying sample for the solution approach and the evaluation. Second, applying and extending the selected constructs and methods from *VM* and *BPM* as well as providing own ideas, a theoretic method is developed that provides a solution for the research topic. In order to test the proposed method, the selected concepts and the formulated approach are used to build models on top the defined use case. Furthermore, the findings of the theoretical solution will then be implemented in a simple software prototype that provides tool support and automation features. Finally in this phase, the models are used in the prototype, to test its basic applicability for the research problem.

Evaluate activity

In order to evaluate the solution approach for the method the implemented prototype is evaluated. Therefore, the prototype is assessed according to the metrics defined in Chapter 6. A discussion of the approach and the evaluation in Chapter 8 will conclude the evaluation.

CHAPTER 5

Use Case

Chapter 3 raised the research questions that this thesis examines and defined *Key Performance Indicators (KPIs)* that are used as metrics to evaluate the solution approach and its impact on the *Customization and Configuration Process (CCP)*. Alongside, Chapter 4 introduced *Design Research* which is used throughout this work as methodology and defines three stages to conduct research in the area of Computer Science.

To apply the methodology on the one hand, but also to underpin the answers to the research questions, on the other hand, the utilization of a specific use case seems helpful, if not necessary. In Chapter 1 a very simplistic example of an engineering process was outlined, that frequently emerges in *production systems engineering (PSE)*. This chapter details on the use case, according to with its necessary processes and the underlying tool integration platform and its services. The use case is referred over the course of the thesis to explain the solution concepts and develop the prototype. Moreover, the example is used to test the methods and models as well as evaluate them on a real-world instance.

5.1 Round-trip Engineering

As described in Section 1.1 engineering companies establish various processes to perform their daily work. Although these processes differ in many activities, several similar and recurring tasks can be identified. As mentioned, Chapter 1 provided a central example, where engineers of different domains manipulated planning artifacts and stored afterward. Activities that support the planning of production systems, like the one described, can be found in most engineering companies. This similarity among customers is an advantage for platform providers, as they can design processes which can be used as blueprints for process definitions in other engineering companies.

A primary process, which is widespread in many engineering companies, is the planning of production sites. This process requires several engineers from different disciplines to

work together on shared parts of a production system. However, the models of these disciplines are rarely stored in the same artifacts. Instead, these models are distributed over various source files, that can usually only be accessed, viewed and manipulated with specific engineering tools.

In contrast, integrated tool applications act as single-source providers, as the data of the engineering models can be accessed by different engineering disciplines over standardized interfaces, although the data is persisted in a repository that transparently propagates changes across engineering disciplines. This kind of engineering is called *round-trip engineering (RTE)* [20] and facilitates parallel planning as well as better traceability of changes in the engineering project.

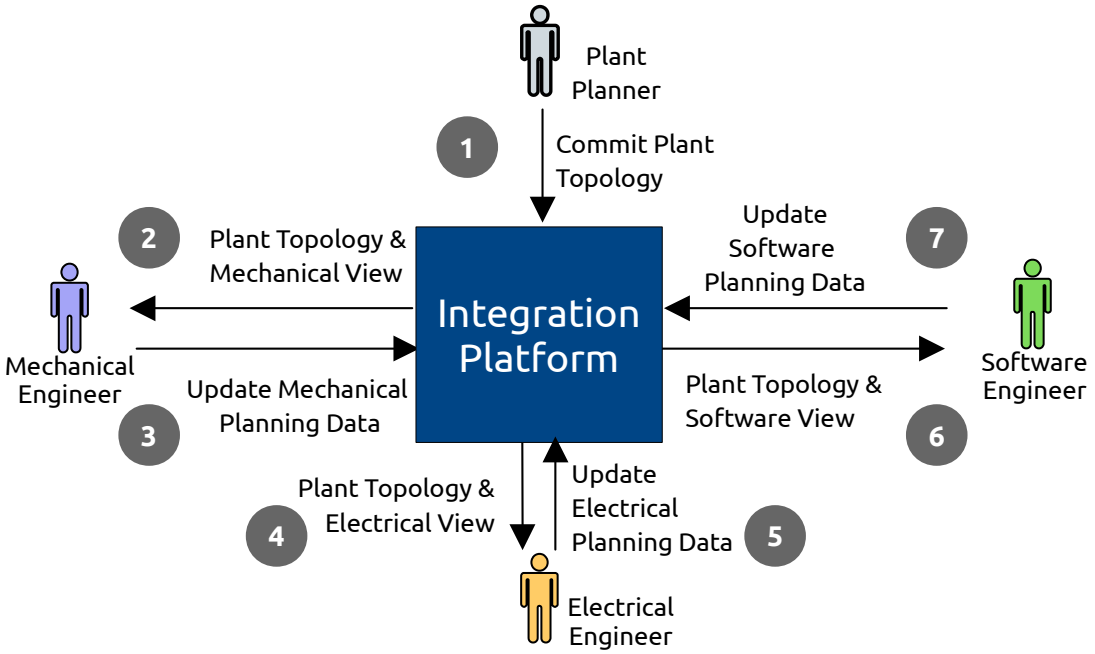

Figure 5.1: Round-trip engineering process

The workflow of *RTE* in *PSE* is displayed in Figure 5.1 and explained in the following. In a first step, a *plant planner* plans and commits the *plant topology* to the integrated engineering application. This plant topology is then checked out, in Step 2, by a *mechanical engineer*, who adds the mechanical view and respective components. Therefore, the engineer changes and updates several engineering artifacts, and commits them, in Step 3, to the integrated engineering application after completion. Fourth, the *electrical engineer* checks out the planning data that exists so far and enriches the plans with the electrical data. These changes are then committed in Step to the integrated engineering application. Finally, in Steps 6 and 7, the *software engineer* checks out the data, implements the necessary components that control the hardware components, and again commits it to the shared repository.

As illustrated, the *RTE* process starts with the planning of the general plant topology. From this point on, the process is not strictly sequential. On the one hand, multiple engineers of the same discipline work on different parts of the production systems. On the other hand, as soon as engineers from different disciplines can use pieces of these plans, they start and perform their tasks.

While *RTE* was explained in general as a frequently emerging process in *PSE* in this section, the next section details on an engineering process and use case from an industry partner of the CDL-Flex, that implemented *RTE* which the *Engineering Service Bus (EngSB)* and *AutomationML Hub (AML.hub)* approach.

5.2 Round-trip Engineering in Practice

Over the course of the CDL-Flex laboratory, several engineering processes of industry partners were investigated, analyzed and optimized for the use in *PSE*. From these investigations, a set of standard processes which are part of *RTE* were identified. Depending on the engineering company but also depending on their projects, these processes showed different variations in their tasks and sequence flows.

In this section, first, the traditional approach of a particular industry partner is introduced. Afterwards, the newly implemented *RTE* process is described and prepared for the development of the thesis's approach and the reply to the research issues. In this way, the variations in the process can be exploited to develop parts of the solution's methods.

5.2.1 Traditional engineering approach

The industry partner is an engineering company that operates in the field of hydropower plant planning and construction. The engineering process of the company follows a *waterfall model* with slight adaptations, taking customer change requests into account and feeding back information from the construction site of the plant. A waterfall model follows a relatively sequential order of steps to design and implement a system. Usually, a requirements phase is followed by a design, implementation and commissioning stage, which is then followed by the operation and maintenance phase. A drawback of the waterfall model is the often missing feedback loop. The lack of feedback means changes that are necessary after the requirements phase or issues found in the commissioning phase are difficult to implement due to re-planning chain and thus very costly.

A typical project, planned by this company, starts with a requirements phase where the engineering company gathers the needs of their customer towards the hydro power plant. Then, in a first engineering phase, the general structure, like buildings and floors, of the power plant is planned and placeholders for units, like turbines and generators, are placed. These plans also represent the topology of the project which uses the strictly structured '*Kraftwerk-Kennzeichensystem (KKS)*' [70]. According to the *KKS* structure mechanical engineers in their engineering tools place and configure the before mentioned units.

The relevant data is then exported to Excel sheets, which are used by electrical engineers to build up the electrical plans in their tools. However, as the export to Excel loses the structural information and encodes the *KKS* data in a field, the *KKS* data needs to be decoded by the electrical engineer from the export. Once again, from these tools, the data is exported to Excel sheets and sent to the customer for inspection and approval. If adaptations are needed, the customer changes the values in the Excel sheets and sends them back to the engineering company. One of the engineers then has to painstakingly examine the sheets for change requests and transfer them to the respective tools by importing parts of the sheets into the tools. In this process, the engineering has to take care, which the customer did not change data that is excepted from adaptations. It needs not to be mentioned that this process is highly error-prone and generates high efforts, but also makes changes between disciplines hard to trace.

To make the situation worse, engineers need to travel to the constructions sites on a regular basis to observe and monitor the construction progress. If something in the facility changes on the construction site, the plans need to be readjusted there offline. As most of the time the engineers on the construction site are not connected with their computers to the engineering company systems, the changes introduced need to be integrated into the plans at the company, after the engineers return.

5.2.2 Round-trip Engineering with the *AML.hub*

From the prior description, it can unquestionably be understood, that the engineering processes and the data exchange policies of the partnering engineering company would benefit from re-engineering and the implementation of an integrated tool application. In a project, the CDL-Flex proposed several improvements to lead the company towards an integrated tool environment and developed a customized solution.

As basic tool integration platform the *AML.hub*, presented in Section 2.1, was used. The *AML.hub*, as mentioned before, is based on the *Open Engineering Service Bus (OESB)* and allows components and services to be loaded during runtime, which makes it very flexible as a re-configuration can be done at any point. Software components or services in the *OESB* are implemented as *OSGi (OSGi)* bundles. *OSGi* bundles are Java packages with an explicitly specified life cycle, that can be dynamically loaded into an *OSGi* environment. The services in the *OESB* always conform to a specific interface, which allows exchanging a service for other services even during the runtime of the integrated too application. To support this concept, Maven[1] which is a build automation and software project management tool, is used. Maven, besides other features, has the capability of resolving necessary dependencies during build time and configuring the components according to a setting that is read from an external file.

An essential component of the *AML.hub* is the unified repository - the *Engineering Database* (see also Figure 2.2), that holds and revisions the data of the engineering tools with the help of *AutomationML (AML)* models. The *AML* models as well as other

[1] Maven Build Tool – `https://maven.apache.org/`

models can be registered in the *AML.hub* and are realized with the *Eclipse Modeling Framework (EMF)*[2]. The *EMF* allows to specify models in a structured format named *XML Metadata Interchange (XMI)* and, furthermore, provides sophisticated tools to create and manipulate these models.

For the project, an integrated data model for hydropower plant engineering was developed in *AML* and implemented in *EMF*, which included the domain knowledge of the involved engineering disciplines. This data model included the shared concepts of the engineering disciplines in a unified perspective, but also provided discipline-specific views on the data. For example, in the common perspective an engineering unit has the *KKS* number and a *description* as attributes, while the electrical view on the integrated model, additionally contains the *component number* that indicates the location in the control cabinet.

Since the engineering tools used by the company did not provide interfaces that could be utilized to connect them to the *AML.hub* directly, their proprietary export formats and functions had to be exploited. The export functions that were insignificantly configurable within the tools generated *Comma Separated Value (CSV)* files that contained the tool data of the specific engineering project. To process these *CSV* files in the *AML.hub* a couple of software components were needed. First, a further model, that represented the *CSV* format was defined and realized as simple table model in *EMF*. This table model maps every cell in a *CSV* file to a cell in the table model. Second, a transformation engine was implemented, that was capable of transforming values of one model to the values of another model. Finally, several user interface components were needed, to let the users interact with the integrated tool application. The *AML.hub*, therefore, provides a framework, that enables the dynamic inclusion of web user interface components. With the support of this framework, a visual editor called *Transformation Editor* was created, that provided the means to define and test transformation rules and save those rules as transformers. These transformers are interpreted by the transformation engine to map between the models.s Furthermore, a component for the import of the tool data into the *AML.hub* and another component for the export of the discipline-specific data view to a tool import format was implemented. Additional user interface components that provided further features, like management capabilities as the project progress evaluation, were designed to support the employees of the engineering company.

Before it is possible to implement the *RTE* workflow in the integrated tool application, it needs to be investigated on an abstract level and break up into its central activities. From Figure 5.1 reveals, that *RTE* consists of certain activities that recur for each engineering discipline. One of these activities is the import of the engineering tool data into the *AML.hub*. The other activity is the export of the data views from the *AML.hub* to the engineering tools. While the export process is simple, because the data just needs to be transformed into the discipline-specific export format and then downloaded, the import of the data into the integrated tool application is more complicated. The next section, therefore, describes the import process (indicated by steps *1*, *3*, *5* & *7* in Figure 5.1) in detail.

[2]Eclipse Modeling Framework – `https://projects.eclipse.org/projects/modeling.emf`

5.3 Signal Change Management Process

From the explanation of *RTE* in theory and practice in the previous two sections, it becomes clear, that several software components and a detailed process is needed, to support engineers in committing their data to the integrated tool application.

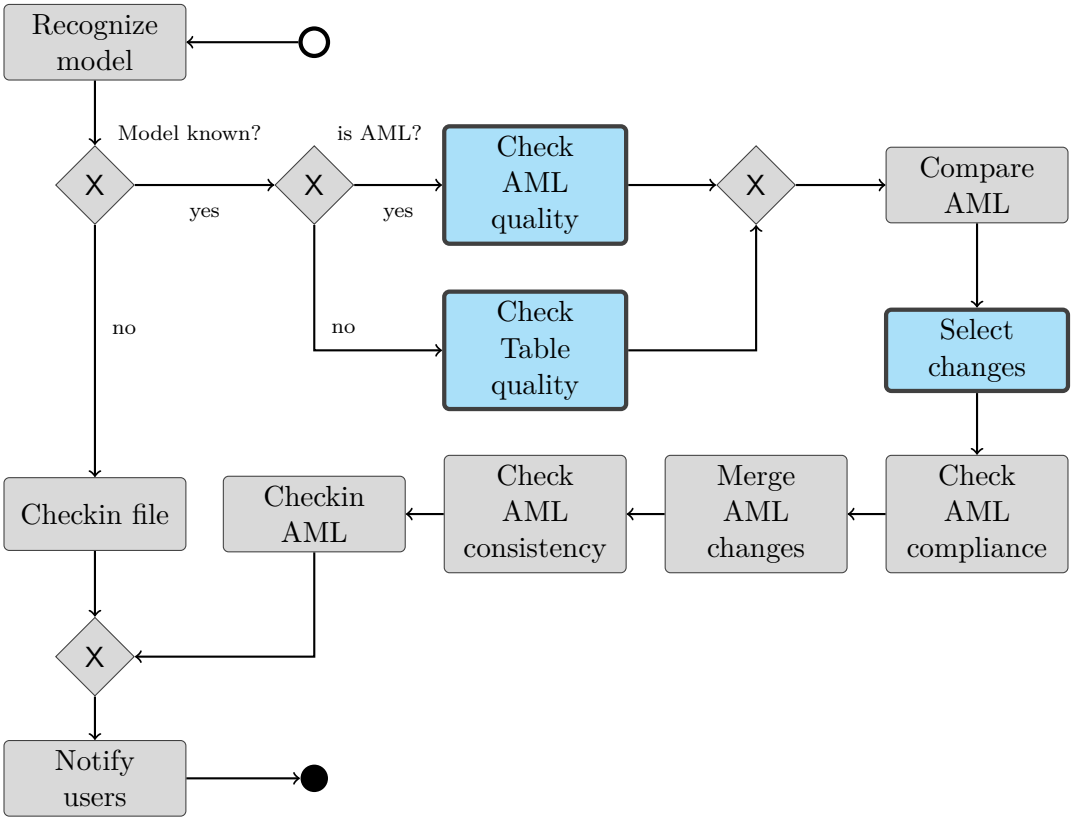

Figure 5.2: Signal change management process (based on [74])

Such a structured process for 'checking in' data to the integrated engineering application, was identified as *Signal Change Management Process (SCMP)* and explained by Winkler et al. [74]. Signals, in this case, are single planning units such as sensors or valves that are planned into the production system. Figure 5.2 displays a slightly adapted process, that was used in the use case of the engineering company described in the prior chapter as leading process.

The process consists of the following steps. The user checks in a *CSV* file from the supported engineering tools. The *Recognize model* task, calls a service that tries to identify whether the file checked in is one of the known file formats. This task can either be done by a service that interprets the file ending of the file or service that peeks into the file and guessed the format with the help of the file header.

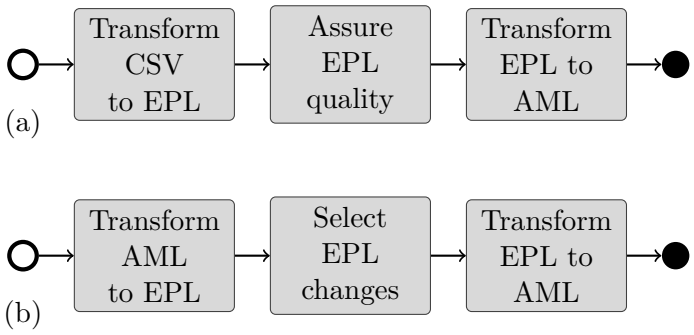

Figure 5.3: Quality assurance and signal selection subsprocess for EPL tool

If the service identifies no model at all, the *Checkin file* task checks in the file into the repository, as it is assumed that the file is an artifact, which does not have a model representation such an electrical plan in a PDF file format. If the service identifies a model, it depends on the model type which action comes next. If the file is already a *AML* file a service checks the quality of the file in the *Check AML quality* step. This means that, for example, the consistency of the *KKS* numbers is checked as this number should be unique in the production system to be planned.

In contrast to the activities before, this activity does not call a service directly, but a sub-process that was defined elsewhere instead. While direct calls to software components are depicted in grey, sub-process calls in Figure 5.2 are represented in a blue color. In case a table model is detected, similar to the *AML* quality check, the table is checked for its quality in the *Check Table quality* activity. However, the called sub-process differs in two things. First, depending on the engineering tools used in the project another set of quality checks is used and second, also depending on the kind of table models two different sub-processes are used. This is because data from OPM, in contrast to EPL, needs to be enriched before it can be translated to the internally used *AML* model. The quality assurance sub-processes for each tool and their activities can be seen in Figure 5.3 *(a)*, respectively Figure 5.4 *(a)*.

After these steps were performed, in the *Compare AML* step the resulting *AML* data is compared to the version in the *Engineering Database* for changes. In the *Select changes* step, depending on the model that is checked in, different sub-processes can be called. In principle, the sub-process does the following. If changes, like additional, deleted or modified signals are present in the checked in data, they are either run through a service, that selects the signals that are later committed to the database or displays the changes in a user interface for a manual selection by an engineer. However, before the changes can be displayed, they might need to be transformed so that they make sense to the engineer. The sub-processes for the EPL and OPM tool can be seen in Figure 5.3 *(b)* and Figure 5.4 *(b)*.

The next remaining steps stay the same for all model and consist of several further quality

checks, an action that merges the changed data into the existing data and commits these changes to the *Engineering Database*. At the end of the process, the relevant users are notified of the changes in the project. Although the steps in this process stay the same, except for the sub-process calls, different services can be used from project to project for the single steps. Also, it is also possible that the sequence flow itself changes, but this example does not cover that case.

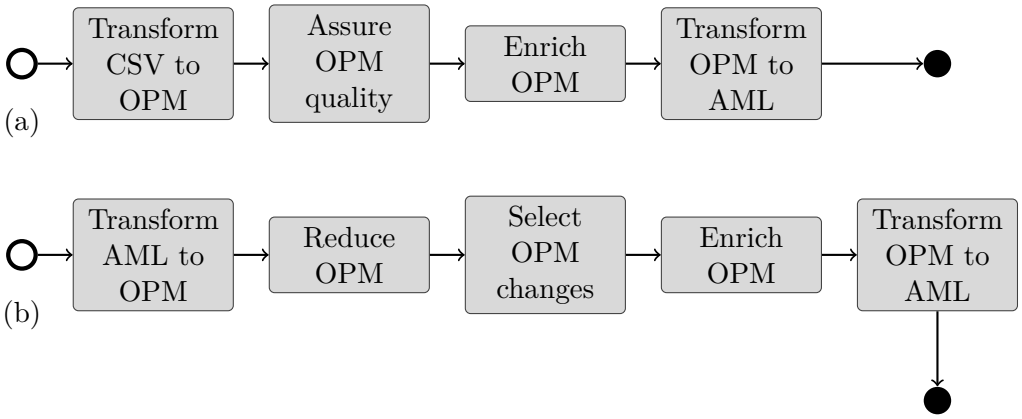

Figure 5.4: Quality assurance and signal selection sub-process for OPM tool

For the thesis, this *SCMP* will be taken as the lead process, which means it will be used to create different process variations and link the activities of these variations to variants of software components.

5.4 Signal Change Management Process Customizations

The last section described the *SCMP*, which allows committing the discipline-specific tool data into the integrated tool application, in detail with its different possible sub-processes. This section explains the *CCP* for this specific process and the tasks that need to be performed to generate a valid process which is linked to the software services that a particular customer requires. It is assumed here, that the processes used are formally modeled in a process notation and that a pretty similar model of the process was already created beforehand or is reused from another customization project.

First, the application integrator selects the process model and copies it to the workspace which he uses for this customization project. For the use case described, he would select and copy the model of the *SCMP* that is depicted in Figure 5.1. Afterwards, the integrator needs to adapt the control flows and add or delete activities of this main process to meet the customer requirements. This practice is called *clone-and-own* [23] as the relevant artifacts are first copied from another project and then owned by the new project.

Second, the integrator investigates all remaining call activities in the main process and checks which sub-processes are called from these activities. He then needs to search where the models of these sub-processes are located. These models can, for example, be located directly in the process modeling software on his computer, but also in a distributed storage on of company doing the customization. When the models are found, the integrator copies the sub-process models into his workspace. Regarding the use case, the sub-processes of the three call activities *Check AML quality*, *Check Table quality* and *Select changes* would be effected.

However, as the sub-process models come from another project, they are present in a configuration that conforms to the requirements of another customer. This means there might be parts of the desired process missing or several, in principle separated activities, are merged into a combined sub-process. The integrator now, in a manual third step needs to either combine the selected sub-processes with additional sub-processes, relevant for the new customer, or he has to cut out parts of the available process.

The fourth step of the integrator is now to go through all tasks of the adapted main process as well as the adapted sub-processes and investigate which service fits the specific task. Therefore, he either has to have the available services, that support a task in the process, in mind or he accesses some documentation database within the company, that describes the available software services and how they match different tasks and requirements. A particular extensive task is to go through the different available services and choose the one that fits the requirements of the customer best.

The next customization activity the integrator performs is the time-consuming work to resolve the dependencies and constraints of the services that are planned for the processes. For example, *Service X* might require another software component here called *Component A* to work correctly. However, *Service Y* which the integrator intends to use for a specific task, might not be compatible with *Component A* and should thus not be used in the customized application. A solution might, for instance, be to use *Component B* instead, which is compatible with *Service X* and *Service Y*. To perform such a dependency resolution requires a deep insight into the platform and their services and is hardly possible without proper tool support.

Finally, the revised processes and the corresponding services that are compatible amongst each other need to be wired together. Therefore, the application integrator configures each task in the process and assigns a specific service call to the task.

From the description above it becomes even more evident, that the customization of the processes and their mapping to relevant services is done in time-consuming work, done by expert users.

5.5 Summary

To summarize, this chapter introduced *RTE*, which is a common practice in *PSE*, in Section 5.1. *RTE* describes a sequence of activities that should be considered in artifact

exchange to foster an improved planning process in the industry. The approach has even more impact if a standard exchange format is used among the engineering disciplines which can, for example, be achieved with *AML*.

The second section, explains how engineering is done by an industry partner of the CDL-Flex and the steps that were taken to support *RTE* with a customized integrated tool application based on the *AML.hub*. This use case and the customized solution will be used to investigate possible solution approaches for the selection and mapping of process variants to software component variants.

Section 5.3, presented a lead process, that was found by the investigation of several industrial partners and explained in Winkler et al. [74]. The section then detailed on a specific variant of the process, that was utilized for an industry partner.

Finally, Section 5.4 described how the *CCP* for a specific process, like the one introduced in this chapter, looks like and discloses the issues that lead to the amount of effort that needs to be investigated in the process. From the description follows that the complexity of the *CCP* grows massively with the number of available software service candidates for a task and their dependencies and constraints amongst each other.

CHAPTER 6

Solution Approach

The previous chapter discussed *round-trip engineering (RTE)* a best practice approach regarding the utilization of standardized exchange formats or integrated tool applications to support the planning process in *production systems engineering (PSE)*. Furthermore, the chapter showed how industrial partners currently apply planning processes and explained how *RTE* can be implemented in engineering companies using the *Engineering Service Bus (EngSB)* approach, which enables a seamless tool integration by adopting concepts of the *Enterprise Service Bus (ESB)*. However, for a successful implementation the *EngSB* requires tailoring to the specific engineering company's needs and especially the customization and configuration of the engineering processes. Therefore, in the last section of Chapter 5 the *signal change management process* as a real-world example was introduced, and some of its variabilities examined. It is an important part of *RTE* and was investigated by Winkler et al. [74] in detail.

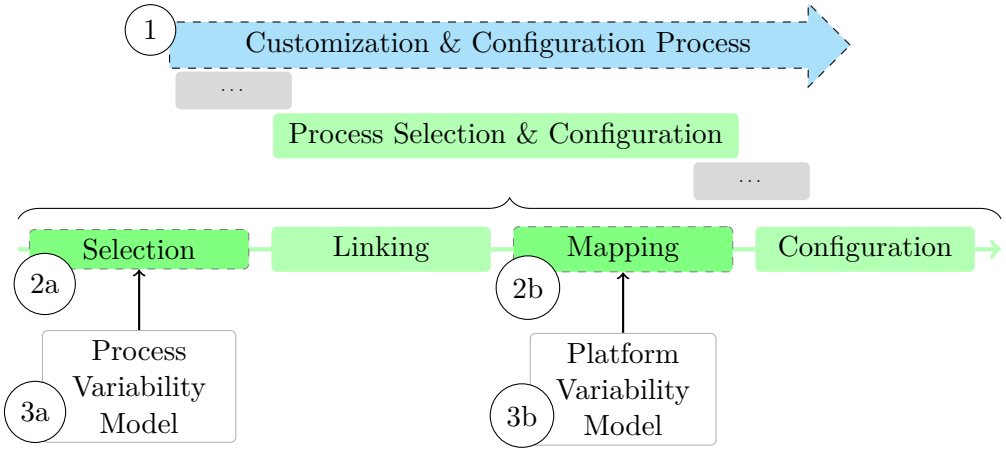

Figure 6.1: Contributions to the solution approach

This chapter proposes a solution approach, to the tedious and error-prone process of engineering process selection as well as the mapping of the process activities to software services, which are called during the execution of the engineering process within the integrated tool application during runtime. Figure 6.1 shows the single contributions of the sections to the overall solution application and how they are related to each other. The structure of the chapter follows the structure of the figure and, as the contributions build on each other, are presented from bottom to top.

First, Section 6.1.2 discusses how variability can be modeled in engineering processes to cover the different variations of engineering processes that engineering companies experience (contribution *3a* in the figure). Section 6.2 after that, picks up the contribution from the section before and defines how engineering process variants depending on each other can be effectively selected during the *Customization and Configuration Process (CCP)* to create valid processes. This contribution is indicated with *2a* in Figure 6.1. Third, Section 6.3 (see *3b* in the figure) characterizes how the variability of software services in tool integration platforms, using the *EngSB* approach, is represented in a model that can be utilized in the next step. Section 6.4, marked by *2b* in the figure, then proposes a method to map the engineering process variants resulting from the selection of software service variants that are represented by the platform variability model. Finally, in Section 6.5 – see *1* in Figure 6.1 – the contributions from the prior sections are summarized, and their impact on the *CCP* is explained when they are put together in a process with proper tool support, which helps to automate central parts of the otherwise manual work.

6.1 Process Variability Model

Besides the possibility to integrate engineering tools via standardized procedures, a valuable feature of an integrated tool application is the visualization and formalization of engineering processes, which are often implicitly utilized by engineering companies. This visualization often also leads to the application of more efficient engineering processes in the companies. Primarily, there are two ways of including engineering processes, found in the requirements phase of the *CCP*, into the integrated tool application. Either an engineering process resembles another process, that was already used in another customization project, or the specific process needs to be designed and implemented for the customer. In the latter case, the resulting process might also be reused in other projects, which have very similar processes. However, it is likely that the processes differ in specific points, which results in different variations of the basic process. This technique is called *clone-and-own* as mentioned in the description of the use case in Chapter 5. One issue with this approach is that it results in various process variants that are each included in a single project but do not build a structured catalog of independently reusable processes. To make the step of engineering process selection and adaptation in the *CCP* more efficient and allow the efficient reuse of processes, a format that enables variants of engineering processes to be described, needs to be developed, which then builds a catalog of possible process instances.

This section aims to develop a very simple *process variability model*, which helps to quickly select specific process variants while being flexible enough to restructure the process if necessary. First, the methods and models for business process modeling from the related work are recapped and evaluated, to select an approach that seems most reasonable for the use case. Second, the types of variability that occur in processes and that are covered by the proposed solution approach are explained. Finally, a simple variability model for processes is defined, that supports a semi-automated selection of processes and their sub-processes.

This section contributes to defining variability on a process level, which is needed for the improvement of the *CCP*. The contribution to the big picture of the solution approach is indicated as *3a* in Figure 1.1.

6.1.1 Process Modeling Method

In Section 2.2 different methods and models of business process modeling were discussed with their advantages and limitations. In the following paragraph, those approaches will be investigated considering their suitability for the expression of engineering processes and their integration to the *AutomationML Hub (AML.hub)* that is used as tool integration platform.

The principles of *Petri nets*, the first method described in the related work, are easy to understand due to their simple logic and their limited concepts, yet *Petri nets* remain very expressive. However, even small *Petri nets* and their logical flows can be hard to comprehend and tend to dramatically grow in complexity when bigger problems are modeled with this method. This lack of understandability and the poor integration possibilities into existing software are a downside for the utilization of *Petri nets* in engineering projects.

The other three approaches presented in Section 2.2 provide symbols that work on higher semantic levels than the ones of Petri nets. For example, *Event-driven Process Chains (EPCs)* allow logic connector like OR and XOR and *Activity diagrams* allow forks, joins and decision nodes. Therefore, they are easier to understand and can better picture complex processes. From the possibilities to model the most relevant workflow patterns [69], *EPCs*, *Activity diagrams* and *Business Process Model and Notation (BPMN)* are equally expressive [73, 37]. Modeling tools such as the Camunda Modeler mentioned before, which allow users to design and specify models in one of the technologies in closer consideration, are publicly available.

However, as *EPCs* do not have a standardized exchange format and no runtime tools, that are openly available, it would be hard to integrate this approach into the *AML.hub* as tool integration platform. *Activity diagrams*, with *Unified Modeling Language (UML)*, provide a standardized exchange format and can also be executed with special frameworks as noted in Section 2.2. With the partially open-source workflow engine Camunda that can also be run in an *OSGi (OSGi)* container, *BPMN* provides the best integration possibilities compared to the other technologies.

BPMN, at a closer look, provides several other advantages. In the description of the use case in Chapter 5, it was mentioned that, depending on the project, different software components could be used for a specific activity in the engineering process. The selection of the changes that are committed to the *Engineering Database* can, for example, either be performed manually by an engineer or automatically by a service, that has a rule set defining the selection criteria. In *BPMN* it can be defined which type an activity has. In this case, it would be possible to define a task as *human task*, to permits links to software components that provide a user interface. Furthermore, Petritsch [54] already sketched some processes, that describe parts of the *change management process* mentioned in Section 5.3. Because of these advantages, *BPMN* was selected as technology to define the engineering processes and based on the model that allows describing the variability of the processes.

6.1.2 Types of Engineering Process Variability

The last section reasoned why *BPMN* is used in the solution approach to model variability in engineering processes. However, before the variability of the engineering processes can be modeled explicitly for specific processes, the types in which *BPMN* processes themselves can vary need to be identified, which is done in this section. Before the variability mechanisms are described in the next paragraphs, the semantics of *BPMN* from Section 6.1.2 are recalled shortly for a better understanding. In *BPMN*, three different types of activities exist in a process. Tasks are atomic activities and execute a specific behavior, sub-processes are self-contained processes, and call activities allow the calling of re-usable tasks as well as sub-processes [44].

In Schnieders [62] and Schnieders and Puhlmann [63] the authors describe several variability mechanisms that frequently occur in *business processes* and outline approaches how these mechanisms can be implemented using the concepts of object-oriented programming in Java. The approaches are not applied directly in this work, as the engineering processes should be designed as models in a formal notation instead of program source code. The most relevant variability mechanisms presented by the authors and adapted in this work are a) encapsulation of varying sub-processes and b) addition, replacement, and omission of single elements.

The *encapsulation of varying sub-processes* is described in the context of the use case in Section 5.3 where, for instance, the *Check AML quality* activity can call a sub-process that might vary in a customization. For the *Select changes* activity even several sub-processes exist that might need to be combined to a single sub-process to be correctly called from the main process.

The *addition, replacement, and omission of single elements* needs to be differentiated for this work. The omission of single elements in regards to the thesis are interpreted as the addition of individual tasks, call activities or control flow elements in the main process or a sub-process. In the example, this means, that a certain task or branch, such as the *Checkin file* branch in Figure 5.3 is omitted. Addition of single elements is very similar to

the described omission. It means that an additional activity is introduced in the process. However, this task should then be considered for all upcoming customizations in the first place and, if not needed in a customization, omitted during the *CCP*. In both cases, the application integrator needs to take care, that the output models of the activity performed first corresponds to the input model of the activity performed later and, that those necessary parameters are forwarded to the next activity. The replacement of single elements is understood, in the context of this thesis, as the replacement of a software service, that is called in a process task, with an alternative software service from the available modules. Considering the use case, this would mean that, for instance, the comparison of the data in the *Compare AML* activity could be executed by two different services and that one particular of them, is selected during the customization.

To summarize, the variability mechanisms considered for the process variability model are a) *process variability*, in terms of varying tasks and sub-processes called from a *BPMN* call activity (see also Figure 2.7) and b) a varying sequence of process activities as well as their addition or omission, and furthermore c) the *service variability*, which is realized by the call of different software service variants in process tasks.

6.1.3 Variability Model for *BPMN* Processes

Considering the types of variability identified in the prior section, this section aims at the definition of a variability model for *BPMN* process models. Each of the next paragraphs is related to one variability mechanism and explains the solution approach to model variability within this type. To support the assignment of *BPMN* elements to activities in a later phase, then the initial modeling phase, the *BPMN* model needs to be enriched in a way that enables the allocation of the right elements. In the use case, for example, the sub-processes which can be used for the *Select changes* call activity must later be found automatically by a configuration software. Therefore, concepts of *BPMN* for the specific elements are exploited to perform the enrichment. The definition of the variability models should consider that these models need to support the following two tasks in the *CCP*. First, the *selection* of processes and sub-processes as described in Section 6.2 and second, the *mapping* of specific process activities to software service instances as described in Section 6.4.

```
1 <bpmn:callActivity id="1" name="Select changes"
2                    calledElement="SelectChanges"/>
```

Listing 6.1: `calledElement` in a `callActivity`

The first variability mechanism of process variability covers *call activities*. In *BPMN* a call activity has an attribute `calledElement` which should reference a `CallableElement`, which is the abstract super class of all *BPMN* activities. Listing 6.1 shows the usage in a *BPMN* snippet for the before mentioned *Select changes* call activity. To enable sub-processes to be related to the call activity it is defined, that the `calledElement` attribute contains a `template` prefix which is used to retrieve corresponding processes.

For example the `calledElement` attribute in the activity *Select changes* is prefixed and results in `template:SelectChanges`. The relevant processes for this call activity are then located by definition in a folder named *SelectChanges-Templates*.

The changing sequence of activities in the variability model and the addition or omission of activities is only considered insofar, that it is assumed for this thesis that these operations can be performed as long as the output and input models of two sequential activities are compatible. This is somehow a limitation that will be discussed in Section 6.2 and is subject to future work.

```
1 <bpmn:serviceTask id="2" name="Compare AML"
2                   operationRef="AMLComparison">
```

Listing 6.2: `operationRef` in a `serviceTask`

The variability mechanism of process variability covers task elements in *BPMN* models. Section 2.2 remarks that task can have different task types, which are depicted in Figure 2.7. These task types can, following the *BPMN* semantics, be separated into three distinct groups. *User tasks*, *manual tasks* and *business rule tasks* are not connected to any automated component in a workflow engine. So, in this case, the implementation of the specific workflow engine has to decide how to handle those tasks. Usually, workflow engines provide components like web user interfaces that allow confirming the completion of such tasks of proceeding further with the process. For *business rule tasks* workflow engines often provide implementations that allow a ruleset to be defined, that is fired on the input parameters and then decides about the outcome of the task. *Script tasks* simply refer the inputs to external scripts, that handle these variables and according to the flow in the script. *Service tasks* and *message tasks* are in the main focus of this work. *BPMN* defines an attribute `operationRef` for these tasks, that reference a *BPMN* `Operation`. Such an `Operation` can be utilized in an interface that is implemented by a particular service. Listing 6.2 shows the attribute used for a service task. It is defined for the thesis that the value of the `operationRef` attribute is prefixed with an `abstract` keyword, to reference service implementations in a task. In case of the service task from Listing 6.2, the attribute value would be `abstract:AMLComparison` and the service implementations need to have a mapping according to this attribute value.

Using the use case from Chapter 5, the variability model will be explained for better comprehensibility. The *Signal Change Management Process (SCMP)*, shown in Figure 5.2, is first modeled with its activities with the help of a *BPMN* tool. Next, the sub-process variants that can be called in call activities are modeled in the same tool as isolated processes. Isolated in this context means, that each variant is defined in its model. For example, for the EPL and the OPM engineering tool, mentioned in the use case description, each variant of the sub-process is created in a different model instead of already combining them. Afterwards, the following enrichments are performed in the notations XML format. First, the `calledElement` attributes of the call activities in the main process model are set to a value that holds a unique name in the model and is

prefixed with a `template` prefix. Second, the `operationRef` attributes of the single tasks, in a further step, are also assigned with a unique name in the model but in this case, prefixed with a `abstract` string. Finally, in both cases, the possible variants that can be selected for the, at the moment placeholder variables, must be resolved and exchanged with the concrete value at a later point.

To summarize, two specific variability mechanisms are achieved by the exploitation and enrichment of the activity attributes defined in *BPMN*. In case of call activities, the `calledElement` attribute is utilized and prefixed with a `template` string to search all applicable sub-processes. In case of service and message tasks, the `operationRef` attribute is enriched with an `abstract` keyword to find corresponding service implementations later in the customization process. These definitions together build a very basic variability model for *BPMN* processes that are used to describe the engineering processes of companies in the domain of *PSE*. This variability model will contribute to the selection process of the *CCP* explained in the next section.

6.2 Engineering Process Selection

In Section 5.3 it was mentioned that during the course of the *Customization and Configuration Process (CCP)*, engineering processes are often selected and defined with an approach called *clone-and-own*. This approach results in various engineering process variants that are scattered over different projects. This dispersal of variants not only prevents the maintenance and improvement of several processes in a single step but also makes it harder to keep track of the different existing variants. However, the engineering processes can often be broken apart into smaller independent processes, which encapsulate a certain functionality and can effectively be reused in other processes of a higher order. Together these processes then build a variant catalog that can be used during the *CCP*. As call activities only support a single process to be called, the independent sub-processes need to be combined and enhanced with, for instance, additional sequence flows, to form a joint coordinated process.

The previous section contributed a variability convention for *Business Process Model and Notation (BPMN)* processes that allows the definition of process and sub-process templates and their dependency amongst each other. However, application integrators performing *CCP* to modify a tool integration platform for a specific customer, need support for the efficient selection of the engineering process templates and their sub-process templates to be able to use the full capacity of such a process catalog. In this section, the selection process, implemented in a software prototype that is introduced in Section 7.2, is explained in detail. Therefore, it is assumed that a process catalog with variants of engineering process templates and their sub-processes already exists or at least, that the sub-processes are defined in independent models.

As a first step of the 'engineering process selection', the application integrator selects a template of the main process, for example, a model of the *Signal Change Management Process (SCMP)*, from the catalog. This catalog can either be stored in a distributed file

system or in, a more sophisticated tooling environment, in some sort of repository that allows the search and resolution of 'artifact addresses'.

After this step, the selected process template is parsed for call activities and their `calledElement` attribute. Within this attribute, the `template` prefix is stripped away to read the unique name or an address where the templates of matching sub-processes are located. The selection tool then resolves the location of the sub-process templates and presents the sub-process variants to the application integrator. From these sub-processes, the integrator selects the ones, which are needed to fulfill the requirements of the engineering process. For instance, based on the use case from Section 5.3 the integrator can identify the need for a process, that allows checking the quality of data from the *EPL* and the *OPM* engineering tool. When the *CCP* support tool parses the main process, it would ask the integrator which sub-process templates should be used. From the different variants, he chooses the ones that cover the customer's requirements best. For example, he would select the relevant sub-process templates for *EPL* and *OPM* quality checks which are subsequently processed by the support tool.

In a next step the sub-processes, chosen in the prior action, need to be merged to a combined sub-process in a *BPMN* modeling tool. This step has to be done manually by the integrator at the moment, which is a limitation of the current solution approach and subject to future research. As an example, the involved sub-processes, like the ones shown in Figure 5.3 - *(a)* and Figure 5.4 - *(b)* need to be merged to share a common start and end event. Therefore, the engineer selects the proposed sub-process templates and the support tool copies them to a place were the modeling tool can access them so that the integrator can combine them.

Finally, the tool saves the combined process and assigns a name that is then written into the `calledElement` attribute of the main process. The name is resolved at runtime by the workflow engine to call the specific sub-process. The prepared main process and the combined sub-processes are then used in the mapping step to link them with the service variants that the tool integration platform provides.

Together with the contribution from the prior section, the introduced concepts allow the creation and effective selection of engineering process variants for the *CCP* in *BPMN*. With tool support, the approach can be further automated to assist the application engineer during the customization better. The next sections describe a platform variability model and a mapping approach that enable the application engineer to map the chosen engineering processes to specific software service variants.

6.3 Platform Variability Model

The last two sections introduced concepts that allow creating different variants of engineering processes in *BPMN* to build a catalog of predefined processes for the customization of tool integration platforms to integrated tool applications according to the needs of specific engineering companies. Furthermore, the sections proposed a solution to effectively select

and configure the process variants to later include them in the integrated tool application. To create processes that can be executed by a workflow engine, the activities in the processes need to be configured to call certain software services. As different services can be used in an activity to fulfill the task, it is needed that the service variants matching an activity are acutely identified and can be located by the engineer during the *CCP*. This section describes how a variability model of the service variants, that are implemented in the tool integration platform and its corresponding modules, can be defined.

6.3.1 Service Variability Modeling Method

In Section 2.3 two approaches were presented that allow the modeling of variability in software. The following paragraphs recap the approaches and examine which of them fits most, to represent the variability in a tool integration platform like the *Engineering Service Bus (EngSB)*.

Feature Modeling (FM) models the commonalities and variabilities of a specific domain as features, in which a domain is defined as *current and future applications* and a feature as a *user-visible aspect of the domain* [34]. For example, a feature can be the audio format a cellphone can understand, but can also be a software service that allows writing an email. Based on this concept several works developed different, but very similar kinds, of *FM*. Features in such a model can be mandatory, which means they need to be included in a specific entity of an application, or optional, which means they can be omitted. Furthermore, the features can exclude each other in the final application or they can run in parallel. Finally, features can also depend on other features, which means if, for example, feature B is needed in an application feature A which feature B depends on also needs to be included. Feature models have a graphical notation that is shown for an example in 2.8 and can be translated to a formal notation. The *FeatureIDE* [36] is a tool framework which allows users to create feature models and generate parts of the source code for further implementation. Additionally, the *FeatureIDE* enables a user to generate a configuration for a specific variant of the model and to afterward validate this configuration according to the constraints in the model.

Decision Modeling (DM) was introduced by McCabe et al. [39] in the Synthesis approach. The approach also aims at understanding the commonalities and variabilities of application domains, but in contrast to feature modeling, concentrates on the variations of the underlying domain [18] and builds entity configurations by *answering the questions that are left open because of variations*. Similar to *FM*, *DM* defines alternatives that can be chosen and which or how many of the alternatives can be selected. However, *DM* leaves out the parts that are mandatory in an application and, therefore, are similar to all application entities. The decisions from the model, that, for example, an application engineer has to make are extracted from the domain as questions and prepared for the user. The *DOPLER meta-tool* [19] is a framework for *DM* based on the *DoplerVML*, a modeling language that allows modeling the decision space and the solution space of a domain to link the decisions with the software artifacts.

Compared to each other, the two approaches are very similar in what they can achieve for modeling variability in software systems. However, *FM* provides some advantages considering the use case described in Chapter 5 and the existing implementations of the *EngSB*.

The software artifacts of the platform build a sharp hierarchy, which seems more natural to model in *FM*, as the concept of hierarchy is built into *FM* due to the *tree-like organization* of features [18]. In contrast to *DM* it also seems more feasible to the author, to extract the features for the feature model using, for example, software component names from the underlying software platform than to extract questions about what has to be fulfilled to produce a particular application variant. Furthermore, as already several implementations of the tool application platform exist, the author from the own experience of customizing such platforms, believes that it is essential for the application engineer also to reveal the commonalities of the platform. Especially in consideration of the adaptation of the model when new components and services are implemented, that need to be reflected by the model. As the *EngSB* projects *Open Engineering Service Bus (OESB)* and *AutomationML Hub (AML.hub)* are build with the software build tool Maven it is more likely that a semi-automated generation of the feature model, based on the dependencies between the components, is successful. Such a semi-automated approach is in the focus of future work. Finally, the existence of the *FeatureIDE* as open source tool to generate feature models and their configuration is a significant benefit in comparison to the *DOPLER meta-tool*, which is of limited availability for the public.

6.3.2 Variability in *EngSB* implementations

In the last section, the use of *FM* for the variability model of *EngSB* implementations was motivated. This section explains how variability is modeled in practice for the existing implementations of the *EngSB*, to support the proposed approach of mapping engineering process variants to software service variants. Accordingly, the *AML.hub* is used as a general example for the existing implementations of the *EngSB*. To recall, the *AML.hub* is a specific implementation of the *EngSB* approach that uses the industrial exchange format *AutomationML (AML)* as a central data model for the engineering domain.

As mentioned before, in Kang et al. [34] features were defined as *user-visible aspects of an application domain*. Czarnecki et al. [18] found that this definition was more and more softened up over time, to a broader concept that regards to properties of different types in a software system. For this thesis, the broader term of a feature is adopted and adjusted for the application of tool integration platforms based on the *EngSB* approach. Therefore, the term *feature* is as a result of this defined as:

> *A self-contained software component or software service providing functionality which adds significant additional value to the execution of a specific engineering task and is specified by a strict interface definition to other services.*

Using this definition, a feature is not limited to components that are directly visible to the user but also includes features like models of engineering tool data or services that run in the background of the platform like REST services.

Section 2.1 mentioned, that the *AML.hub* uses *OSGi (OSGi)* as technology to achieve a high level of modularization. The implemented software services and components of the tool integration platform are deployed and run in the *OSGi* container *Apache Karaf*. These services can be installed, started, stopped and even updated during runtime, which makes the approach very flexible and comparable to a microservice architecture. Apache Karaf, furthermore, provides a sophisticated communication and modularization concept, which allows several instances of the container to be run in parallel and to discover and call services across container borders.

In *OSGi* it is common practice to separate the definition of services from their implementations, using abstraction via *interfaces*, and deploy them independently. *Interfaces*, in software engineering, describes the abstract operations of an entity as 'method signature' without containing data or executable code. This concept is mainly used to abstract the function of software from the implementation. Consequently, the realization of an interface is called *implementation*. This means in *OSGi*, that an interface of a service is defined and realized, for example, as *Java interface*, see Listing 6.3 - line *1*, and compiled to an *OSGi bundle*. An *OSGi* bundle is a *JAVA archive (JAR)* with additional metadata that provide a symbolic name to discover the bundle and the exported interfaces, but also the requirements it has towards other bundles. A *JAR* itself is a *ZIP archive* which includes the compiled program files and a `META-INF/MANIFEST.MF` file that contains at least the Java version and the name of the main entry method, if it exists. At runtime, the interface bundle is then deployed to the container so that it can be discovered by the system components.

```
1  interface X { int returnAbsolutZero(); }
2
3  class Y implements X { int returnAbsolutZero() { return -273; } }
4
5  class Z {
6    private X x;
7    public Z() {
8      x = serviceLocator.getService(X.class, "filter:ranking=1");
9    }
10   public void doSomething() { x.returnAbsolutZero(); }
11 }
```

Listing 6.3: Interface, service provider and service consumer in *OSGi*

Software bundles that provide services implement a specific interface and its methods and export an interface reference. Additionally, service provider bundles can export several service properties, which make them discoverable and filterable within the container. Listing 6.3 shows such a simple service implementation in line *3*. For example, a service

`Y` exports an interface `X` and provides an attribute `ranking` with value `1`. Services that implement the interface can then be discovered by its interface class, and the resulting list can be narrowed down with a filter, for instance, `ranking=1`.

Service consumers, in their program code, use the interface reference to call certain functions of a service provider implementation. In lines *5 - 11*, Listing 6.3 shows class *Z* which consumes the service method `returnAbsolutZero` solely via the interface definition. At runtime, the service consumers then find the service implementations in the *OSGi* container by their interface class and the mentioned optional filters, which can be seen in line *8* of the listing. The advantage of this code is that the implementation of the service can be easily exchanged without changing any of the code.

So the most common way variability is realized in the *AML.hub* is via the concept of abstraction and inheritance using interfaces and their implementations. Another way is the dynamic parametrization of the software components. As mentioned *OSGi* allows the reconfiguration of bundles during runtime. *Apache Karaf*, therefore, provide functionality to change the attributes of services via a console interface and also a web interface. The initial values of the service attributes are defined at the compile time of the software bundle. This thesis focuses on the first variability mechanism to build a feature model of the platform services and modules that can be used during customization.

6.3.3 Feature Model for *EngSB* implementations

In the last section the meaning of the term *feature*, in the context of this thesis, was defined as a software component that provides additional value to the execution of a task. Furthermore, the *interface design pattern*, as it is used in *OSGi* to separate different service implementations and how it is used in *EngSB*, was explained. Picking up these concepts, this section describes how feature models for *EngSB* implementations can be built. To underpin the description, the example use case from Section 5.3 and specially the *signal change management* process depicted in Figure 5.2 is used.

In the further course, the feature model is constructed from the tool integration platform as described in the following paragraphs. Therefore, Figure 6.2 will be used to explain the assumptions and construction rules for the feature model of the tool integration platform. The figure shows a detail of the feature model of the tool integration platform services that are relevant feature candidates for the *signal change management*. The model derives from a common root for organizational reasons and to distinguish it from other feature models. In Figure 6.2 it is named *Checkin*.

Abstract features, shown in light blue in the figure, represent interface bundles, that serve as a functional definition of service implementations. Their name is derived from the bundle name for simplicity and must be unique in the model. For example, the model defines the abstract optional feature *Commit*, which can be seen in the middle of the figure. This feature represents the functional definition for the services that commit data to the integrated repository. Thum et al. [66] discuss several issues that come with the usage of abstract features. One issue is, for example, that the satisfaction

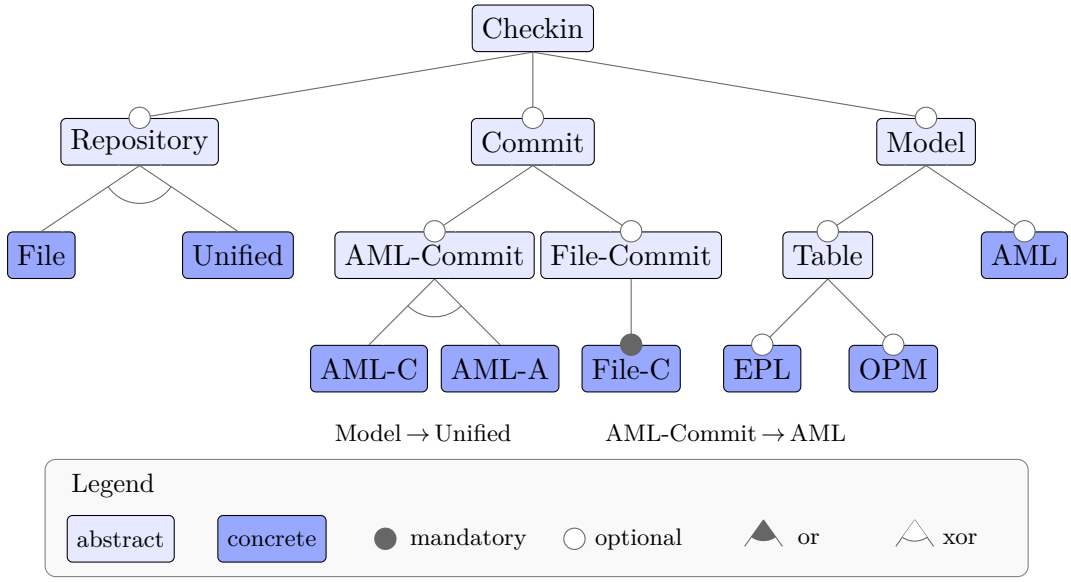

Figure 6.2: *Signal change management* detail of the platform feature model

constraint resolvers used today, can run into errors while evaluating the configuration for a particular feature model. The author is aware of the problems that come with the usage of abstract features, and tries to circumvent them by carefully creating the feature models for the tool integration platform. Nevertheless, it seemed the best way to use abstract features for interface bundles, as they have no functionality themselves. The branch of the *Commit* feature is optional as it can be omitted in the final version if, for example, the quality assurance activities are not needed at all in an engineering process. The two abstract features *AML-Commit* and *File-Commit* descend from the quality interface bundle and thus the respective feature. If the feature model gets too cluttered, abstract features might also be used to organize it farther. However, using abstract features for grouping should be used sparingly due to the threat of misunderstandings and the reasons mentioned in [66].

Concrete features, shown in dark blue, are features that refer to an individual interface implementation. For instance, the concrete features *AML-C*, *AML-A* and *File-C* each refer to a specific service implementation in the pool of services of the platform which all implement the general *Commit* interface. If you recall Figure 5.2 there are two service tasks called *Checkin file* and *Checkin AML*. These tasks can call one of the variants depicted in the feature model. So for example, the feature model provides two alternatives, *AML-C* and *AML-A*, for the *Checkin AML* task. The features in the model have a direct mapping to one of the service implementations in the platform.

One essential part of the feature model is that the constraints between the features, which represent the dependencies amongst interfaces and services can be defined. In Figure 6.2 the constraints are stated directly over the legend of the feature model. In

the use case, several services need the model of the data format that is checked into the platform. For instance, the *AML* model component is needed for operations with the *AML* tool data and the *EPL* and *OPM* model bundle is need when *CSV* data from the respective tools are imported. In case of the example, the abstract feature *AML-Commit* requires the *AML* model feature to be present in a valid configuration of the integrated tool application. Besides, any *Model* feature requires the *Unified* repository to be present, that allows the check-in of models as well as files.

Another relevant part is how the features are marked according to the options that a feature model provides. The provided feature options and their relation amongst each other, are the ones in the middle and right section of the legend in Figure 6.2 in particular *mandatory*, *optional*, *or* and *xor*. The rules that are defined for the feature model are explained in the following. If an *abstract* feature is needed by all services, it should be marked *mandatory*. If not, it should be marked *optional*, and the features using it need to define a specific constraint, that indicates the dependency. For example, in Figure 6.2 the *Commit* feature is marked optional, as it might not be needed for the integrated tool application if the data should not be stored, but, for instance solely checked for its correctness. If an *abstract* feature has several concrete features, as, for example, the *Repository* feature in the figure, those features have to be marked as alternatives, in order that at least one of them is selected during the configuration. *Concrete* features that are the only child of an *abstract* feature, need to marked as mandatory so that if the *abstract* ancestor is selected in a configuration, the descendant needs to be selected for a valid configuration.

The feature model with its features and constraints must be created in an initial effort by an engineer which is quite familiar with the tool integration platform and experienced in the *CCP*. This effort must also be counted to the overall effort of the *CCP*. However, with an automated process, the invested effort should pay off quickly. Another aspect, which has to be taken into account, is the change of the feature model when the platform modules change. This would, for example, be the case if new modules are implemented and represent a new service variant or old modules are 'retired' for further usage in projects. Such a marginal adaptation of the feature model should not result in too much effort as also only parts of the model change and constraints might have to be reformulated. As dependencies between the interfaces and their implementations are also existing in the source code which is supported by the build tool Maven, the author proposes to investigate the possibility to semi-automatically create parts of the feature model based on these dependencies in future work.

6.3.4 Tool Support and Data Format

The *FeatureIDE* as tool supports the creation of feature models and their configuration, as mentioned in Section 2.3. The feature model for the use case was manually created within this tool. The *FeatureIDE* also provides an XML notation for the feature model and its constrains. A detail of the XML representation of the feature model for the *signal change management* engineering process can be seen in Listing 6.4.

The structure of the feature model is described within the `struct` section, while the constraints are defined within the `constraints` section of the XML document. Features have a `name` as attribute and can be set `abstract` and `mandatory` via the respective XML attributes. The structure of the feature model is built via `and` tags, or in the case that the features exclude each other, via `alt` tags. Features that are leaves in the feature graph use `feature` tags. Constraints are defined via the `rule` tags and contain several basic commands to build such rules. For instance, as the *Model* feature requires the *Unified* feature to be present, the feature names are used as variables and combined by the `imp` (*implies*) tag.

The XML exchange format for feature models provided by the *FeatureIDE* will be used to support the mapping of service instances represented in the model with the variants in the *BPMN* process descriptions. This process will be described in detail in the next section.

```
1  <featureModel>
2    <struct>
3      <and abstract="true" mandatory="true" name="Checkin">
4        <and abstract="true" name="Commit">
5          <and abstract="true" name="AMLCommit">
6            <feature mandatory="true" name="AMLGitCommit"/>
7          </and>
8        </and>
9        <alt abstract="true" mandatory="true" name="Repository">
10         <feature name="File"/>
11         <feature name="Unified"/>
12       </alt>
13     </and>
14   </struct>
15   <constraints>
16     <rule>
17       <imp>
18         <var>Model</var>
19         <var>Unified</var>
20       </imp>
21     </rule>
22   </constraints>
23 </featureModel>
```

Listing 6.4: Detail of the *signal change management* feature model in XML

To summarize this section, the concepts of interfaces as abstractions for concrete implementations and how this can describe a way of variability was discussed. Furthermore, the section explains how the concepts are used in *OSGi* and the *EngSB* implementations. Afterwards, an approach was presented, that considers these concepts to build feature models from the services of the tool integration platform by defining interface bundles as abstract features and their implementations as concrete features in the model. The

definition was build looking at the use case of *signal change management* as a real-world example. Finally, it was discussed that the *FeatureIDE* enables the creation of feature models but also provides an XML notation for the model that contains the structure as well as the constraints. This XML notation will be exploited to map features with *BPMN* process activities.

6.4 Mapping of Engineering Processes to Software Variants

In the last sections, the basis was laid to allow the generation and selection of engineering process variants. Furthermore, a variability model for the existing software service variants of the tool integration platform was defined.

This section now describes how the engineering process variants should be mapped to service variants, which are described in the feature model. The feature model must, however, already exist or an expert must create it beforehand.

The mapping process will be explained using Figure 6.3. The upper part of the figure contains an engineering process template for the quality assurance process in *BPMN*, that should be mapped to a particular variant of a software component configuration. The lower part of the figure depicts a feature model of the services available in the tool application platform. The feature model displays features representing software components, which can be called by the process activities and are mainly services as well as their abstract ancestors. Furthermore, the feature model shows dependencies in between the features, in this case, that the feature *EMF* requires a *Unified Repo* to be present to work as expected. In the middle of the figure, a placeholder for the configuration of the feature model is illustrated. Such a configuration contains a set of features as well as the features they depend on, according to the constraints in the feature model. The configuration can have two different states. The first state is a valid one in respect to the feature model, and thus holds all necessary features to meet the constraints of the model. The second state is incomplete or invalid, which means, either concrete features are missing and the configuration needs to be further completed, or the configuration violates the model constraints, for example, because two features are selected that exclude each other.

In Section 6.1.2 it was mentioned, that for process tasks the *BPMN* notation defines an attribute `operationRef`. This attribute is used to reference a specific operation or service that is called during the execution of the process. In the same section, the author stated that for the approach of the thesis, this attribute is filled with a placeholder, instead of the service call itself, which is created from a name and the prefixed keyword `abstract`. At the same time, the feature model, which is used to model the variability of software services in the tool integration platform as described in Section 6.3, contains an abstract feature, which is specified by the same name as the *operationRef* attribute, but without prefix. So *BPMN* process tasks, in their XML notation, reference names of

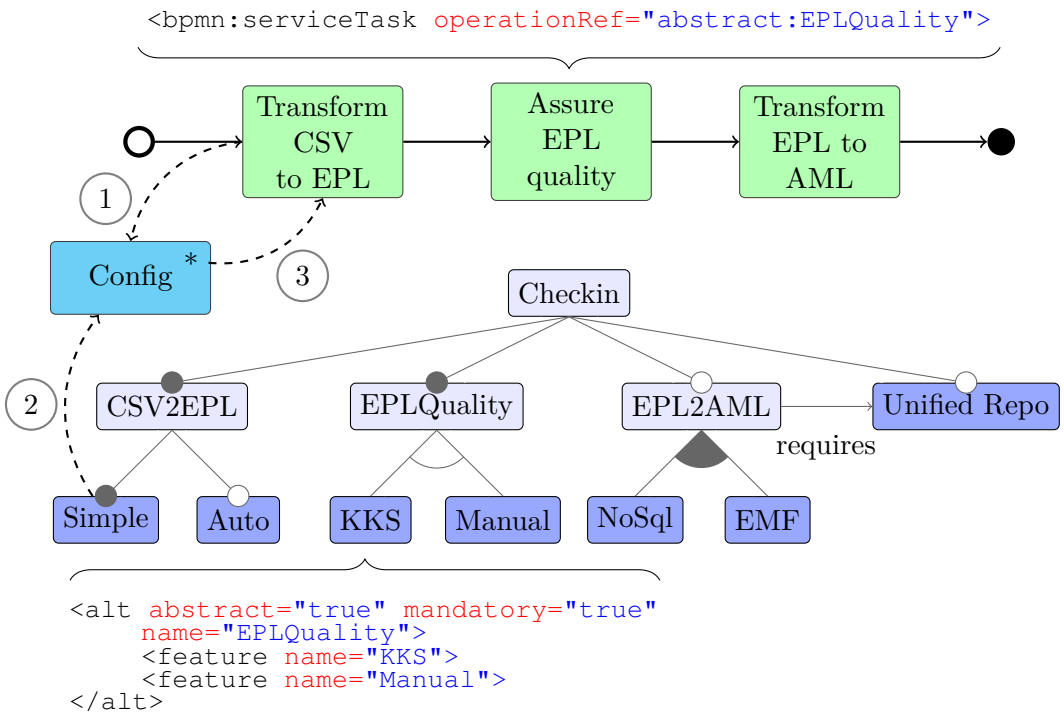

Figure 6.3: Mapping of engineering process activities with software features

abstract features from the feature model. For example, in Figure 6.3 the *Assure EPL quality* task, in its *operationRef* attribute, holds the value `abstract:EPLQuality`. The same value, without the prefix, is used for the abstract feature *EPLQuality* in the feature model. The *EPLQuality* feature has the two concrete features *KKS* and *Manual* as descendants. These features refer to specific services of the tool integration platform, which can be added to the integration tool application during the *CCP* and then called during runtime. To map the process activities to the services, one of the concrete features need to be selected during the mapping process.

The mapping process itself consists of the following subsequent steps that are described in the paragraphs below. First, from the *BPMN* process templates chosen and combined by the application integrator, all `operationRef` attributes are extracted, and their prefixes are stripped away to receive the relevant part. The values are then written into a basic configuration for the feature model. This step is marked in Figure 6.3 with *1* and results in a configuration with the values *CSV2EPL*, *EPLQuality* and *EPL2AML*. The configuration now contains the abstract features that are referenced by the engineering process template. However, this configuration is incomplete and invalid as a valid configuration might only contain concrete features, and most likely the constraints of the feature model are not met. For example, the *requires* dependency between the *EPL2AML* and the *Unified Repo* feature are not met.

The next step is to load the incomplete configuration into a tool like the *FeatureIDE* that is capable of manipulating and validating the configuration according to the feature model. In this tool, the application integrator selects concrete features corresponding to the proposed abstract features. This step is indicated by *2* in the corresponding figure. The configuration is validated on-the-fly to inform the application integrator whether his choices are allowed or not.

The valid configuration resulting from the previous step contains the needed features as well as the dependencies these features require. In the XML notation of the feature model, the features have a `name` attribute, as shown in Listing 6.4, containing a value that is unique within the model. However, this name does not necessarily match the exact call that needs to be used in the specific process task to trigger the service. Therefore, a mapping table is created during the definition of the feature model, that maps the name of the concrete feature of the model to the concrete call that is used to trigger the service during runtime. In the final step, highlighted with *3* in Figure 6.3, the process tasks are enriched with the concrete service calls, which correspond to the features in the final configuration.

To summarize this section, based on the previous contributions that provide a variability model for *BPMN* processes and a variability model for the software services of a tool integration platform, a method was explained how these models could be mapped to enrich the process tasks with valid service variants according to a feature model.

6.5 Improved Customization and Configuration Process

The last four sections provided concepts to model the variability in engineering processes – contribution *3a* – and the variability in the software services of the tool integration platform – contribution *3b*. Furthermore, these sections described a mechanism to select engineering process templates and their dependencies based on the process variability model – contribution *2a* – and finally characterized a method to map the engineering process templates for software service variants – contribution *2b* – in order to generate engineering process variants, that are able to call services in a customized integrated tool application.

This short section puts the bits and pieces from the previous sections together, to form the big picture of the solution approach. This way, the improvements that were achieved for the *CCP*, in comparison to the traditional approach, which was performed manually, will be shown.

The following paragraphs will discuss the parts that changed in the *CCP*. At the beginning of this chapter, Figure 6.1 was used to structure the sections and contributions of the solution approach. This time, the figure will be used to explain what the solution approach changed in the *Process Selection and Configuration* task of the *CCP*.

In the traditional approach the *Selection* step (see *2a* in the figure) was performed by *cloning and owning* existing engineering processes from other customization projects or

completely generating them from scratch. The catalog of engineering process templates, which is created using the *Process Variability Model*, helps the application integrator during the *CCP* to select process templates, that are then adapted to the requirements of the customer.

The *Linking* step, where parts of the selected engineering processes are rearranged or combined in case of sub-processes, is still the way it was in the traditional approach. However, the starting situation for this step has changed, as only processes need to be adapted, that were already pre-filtered by the selection backed by the solution approach.

One especially tedious activity in the *CCP* is the *Mapping* of the engineering process tasks to the service variants that are called by the processes during runtime (see *2b* in Figure 6.1). In this step, the application integrator needs to search in a sometimes vast catalog of possible services and their dependencies to choose valid service configurations. The solution approach supports this step by providing a method that allows a systematic mapping of engineering process templates to service variants, modeled in the *Platform Variability Model*.

Finally, the enrichment of the process tasks with the service calls was done by searching the respective call values and writing them into the process tasks. The solution approach provides a mapping table that allows finding the values corresponding to the chosen service variants quickly.

Overall, by the application of the solution approach to the *CCP* two main goals should be reached. First, the process should gain tremendously in efficiency compared to the traditional approach, that was done in tedious and manual work by the application integrator with his expertise. Second, the improved process should provide additional spots where quality assurance mechanism can be plugged in to prevent the production of error-prone integrated tool applications.

6.6 Summary

This chapter proposed a solution approach to the manual and error-prone *Customization and Configuration Process (CCP)* for tool integration platforms by applying methods and models from *Variability Modeling* and weaving them into an improved process. The chapter was organized into five sections, that each contributed to the whole solution approach, according to Figure 6.1. The first section explained how a *Process Variability Model* could look like for engineering processes that are modeled in *Business Process Model and Notation (BPMN)* by using attributes of the syntax to refer to templates of sub-processes and service references. The second section illustrated a method that uses the *Process Variability Model* from Section 6.1 to support the *Selection* phase of the *CCP* by following the template references to gather the relevant parts of an engineering process. Third, the *Platform Variability Model* and how it is built on *Feature Modeling (FM)* was illustrated, that serves as input for the *Mapping* phase of the *CCP*. Section 6.4 introduces a method to map the engineering process templates, resulting from the *Selection* phase,

with the software components of the tool integration platforms represented by the feature model whose semantics were defined in Section 6.3. Finally, a short section on the changes in the *CCP* summarizes what was improved for the process.

The next chapter presents the evaluation of the solution approach proposed in this chapter supported by a prototype that was developed for this purpose, by applying it on a real-world use case, which was investigated at an industry partner.

CHAPTER 7

Evaluation

In the previous chapter, a solution approach was formulated to improve the *Engineering Process Selection and Configuration* phase of the *Customization and Configuration Process (CCP)* for tool integration platforms based on the concepts of the *Engineering Service Bus (EngSB)*. Following the methodology for this work described in Chapter 4, the previously proposed approach was evaluated utilizing a 'real world' example in combination with a simple prototype, which implements the relevant parts of the approach. The evaluation set up and the findings of this evaluation are described in this chapter.

The remainder of this chapter is organized as follows. Section 7.1 describes the tasks that needed to be performed to create the *Platform Variability Model* and to prepare the existing engineering processes to generate the *Process Variability Model*. Section 7.2 introduces the prototype and explains how it supports the evaluation regarding assisting the application integrator during the *Engineering Process Selection and Configuration*. Section 3.2 discusses how the use case from Section 5.3 was applied and how the evaluation was conducted. Finally, Section 7.4 presents the results of the evaluation.

7.1 Preparation of the Evaluation

As mentioned in the introduction of this chapter, the solution approach was evaluated by taking advantage of a 'real world' example. Section 5.3, therefore, introduced the *Signal Change Management Process (SCMP)*, which was identified as an important process part of *round-trip engineering (RTE)*. The *SCMP* was observed during investigating the processes of an industry partner and described by Winkler et al. [74]. It was also observed, that different variants of the *SCMP* were utilized by the engineers of the specific industry partner. Additionally, researchers of the CDL-Flex experienced that other industry partners, whose processes were examined, applied very similar processes to the one already described. Because of the frequent occurrence of this engineering process at industry partners, the evaluation of the approach proposed in Chapter 6 was

based on the use case of one of the specific industry partners. This industry partner already uses a customized version of the *AutomationML Hub (AML.hub)* as integrated tool application and implemented an individual version of the *SCMP*. For the evaluation this means the engineering process of the industry partner was used as a basis for the *Process Variability Model* and the existing platform and its services were used as a basis for the *Platform Variability Model*. Before the evaluation could be conducted, several elements had to be set up beforehand.

7.1.1 Platform Variability Model preparation

The first preparation task was to create the *Platform Variability Model* from the software services of the tool integration platform. This step had to be performed manually by an expert with a good knowledge of the platform, the software services within the platform and especially their dependencies amongst each other. The expert thoroughly investigated the platform and created a feature model from the services and their dependencies in the *FeatureIDE*.

As described in Section 6.3, the interface bundles, which are deployed separately from the service implementations to the *OSGi (OSGi)* container, were used as *abstract* features of the feature model. For example, for the service that checks the *AutomationML (AML)* consistency and is used for the *Check AML consistency* task shown in Figure 5.2, a Maven module was available, that only contained the Java interface. As an explanation, Maven is a software project management and build tool that is widely used in Java programming to structure and build software projects. In Maven it is possible to structure and separate components using Maven modules. This also allows components to be build and deployed separately. Furthermore, Maven provides a dependency management system, that automatically resolves dependencies that are stored in a Maven repository. This interface contained the functional specification of the methods, their input values as well as their return parameters for a service that is able to check the correctness of an *AML* file. The expert also found an obvious dependency to a software interface bundle, that described the *AML* model. This dependency was added as a constraint to the feature model. For better comprehensibility, the expert, added several abstract to further structure the model.

The *concrete* features of the feature model were derived from the specific service implementations that existed in the platform. Therefore, the expert listed the existing service implementations and then examined which interfaces these implementations used. He then created the respective concrete feature in the feature model below the corresponding abstract feature that represents the service interface.

According to the rules defined in Section 6.3 the features in the model, then needed to be classified according to the available options. In particular, this means that the features needed to be marked whether they are *mandatory* or *optional* and if two concrete features are an alternative (*xor* relation) or can be used in combination (*or* relation). So the abstract features were then marked as *optional* to indicate that they can be selected in

the configuration. If a concrete feature was the only descendant of an abstract feature, it was marked mandatory. If a concrete feature, on the other hand, had a sibling, it was marked as an alternative using the *xor* relation, to indicate that either one of them can be used in the final configuration.

Finally, the expert had to create the mapping table, which defines the service calls which are used in the service tasks to call the specific service instance. Therefore, the expert created a simple text file, which contained the names of the concrete features and for each of these names the correct service call under which the service can be reached from the workflow engine.

The preparations described in this section resulted in a feature model that represented the services and components of the tool integration platform and their dependencies amongst each other. The feature model was created using the *FeatureIDE* and saved as a model file in an XML representation. Furthermore, a mapping table was defined to link the feature names of concrete features with the respective service calls that are used within the running platform. These two artifacts together build the necessary parts for the *Platform Variability Model*, that is used in the *Mapping* activity of the *Engineering Process Selection and Configuration* task.

7.1.2 Process Variability Model preparation

The second preparation activity was to prepare the existing engineering processes, to afterward create a *Process Variability Model*, as proposed in Section 6.1. Therefore, the process needed to be modeled in *Business Process Model and Notation (BPMN)* first, which was done using the Camunda Modeler[1]. The Camunda Modeler is a freely available tool, which allows creating *BPMN* process and *DMN* [50] diagrams and is distributed by Camunda, which is the same company that develops the Camunda Workflow Engine. This modeling step resulted in four *BPMN* process diagrams. One that looks very similar to the engineering process shown in Figure 5.2 and three sub-processes diagrams for the respective call activities. Each of these diagrams is backed by an XML file containing the process models in *BPMN* notation. Figure 7.1, for example, shows the sub-process of the *Select changes* call activity. Depending on the model (*AML*, *EPL* or *OPM*) that is checked in, different steps have to be executed before the changes in the data can be selected to commit them to the database afterward.

As the sub-process contained a combined process of rather an independent process, the sub-process need to be separated into its minimal viable processes, as recommended in the solution approach. This second step was necessary, to build the catalog of templates, that can be reused in the *Customization and Configuration Process (CCP)*. This step resulted in the blueprint of the main process, as described above, and several separated sub-process templates, that look very similar to the processes shown in Figure 5.3 and Figure 5.4. For example, the *Select changes* sub-process was split into three sub-process templates, corresponding to the type of model that is checked in.

[1]Camunda Modeler – `https://camunda.com/download/modeler/`

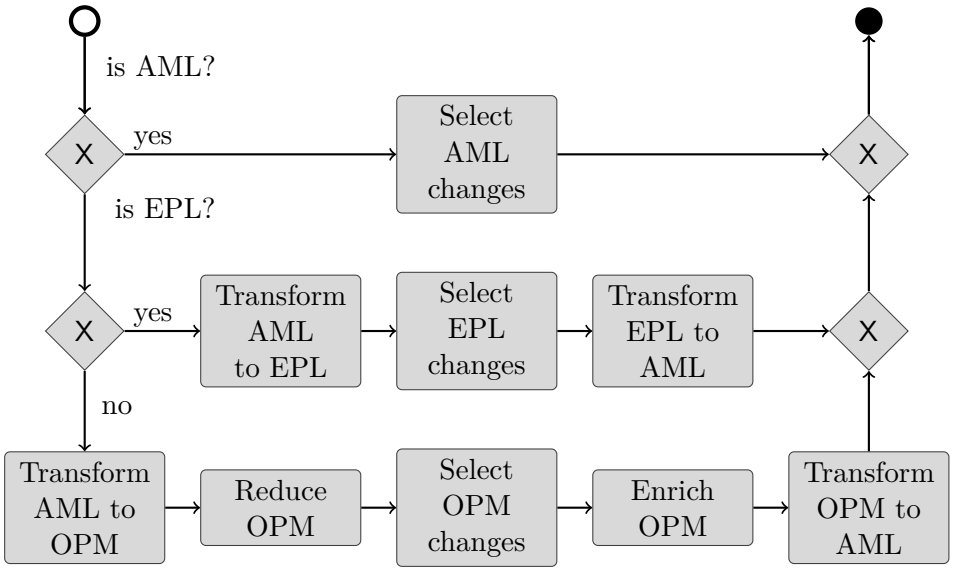

Figure 7.1: Sub-process before separation to minimal processes

The process templates were then stored as follows, to create the process catalog as mentioned in Section 6.1. A root folder was defined, that held all top-level engineering process templates. For example, the main process, depicted in Figure 5.2, was saved as *Checkin.bpmn* in this root folder.

The sub-process templates for each call activity, which derived from the separation of the combined sub-processes, were saved in a folder named after the call activity postfixed by the string `Templates`. For instance, the three separated sub-process templates of the *Select changes* sub-process, shown in Figure 7.1, were stored in a folder named `SelectChanges-Templates`.

Finally, the *BPMN* process activities needed to be adapted to represent the variability of the processes. This means, *call activities* needed to refer to sub-process templates and *process tasks* needed reference an abstract feature of the feature model. As mentioned in section 6.1.2, *BPMN* provides the semantics to call sub-processes and services via specially defined attributes within process activities. Therefore, on the one hand, each of the `calledElement` attributes in a *call activity* got the value of the folder assigned, where the sub-process templates were stored, prefixed by the `template` keyword. On the other hand, each *process task* was enriched by the name of the corresponding abstract feature from the feature model, prefixed by the `abstract` keyword for the `opertionRef` attribute. However, during the modeling of the processes in Camunda Modeler, the author experienced, that the process model defined within the Modeler does not include an `operationRef` attribute in the *process task* that can be easily accessed in the tool. The author, therefore, decided for the evaluation of the solution approach, that the attribute `camunda:delegateExpression` is used instead of the `operationRef`

attribute to reference the abstract features. In future work, this issue should be addressed by either extending the modeling tool or the prototype.

To sum up, this section described which tasks had to be performed, to create the *Process Variability Model* from the engineering processes implemented at the industry partner. First, the processes were modeled in *BPMN* and their sub-processes separated into their minimal viable processes. Second, the process templates were stored in a folder structure that can be used as a catalog for the *CCP*. Finally, the processes were adapted corresponding to the recommendations in Section 6.1.2 to refer to sub-process templates and abstract features.

7.2 Prototype for the Evaluation

To properly evaluate the solution approach proposed in Chapter 6 and show its feasibility, a simple prototype was created, that reproduces the *Customization and Configuration Process (CCP)* and in particular, the tasks of the *Engineering Process Selection and Configuration* as depicted in Figure 6.1. This section introduces the prototype and the functionality it provides, to support an application integrator during his configuration work.

To better explain what the prototype does and how the evaluation was conducted, the tasks of the *Engineering Process Selection and Configuration* in its improved form are recapped. First, the top-level process templates as well as the sub-process templates, that are referenced by the top-level processes, need to be selected by the application integrator. Second, in a manual step, the application integrator needs to combine the selected sub-processes of each *call activity* to form a single sub-process. In a third step, the chosen processes are analyzed and basic configuration for the *Platform Variability Model* is written. Afterwards, this configuration is completed by the application integrator in the *FeatureIDE* to create a valid configuration according to the constraints of the *Platform Variability Model*. Finally, the valid configuration is used to enrich the chosen processes with the service calls that are used during runtime by a workflow engine to trigger the specified functions. The described functionality should be supported by the prototype as good as possible, which also means, that as many of the steps as achievable should be automated.

```
1 xmlstarlet sel -t -m "//bpmn:callActivity" -v "@calledElement" -n $x
```

Listing 7.1: Extraction of sub-process template

The prototype itself consists of a set of scripts, which mostly make use of XMLStarlet[2]. XMLStarlet is a command line tool to directly query but also manipulate the XML

[2]XMLStarlet – http://xmlstar.sourceforge.net/

documents utilizing XPath[3] and XSLT[4]. In this case, the tool is used to query and manipulate representations of the *Business Process Model and Notation (BPMN)* process templates as well as the feature model.

The prototype requires the following parameters to start the process: a) the location of the root folder was the process templates are stored; b) the location of the folder were the configured processes are saved; c) the file location of the XML representation of the feature model that is analyzed; d) the file configuration of the configuration file where the basic configuration is saved, and the finished configuration is written to, and finally e) the file location of the mapping file that contains the service call mapping.

```
1 xmlstarlet sel -t -m "//bpmn:serviceTask"
2                -v "@camunda:delegateExpression" -n $x
```

Listing 7.2: Extraction of the abstract feature reference

The prototype now does the following. The prototype, first, searches all existing top-level process templates by their `.bpmn` file ending from the root folder and presents them to the application integrator. The application integrator selects the templates that are suitable for the specific customization of the tool integration platform. From each of the selected top-level process templates the sub-process template references are automatically extracted using the command shown in Listing 7.1, whereas `$x` is the specific *BPMN* file, and cutting of the `template` prefix. Similar to the process templates, the list of extracted sub-process templates is presented to the application integrator, who selects the relevant ones. The top-level templates with their corresponding sub-process templates are copied to the target folder for further processing.

```
1 xmlstarlet sel -t -m "//bpmn:callActivity[
2                         starts-with(@calledElement,'template')]"
3                -v "@calledElement" -n $process
```

Listing 7.3: Retrieval of all *call activity* starting with `template`

After these steps, the application integrator needs to manually combine the sub-process templates of a specific call activity to a single sub-process. Furthermore, he needs to change the top-level process templates the way he needs them to be from the perspective of the sequence flow. When the task of changing and combining the process templates is finished, the prototype can perform the next steps.

To create the initial set of values for the feature model configuration, the prototype would go through all the remaining `.bpmn` files in the folder and extracts their service call references defined in the `operationRef` attribute. However, as mentioned before, instead of the `operationRef` attribute, the Camunda specific `camunda:delegateExpression`

[3] W3C XPath – `https://www.w3.org/TR/xpath/`
[4] W3C XSLT – `https://www.w3.org/TR/xslt/`

attribute was used for the evaluation. Listing 7.2 shows the `xmlstarlet` command that was used for the extraction of the abstract feature references from the process templates. The `abstract` keyword is stripped from the values, and they are then written to the initial configuration file.

```
1 xmlstarlet ed --inplace
2               --update "//bpmn:callActivity[
3                           @calledElement='$template'
4                         ]/attribute::calledElement"
5               -v "$variable" $process
```

Listing 7.4: Enrichment of the *call activity*

The basic configuration then needs to be completed by the application integrator in the *FeatureIDE*. Therefore, the application integrator loads the basic configuration into the configuration editor of the, where he gets a tree view of the configuration with the abstract features checked. The integrator then needs to subsequently select concrete features for the abstract ones until the editor validates the configuration as valid. The configuration editor, in the background, automatically checks with every change, whether the configuration is still invalid or already valid considering the constraints of the underlying feature model. When the application integrator is satisfied with the configuration, and the configuration is valid he saves the file and proceeds with the process by using the prototype again.

```
1 xmlstarlet sel -t -m "//feature[
2                         @name="'$feature'"]/ancestor::*[@abstract][1]"
3               -v "@name" -n $model_file
```

Listing 7.5: Retrieval of the ancestor of a concrete feature

The prototype now does two things. First, the `calledElement` attributes of the top-level process blueprints in the target folder are configured with the correct call of the sub-process instead of the template files. This enrichment is done with the two commands shown in Listing 7.3 and 7.4. The first command retrieves all attribute values of `calledElement` attributes that start with the keyword `$template`. This list is then fed to the second command, which searches the `callActivity` element with the particular `$template` value and changes it from the sub-process template placeholder to the specifically combined sub-process, that was created manually by the application integrator.

Second, the prototype enriches the process tasks with the concrete service calls. Therefore, it takes the concrete feature values from the valid configuration file one by one and searches the first direct ancestor of the concrete feature in the XML representation of the feature model. This feature is the corresponding abstract feature that is referenced in the process template. The command and query for this task are listed in Listing 7.5.

The prototype then with a second command – see Listing 7.6 – searches the abstract feature in the process template and assigns the the value of the concrete service from the mapping file (see Line *6* of the listing – `${${service_mapping[$feature]}}`) to the `camunda:delegateExpression`.

```
1 xmlstarlet ed --inplace
2               --update "//*[
3                           @camunda:delegateExpression=
4                           'abstract:$service'
5                         ]/attribute::camunda:delegateExpression"
6               -v "\${${service_mapping[$feature]}}" $process
```

Listing 7.6: Enrichment of the *process task*

After this step, the *BPMN* files stored in the output folder, were assembled according to the requirements of the application integrator respectively the customer. The service tasks within the engineering processes models were mapped to the selected software service variants and configured with the proper service calls. The *BPMN* files, after this task, are ready to be deployed to the integrated tool application that runs a workflow engine like Camunda.

To sum up, this section provided an overview of the prototype and how it was implemented. Furthermore, the section outlined what the prototype does to support the *Engineering Process Selection and Configuration* tasks in consideration of the proposed solution approach of Chapter 6.

7.3 Evaluation Procedure

The last section described the prototype that was used to evaluate the proposed solution approach for an improved *Customization and Configuration Process (CCP)* for tool integration platforms. This section discusses how the evaluation was conducted to measure the relevant results for the *Key Performance Indicators (KPIs)*, which are used as performance metrics, to compare the traditional and the improved process.

7.3.1 Evaluation use case

The evaluation of the solution approach was similarly performed on the *Signal Change Management Process (SCMP)* use case described in Chapter 5, for both the traditional as well as the improved *CCP*. This includes, on the one hand, the *SCMP* engineering process, as outlined in Section 5.3 (see also Figure 5.2), and, on the other hand, the customized *AutomationML Hub (AML.hub)*, as it was delivered to the industry partner, as an example for an integrated tool application. First, the traditional approach was played through and measured, afterward, the improved approach was performed and the results recorded. The feature model utilized for the improved approach represented a detail of a bigger feature model, which models the services and components available

for and within the tool integration platform. To explain the tested real-world example in detail, table 7.1 shows some relevant statistics of the *SCMP* engineering process and the underlying customized *AML.hub* and accordingly the created feature model for the integrated tool application. Additional material like the sources for the use case, the processes and the feature model can be found in the corresponding repository[5].

SCMP	Entities	Sum
Process tasks	Recognize model, Compare AML, Check AML compliance, Merge AML changes, Check AML consistency, Checkin AML, Checkin file, Notify users	8
Call activities	Check AML quality, Check Table quality, Select changes	3
SP variants	Check AML quality, Check EPL quality, Check OPM quality, Select AML changes, Select EPL changes, Select OPM changes	6
SP tasks	Transform CSV to EPL, Assure EPL quality, Transform EPL to AML, Transform AML to EPL, Select EPL changes, Transform CSV to OPM, Assure OPM quality, Enrich OPM, Transform OPM to AML, Transform AML to OPM, Reduce OPM, Select OPM changes, Select AML changes, Assure AML quality	14 (+3)
Services	see corresponding Repository	27 (+2)
Constraints	see corresponding Repository	20

Table 7.1: Statistics of the *SCMP*

For the *engineering process* itself the following statistics apply. The top-level process template contained 8 regular *process tasks* – see also Figure 2.7, which are listed in the first row of Table 7.1. Additionally, the top-level process included 3 *call activities*, which referred to different sub-process template categories. For these *call activities*, 6 independent sub-process templates were listed in the process catalog that was built from the process. Within the sub-process templates, 17 process tasks were found, whereas 3 tasks reoccurred in other sub-process templates, which makes 14 distinct process tasks with the templates.

For the corresponding *feature model* the last two lines of the table are relevant. First, 24 services and components relevant for the engineering process itself were identified and modeled as features, at which for two abstract features – *Notfication* and *EPLQualityAssurance* – in each case two service implementations were found as variation points. Furthermore, 5 components were found and modeled as features, which needed to be included because of the dependencies between them. Finally, 20 constraints were formulated in the feature model, to describe the dependencies among the software components respectively the features in the model.

[5]Additional material – `https://bitbucket.org/mercynary/thesis-additional`

7.3.2 Measurement of KPIs

Section 3.2 defines several *KPIs* that were used to measure the performance of the solution approach. The following paragraphs describe how the measuring was done for the different *KPIs*. In the course of this section the term 'configuration process' is used for the tasks of *Engineering Process Selection and Configuration* for simplicity. Furthermore, the term 'approach' is used to either describe the traditional configuration process or the improved one.

The first two *KPIs* defined – *KI-1.1* and *KI-1.2* – consider the number of activities that need to be performed during the configuration process. While *KI-1.1* uses the manually performed activities, *KI-1.2* also takes the automated activities into account. For both *KPIs* the number of activities was counted to record the values and calculate the *KPIs*. For the traditional as well as the improved configuration process the following activities were counted as one point: a) each creation of a single *feature* in the feature model, regardless whether the feature is abstract or concrete; b) each creation of a single *constraint* in the feature model, whereas the classification as, for example, *mandatory* also counts as constraint; c) each creation of an entry for the mapping table; d) each selection of a top-level process or template; e) each selection of a sub-process or template; f) each combination or separation of a sub-process or template; g) each search step for a service call – meaning if several alternatives exist, each alternative that needs to be found is counted as one step; h) each selection of a service to call; i) each resolution of a dependency for a selected service, and finally; j) each configuration of a process task with the specific service call. Obviously, for activities not needed for one of the approaches, zero points are counted.

The second set of *KPIs* uses the working effort of the configuration process as measurement category. The first *KPI* (*KI-2.1*) measures the overall effort invested in the configuration process. Therefore, the time needed to go through the configuration process once was measured in minutes for each of the approaches. However, the manual task of process combination was left out of this metering. The combination task is either needed to cut out some activities of a sub-process that was *cloned* or to combine the sub-process templates from the *Platform Variability Model*. In this task, the time of modeling the process in Camunda Modeler strongly depends on various fuzzy values like, tool skills or diagram alignment, and is thus not very accurate. It is estimated, that these two tasks take nearly the same amount of time. The second indicator (*KI-2.2*) measures the effort when: a) a process task is added to an engineering process that uses an existing service, b) an additional service is added to the platform, and c) a process task is added together with an interface and two service implementations. For this indicator, the additional steps that are necessary to adapt and configure the engineering process were measured.

At last, the *KPIs KI-3.1* and *KI-3.2* measure were quality mechanisms, to ensure the correctness of the resulting processes, are already implemented in the configuration process and were such quality mechanisms can be run in an automated fashion to minimize the manual effort. For each of these indicators, the number of mechanisms in the traditional

and the improved approach is counted and summed up.

The values for the *KPIs* were measured by hand. Although this is not the most reliable method, for the traditional *CCP* it is not possible to measure the values otherwise. An improvement, however, would be to measure the activities under the supervision of other researchers or in a case study to be able to calculate an average from the results of different test subjects.

7.4 Evaluation Results

The following section presents the evaluation results of the solution approach performed with a prototype, implemented, therefore, compared to the traditional approach of *Engineering Process Selection and Configuration*. The sections are grouped by the sets of *KPIs* described in Section 3.2 and Section 7.3.2.

7.4.1 Evaluation results for Complexity indicators

First, the results for the measuring of the first set of *KPIs* (*KI-1.1* and *KI-1.2*) are presented. Table 7.2 shows the raw numbers for the measuring of the traditional and the improved approach for the calculation of *KI-1.1* and *KI-1.2*.

Activity ↓ Manual steps →	Traditional	Improved
Feature model *a)*	0	70
Constraints *b)*	0	75
Mapping entries *c)*	0	29
Process selection *d)*	1	1
Sub-process selection *e)*	0	6
Process combination*f)*	3	3
Service search *g)*	22 + 2	0
Service selection *h)*	22	22
Constraint resolution *i)*	20	0
Task configuration *j)*	25	0
Sum initial run	95	206
Sum every additional run	95	32

Table 7.2: Evaluation measurements for *KI-1.1* and *KI-1.2*

The creation of the features in the feature model – see Section 7.3.2 *a)* – was performed in 70 manual steps. This also included the proper structuring of the model to make it easy to understand for engineers, that are not very familiar with the underlying tool integration platform.

The creation of the constraints – *b)* – took 75 manual steps. This included the definition of explicit constraints in the constraint language of the feature model, as well as implicit

constraints such as the categorization of a feature as *alternative* (xor relation) to another one.

The creation of the entries for the mapping table took 29 steps, as for each of the concrete features an entry in the mapping table needed to be created.

The selection of the top-level process – *d)* – needed to be done in the traditional approach and the improved one. However, the selection, in the latter case, was supported by the prototype instead needing to search the process. Nevertheless, for each of the approaches, 1 point was counted.

For the sub-process selection – *e)* – it is assumed, that the *cloned* engineering process already includes the sub-processes to be called in the same diagram. Therefore, no manual step was needed and counted.

On the other hand, the prototype developed for the improved approach, let the application integrator choose from the sub-process templates which ones to take. As 6 sub-process templates were needed for the *SCMP*, the application integrator selected them in 6 manual, but prototype supported steps.

The possible adaptation of the sub-processes and the combination of the selected sub-process templates – *f)* – was counted with 3 points for each of the sub-process calls that are made by the call activities.

For the step of searching the services that fit to the engineering process tasks – *g)*, 24 manual steps were needed in the traditional approach, 22 steps were needed to search the services for the engineering process and 2 additional steps were needed to find the alternative services that could have been selected for the final engineering process. The improved approach already has to services defined in the feature model and finds them automatically, so no manual step is needed.

The selection of the concrete service – *h)* for the process tasks took 22 steps for each of the approaches, as each for process task that was found in the engineering process a service was selected. For the traditional approach, it is assumed, that the application integrator remembered the service that was used for the three reoccurring process tasks. Otherwise, the process selection count for the traditional approach would have been 25. For the improved approach, the application integrator needed to select 22 services in the configuration editor of the *FeatureIDE*.

The constraint resolution – *i)* – took 20 manual steps for the traditional approach and zero for the improved approach as the configuration editor does this resolution based on the constraints defined in the underlying feature model.

Finally, the task configuration – *j)* – was done in 25 manual steps for the traditional approach as each of the process tasks needed to be touched once to assign the service call to the particular task. It is also assumed for the measuring, that the application integrator, in the traditional approach, recalled or wrote down the specific service calls for the services he selected in step *h*, otherwise, he would have needed 25 additional manual steps to search the service calls for the services he selected.

	KI-1.1	KI-1.2
Traditional Approach	95	95/0
Improved Approach	206 (32)	206/69 (32/69)

Table 7.3: *KI-1.1* and *KI-1.2* for the traditional and improved approach

From these raw numbers the *KPIs* were calculated as seen in Table 7.3. The numbers in parentheses are the adjusted *KPIs* after the first run of the configuration process. *KI-1.1* for the traditional approach was 95, which are the overall manual steps performed in the *Engineering Process and Selection* phase. For the improved approach, *KI-1.1* was 206, including the steps to prepare the engineering processes for the approach and create the feature model. A second run of the configuration processes would take 32 steps, as the preparation tasks do not need to be executed.

KI-1.2, for the traditional approach, accounted with a relation of 95/0 or 95 manual versus zero automated steps. For the first run of the improved approach, the relation is 206/69 (or ~3 in absolute numbers), and 32/69 (or ~0.46) for the runs after the first one.

7.4.2 Evaluation results for Effort indicators

As mentioned in Section 3.2 the second set of metrics considered the working effort that is invested tasks of the configuration process. As *KI-2.1* and *KI-2.2* in contrast to the previous set of indicators examine different aspects, they are split up into two parts.

	KI-2.1
Traditional Approach	125 minutes
Improved Approach	1200 (25) minutes

Table 7.4: *KI-2.1* for the traditional and improved approach

For *KI-2.1* the working effort to go through the configuration process once was measured in minutes. The *KPIs* are displayed in Table 7.4. It took the author 125 minutes to go through the traditional approach. This included the *cloning* of the engineering process, the manual search of the services in the Maven modules and the artifact repository, the service selection, and constraint resolution, and, finally the process task configuration. For the improved approach, it took the author approximately 12 hours to create the feature model to a point where it was mature enough to serve as a basis for the process. Furthermore, the author invested roughly 5 hours in preparing the engineering processes and their minimal viable sub-processes. Afterwards, it took about 25 minutes to go through the configuration process with the support of the prototype. This is also the time that is needed after the first run for each additional run. The development of the prototype is not included in this effort.

The results for *KI-2.2* are displayed in Table 7.5 and split into three columns to address the three cases described in Section 7.3. For the first indicator – *KI-2.2 a* – a single

	KI-2.2 - *a*	KI-2.2 - *b*	KI-2.2 - *c*
Traditional Approach	2	0	3
Improved Approach	2	4	8

Table 7.5: *KI-2.2* for the traditional and improved approach

process task is added to a top-level process template, which refers to a service that is already implemented and can be used multiple times, such as the *Transform AML to EPL* process task, shown in Figure 7.1. In both, the traditional and the improved configuration approach, the additional process task first needs to be weaved into the engineering process respectively the process blueprint. Also in both cases either the service reference is already assigned (this is the case when the task is copied within the process), or the process task needs to be configured with the specific service call or the placeholder in the improved approach.

For indicator *KI-2.2 b*, a newly implemented service is added to the tool integration platform. For the traditional approach, the service is just implemented, for example, as a Maven module, nothing has to be done for the configuration process. For the improved approach, the service needs to be added to the feature model. It is assumed here, that the service implements an already existing interface and that there is one dependency to another service. The creation of the service in the feature was done in 1 step. Additionally, 1 step was used to mark the feature as an alternative and 1 step to define the constraint for the dependency. Furthermore, the service call needed to be added to the mapping table in 1 steps.

KI-2.2 c measures the steps necessary, to add a process task together with an interface component and two service implementations. In the traditional approach, in this case, an engineer added the process task to the engineering process, then needed to find the correct service call in one step and afterward assigned the service call to the process task. This resulted in an overall of 3 manually performed steps. In the improved approach, several other steps needed to be performed. First, an abstract feature was created in the feature model for the interface. Second, the two service implementations were modeled as concrete features. In a next step, the xor constraint between the two concrete features was defined. Furthermore, for each of the service implementations, an entry in the mapping table was created. Afterwards the process task was added to the engineering process blueprint, and finally, the process task in the blueprint got the abstract feature name assigned. This task was performed in 8 manual steps.

7.4.3 Evaluation results for Quality indicators

The third set of*KPIs* covered the quality mechanisms, which are utilized by the different approaches. The first indicator – *KI-3.1* – investigates the quality assurance mechanisms that come out of the box for the specific approach. The second indicator – *KI-3.2* – examines the quality mechanisms that can easily be implemented and supported by

	KI-3.1	KI-3.2
Traditional Approach	0	1
Improved Approach	1	5

Table 7.6: *KI-3.1* and *KI-3.2* for the traditional and improved approach

automation to minimize manual work. Table 7.6 visualizes the results for the evaluation of the indicators.

The traditional approach had no explicit quality assurance mechanisms implemented, apart from manual control of the outcomes of the process. The improved approach at least provides an automated quality assurance mechanism, as the configuration is consistently checked for validity during the configuration process.

The second indicator analyzes where quality assurance mechanisms can be plugged in to run as far as possible automatically and independently to minimize the effort of application integrators. For the traditional approach, the author only found a single extension point in the configuration process, where quality assurance checks could be installed, considering the engineering process and service calls of the process. A point in the traditional approach where a quality mechanism could be easily implemented and run automatically was the finished engineering process, were each of the process tasks could be checked, whether they have a service call assigned. However, the correctness of the assignment could only be tested in a test run of the system during runtime.

For the improved approach, the author found 5 extension points, where quality checks could be implemented. First, the engineering process templates can be tested, if every call activity has a template value assigned respectively if each process task has an abstract feature value assigned. Second, before the configuration process, the assigned features values for process tasks can be checked for validity considering the feature model and its abstract features. A third test would be to check whether all concrete features are present in the mapping table. Finally, after the configuration of the engineering process, it can be analyzed, if all call activities have a particular sub-process model assigned and if every service was configured with a specific service call from the mapping table.

7.5 Summary

This chapter explained the evaluation setup and procedure and presented the evaluation results. The first section outlined which tasks had to be performed to prepare the evaluation. On the one hand, the section covered how the feature model, which represents the *Platform Variability Model* from Section 6.3, was created from the services and components of the *AML.hub*. On the other hand, the section illustrated how the existing *SCMP* was translated to the *Process Variability Model*. Section 7.2 then introduced the prototype, which is based on several scripts that exploit the command line tool *XMLStarlet*, and how the prototype uses and manipulates the two variability models

described before, to support the application integrator during the *CCP*. The third section detailed on the use case from Chapter 5 as well as how the evaluation and notably the measurement of the metrics was conducted. The last section of this chapter presented the results of the evaluation, including the performance results of the traditional approach and the improved approach, that was supported by the prototype, but also the values for the *KPIs* defined in Section 3.2 and refined in Section 7.3.

The next chapter – Chapter 8 – interprets the results, presented in this chapter, and discusses the findings in consideration of the research issues.

CHAPTER 8

Discussion and Limitations

The last chapter explained the evaluation setup and the prototype that was developed to support the evaluation of the solution approach. Furthermore, the chapter presented the evaluation results according to the proposed *Key Performance Indicators (KPIs)*.

This chapter picks up the results from the previous chapter, attempts an interpretation considering the research issues motivated in Chapter 3 and discusses the findings. Furthermore, the limitations of the solution approach and the prototype will be discussed.

8.1 Research Issues

In the following sections, the research issues are discussed considering the evaluation results. The section starts with *RI-1*, but then discusses *RI-3* before *RI-2* as the latter builds on the concepts of *RI-3*.

8.1.1 *RI-1* – Improvements of the Customization Process

The aim of *RI-1* was to find out, to what extent the proposed solution improved the phase of *Engineering Process Selection and Configuration*, to create fully configured variants of engineering processes. The *Engineering Process Selection and Configuration* phase is a sequence of customization and configuration activities within the *Customization and Configuration Process (CCP)* for tool integration platforms, as depicted in Figure 6.1. By *fully configured*, this work understands engineering processes which, after configuration, refer to software services, which were selected during the customization of a tool integration platform as service variants and are deployed with the integrated tool application, resulting from the *CCP*. *RI-1* also listed three particular areas of improvement, which needed to be addressed by the proposed solution approach. First, the improvement of efficiency of the process, due to a lower level of complexity and effort. Second, the improvement of

efficiency of the process, due to a higher level of automation, and third, the effectiveness of quality assurance.

To underpin the expected improvements of the solution approach, several *KPIs* were proposed for the evaluation in Section 3.2 and further refined in Section 7.3.2. The *KPIs* relevant to answer the first area of *RI-1*, as mentioned in the paragraph above, are *KI-1.1* as well as *KI-2.1* and *KI-2.2*. *KI-1.1* measured the complexity of the *Engineering Process Selection and Configuration* regarding activities that had to be performed manually during the phase. While this indicator was primarily meant to measure complexity, the author found during the evaluation, that it also represents a metric for effort, which has to be invested into the configuration process and that the indicator is very similar to *KI-2.1*. *KI-2.1* directly aimed at the measurement of the working effort in time that the configuration process took to run through it once. *KI-2.2* measured the impact of changes in the underlying engineering processes and software components to the different approaches.

During the evaluation of the solution approach, which was executed on the *Signal Change Management Process (SCMP)* as an example, the author recorded the raw data for the *KPIs* to later calculate their values. Table 7.2 shows the raw data for the indicator *KI-1.1* for each, the traditional and the improved approach. Table 7.3 shows the calculated values of the indicators for the approaches and, furthermore, the first and every other run of the approaches. Table 7.4 and Table 7.5 show the values for *KI-2.1* and *KI-2.2*.

The results of *KI-1.1* and *KI-2.1* reveal that the preparation of the engineering process templates and the creation of the feature model are complex and time-consuming, as these tasks took 174 manual activities respectively approximately 16.5 hours to complete. However, the indicators also reveal, and this is even more important, that after this initial preparation and creation phase, which is needed for the improved process only, for each additional iterations of the process, a significant amount of time is saved and the complexity of the process stays at a much lower level. This decrease can obviously be seen, if the values of *KI-1.1* and *KI-2.1* for the traditional approach and the improved approach, after the first run, are set in relation. While the traditional approach required 95 manual activities to be performed – *KI-1.1*, the improved approach required only 32 manual activities to be performed, which is roughly a third of the steps. The time needed for the configuration process to go through once – *KI-2.1* – was 125 minutes for the traditional approach, in contrast to 25 minutes for the improved process, after the first iteration.

As mentioned before, *KI-1.1* can also be interpreted as effort, as every manual activity means an invested amount of time as well. If the indicator is interpreted this way, it can be easily expressed after how many iterations the improved process is superior to the traditional one. This relation is expressed and shown in Figure 8.1. The `x-axis` shows the number of iterations that the configuration process is performed. The `y-axis` displays the number of manual steps that are performed. `TA` labels the function of the traditional approach and `IA` labels the function of the improved approach. It can be

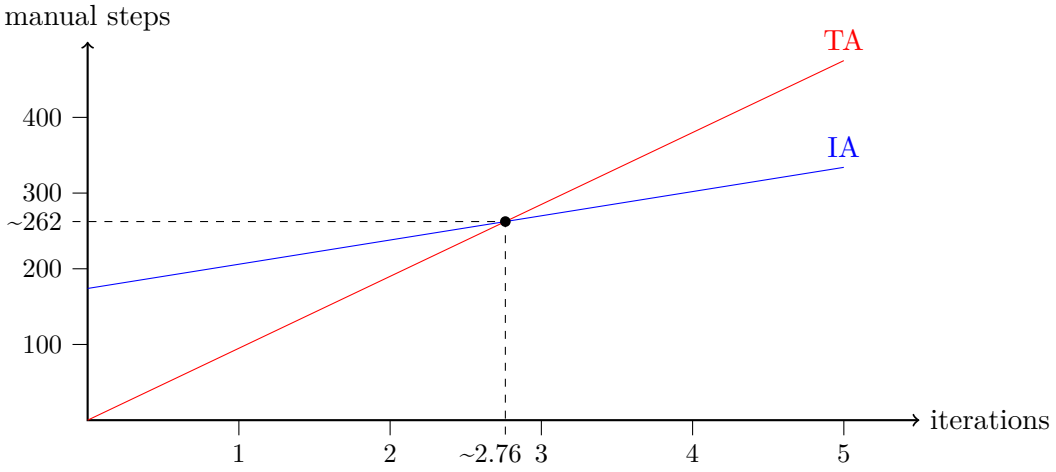

Figure 8.1: Development of *manual steps* over configuration *iterations*

seen, that the improved process, supported by the prototype, pays off after a little less than three customization iterations.

An important part to show, that the proposed solution improves the *Engineering Process Selection and Configuration* considering a raised efficiency, is that the introduction of variability models must not over-proportionally increase the maintenance cost in relation to the savings from the decreasing complexity and effort. Table 7.5 shows and compares three different examples and sums up how many steps were needed to adapt the 'solution set' for the configuration process in either case of the traditional and the improved approach. By 'solution set' the author understands, in case of the traditional approach the engineering process regardless whether it was already *cloned* or adapted in place of the customized application as well as the collection of services in the platform. In case of the improved approach, the author means the adaptation of the engineering process templates, the feature model, and the mapping table. *KIK-2.2 a* is equal for both approaches and thus not discussed any further. *KI-2.2 b* – the addition of a single service to the platform – and *KI-2.2 c* – the addition of an engineering process task, an interface, and two corresponding service implementations – require more steps to be performed in the improved as in the traditional approach. However, if these numbers are compared to the decreased steps in the configuration process, the comparison shows, that the traditional approach outnumbers the improved approach in steps and is hence inferior. Table 7.5 shows that for the adaptation of the solution set 3 steps in the traditional approach and 8 steps in the improved approach are needed. In contrast, using the traditional approach during the configuration process following the description in Section 7.3, it needs two steps to search the demanded service, one step to find the correct service call and one step for the configuration of the process task. Summed up, these are 4 additional steps, which need to be performed in the configuration process in comparison to the improved approach. Once again, *KI-2.2* is defined as the number of

steps needed to adapt the underlying solution set. The values for *KI-2.2 c* – the worst case – in relation to the additional steps that are needed during configuration shows – similar to *KI-1.1* – that the improved approach outperforms the traditional one after about three iterations.

In consideration of the second area, which covered the improvement due to a higher level of automation, Table 7.3 *KI-1.2* shows, which compared to the traditional approach the improved approach offers an essential higher level of automation. In the traditional approach, no automated tasks were performed. In the improved approach for the first iteration, which also includes the preparation of the solution set, *KI-1.2* shows, that about a quarter of all steps in the configuration process was automated. Even better, for the iterations after the first one, approximately two-thirds of all steps are automated.

The third area of improvement, that was motivated in Chapter 3, and takes the quality mechanisms into account, that can be realized based on the two approaches. While *KI-3.1* examines the number of quality assurance mechanisms that are implemented out-of-the-box, *KI-3.2* represents the number of quality assurance mechanisms that can easily be implemented to support the approach. For the traditional approach, it can be seen, that no quality assurance mechanisms whatsoever where implemented – see Table 7.6 *KI-3.1*. The quality assurance in the assembly process so far concentrated on the experience of the application integrators and integration tests, that where run after the customization. It must, nevertheless, be mentioned, that the customization of such an integrated tool application often lasted over several months and was frequently but manually checked. The improved approach provided the automated validation of the configuration as an automated mechanism for quality assurance, which is only a drop in the bucket of what would be favored.

If *KI-3.2* is investigated, it shows, that for the traditional approach the possibility to implement additional quality assurance mechanisms is limited. For the improved approach at least several extension points were found that support the application of techniques to check the correctness of the intermediate as well as the final results of the *Engineering Process Selection and Configuration*.

To summarize, the *KPIs* and their interpretation unquestionably show that the proposed solution approach, improves the *Engineering Process Selection and Configuration* to a great extent. This holds true if the approach is systematically applied, as the effort of preparation of the engineering process templates and creation of the feature model should not be underestimated. However, after only about three platform customizations on the same solution set, the usage of the solution approach pays off.

8.1.2 *RI-3* – Variability modeling for processes and software services

In the context of the discussion, *RI-3* is answered before *RI-2* as the solution approach for the latter one is based on the answer to this research issue.

RI-3 investigated how concepts of variability modeling can be utilized for modeling engineering processes and software systems to allow mapping between variants of each of

these models. To answer this question, it is split into two questions. The first question considers the software systems that are represented by the services and components of the tool integration platform. The second considers the engineering processes that exist in the engineering companies to organize the work.

The following two paragraphs, try to discuss and answer the first of these questions. At the beginning of this question the first sub-question of *RI-1*, which parts of the services vary between the different customizations of the tool integration platform, needed to be answered. The author found, that the services in the platform are distinguished based on which interfaces they implement. This distinction can be made because in *OSGi (OSGi)* service providers are discovered and resolved based on the interfaces they implement and also export to provide certain functionality. In an *OSGi* container, this means the following. The interface bundles are deployed as independent components. The service providers implemented the functionality of these interfaces and exported them by themselves. The container then resolved these dependencies and automatically linked the service providers to the service consumers. This concept was also applied in the different customizations of the tool integration platform. The service interfaces were deployed to the integrated tool applications, and then the service implementations, that fit the specific problem of the customer were deployed to the customization too.

Having these occurrences of variability in the tool integration platform in mind, the author investigated related work and discovered, that feature models from the *Feature Modeling (FM)* approach provide variability mechanisms that fit this pattern. *FM* as an approach to model variability in systems, provided as a method to model variability an artifact called feature models. This concept fits well because, feature models provide the means to describe the commonalities and the variabilities of systems, which is very similar to the services of the platform that share the interface as a description of the functionality, but vary in methods or implementation. This work, therefore, picked up the concepts of the feature model and adapted them to its needs, as *FM* itself provides a limited explanation on what should be modeled how. This answers the second sub-question of *RI-1*. The author defined for this work that the service interfaces are modeled as abstract features in the feature model. Furthermore, is was defined, that service implementations are modeled as concrete features that are descendant features of the features that represent their implemented interfaces. Also, the solution approach assumes, that the dependencies between the software components are modeled as constraints in the feature model. From these rules a feature model for tool integration platforms based on the *Engineering Service Bus (EngSB)* approach was derived and used in the solution approach.

Considering the second question, raised above, this work examined the engineering processes and their variability as they occur at industry partners. Especially the *SCMP* as engineering process of *round-trip engineering (RTE)* was taken into consideration, as it is used in several variations at the industry partner and was already discussed by Winkler et al. [74]. Picking up the first sub-question of *RI-1*, which components between customizations vary, the author identified, that most of the time either engineering sub-processes differ or, that services called from the engineering processes vary. Based

on these findings, a simple variability model for engineering processes based on *Business Process Model and Notation (BPMN)* was proposed. Therefore, the engineering processes were first modeled in *BPMN* separating the sub-processes from their main processes and separating them from each other by creating minimally viable sub-processes. These process fragments were used in the further course as engineering process templates and together build a catalog of engineering process templates. The calls to sub-processes and services within the *BPMN* process templates were replaced by placeholders that followed a specific convention. Call activities got placeholders with a `template` keyword prefixed that refer to a certain set of sub-process templates. Process tasks were enriched with placeholders that referred to the feature model and were prefixed with an `abstract` keyword.

The evaluation showed, that both of these variability models can be used, to develop a mapping mechanism to increase the efficiency and quality of the *Engineering Process Selection and Configuration* as part of the *CCP*.

8.1.3 *RI-2* – Mapping of process variants to software variants

RI-2 aimed to investigate and answer how variants of engineering processes can be used in the *Engineering Process Selection and Configuration* phase together with software service variants, to create fully configured engineering processes. In the solution approach, the author described a method, how *BPMN* process templates can be selected and afterward be mapped and configured to refer to service instances that are modeled in a *feature model*. To answer the question in detail, it is split up into three parts: a) how can engineering process variants be selected with tool support, b) how can the selected engineering process variants be mapped to software service variants, and c) how can the resulting engineering process variants be configured.

To answer part *a* of the question, the proposed approach from Section 6.2 is recalled. As described, before a first iteration of the selection process can be performed, the engineering process catalog, containing the process templates, needs to be prepared. The selection of the engineering process templates is based on an addressing convention, which utilizes the concepts defined for the process variability model, that allows a tool to locate the templates. While a very simple convention is proposed in this work to show the feasibility, a more sophisticated convention is planned for future work. A tool that supports the convention, in this case, the prototype, reads in the template catalog and presents the top-level processes to an application integrator, who selects them. In a subsequent activity, the tool goes through all selected top-level templates, resolves the referenced sub-process templates and offers them to the application integrator for further selection.

The concepts to answer part *b* of the question above, were presented in Section 6.4 and will be recapped in this paragraph. The feature model for the software services of the tool integration platform contains abstract and concrete features. While abstract features refer to interface bundles of the platform, concrete features refer to service implementations. In

the engineering process templates the process tasks were enriched with placeholders, which referred to the abstract features of the feature model. The mapping of the engineering process variants to the feature model was proposed as follows. The placeholders from the engineering processes are collected and written into a basic configuration. This basic configuration is loaded into the *FeatureIDE* and completed an application integrator in the configuration editor of the *FeatureIDE* with the concrete features. In the background, the configuration is continuously validated for correctness according to the constraints of the corresponding feature model. The result is configuration that mapped the process tasks to the concrete features, which on their side refer to particular service implementations.

Part *c* of the question is directly tied to part *b*. From the completed and validated configuration, resulting from the previous step, the concrete features are taken as input for the search of their abstract feature ancestors in the XML representation of the feature model. The values of these abstract ancestors are searched within the engineering processes that were selected before and changed to process task calls, which directly refer to the correct service calls. These service calls are again read from a mapping table, that contains the concrete feature names and their service call reference. The resulting configured engineering processes as *BPMN* files are then stored at a location desired by the application integrator for further use.

To quickly summarize, the last three paragraphs described how variants of engineering processes are selected, mapped to software component variants and configured with the correct service calls. The evaluation based on the proposed solution approach and supported by a prototype demonstrated, that the described method is feasible and outperforms the traditional approach of the configuration process.

Finally, one question needs to be answered for *RI-2*, that asks which parts of the *Engineering Process Selection and Configuration* still need to be done manually and which parts were automated. Table 7.2 shows that the following tasks need to be performed manually: 1. the process selection, which, however, is supported by the prototype that searches for process templates 2. the selection of the sub-process templates, which is now also supported by the prototype 3. the aggregation of the sub-process templates to a combined sub-process, and 4. the service selection during the configuration, which is also supported by the configuration editor. On the other hand, the following activities were automated by the solution approach: 1. the search for the services and their alternative services 2. the resolution of the constraints within the tool integration platform, and 3. the task configuration of the process tasks in the engineering processes. While this might not sound as the *Engineering Process Selection and Configuration* was much improved, the evaluation shows that, although the engineering process templates need to be prepared and the feature model needs to be created, the solution approach significantly improved the traditional approach.

Nevertheless, the proposed and tested solution approach still has a couple of limitations, that will be discussed in the next section.

8.2 Limitations

During the course of the thesis, several limitations of the approach, the prototype, and the evaluation were found, which are brought up in the following paragraphs. These limitations could not be covered in the restricted scope of this thesis. However, they can be addressed in future work (see also Section 9.2 - 'Future Work').

8.2.1 Limitations of the Evaluation Procedure

As mentioned in Section 7.3.2 - 'Measurement of KPIs' the raw data for the *KPIs* was recorded by the author by hand. This was mainly done because the traditional approach is manually performed and does not allow the implementation of an automated step-counting mechanism. Furthermore, the *CCP* was only documented as text without certain checklists, which could have helped to provide a stricter protocol. Although the author thoroughly conducted the evaluation and counter-checked the results, the threat of minors errors, which is a threat to validity, can not be eliminated. However, the author is aware of this and proposes at least a supervised measurement of the *KPIs* and an adaptation of the prototype to automatically count the values for the improved approach.

The improved solution approach was also only tested on the *SCMP* as a single engineering process. The *SCMP* is a process of medium complexity, that uses quite simple sequence flows. Nevertheless, was directly derived from a real-world example at an industry partner. In the discussion of this thesis, the author shows, that the proposed solution approach after several iterations of the *CCP* is in principle superior to the traditional approach. However, the author cannot guarantee, that for engineering processes of much higher complexity or engineering processes that are modeled very differently, and use, for example, extended message flows, the same holds true.

Furthermore, the evaluation was solely executed and tested on a single customized version of the *AutomationML Hub (AML.hub)* as there were limited customized integrated tool applications available. This as well represents a threat to validity for the evaluation results and their interpretation. An evaluation of the prototype and the solution approach on a broader base would back the findings of this thesis.

8.2.2 Manual creation of the *Feature Model*

As mentioned in Section 7.1.1, the feature model proposed as variability model for the service components of the tool integration platform needs to be created in advance to the execution of the proposed approach. From the values of the *KPIs* introduced in Section 3.2 and detailed in Section 7.3.2 it can certainly be seen that this part of the improved approach is by far the one that is associated with the most effort and complexity. On top of the effort, the preparation of the feature model is the part with the highest probability of errors as it is performed manually. Even a semi-automated creation of the feature model based on the software components of the underlying tool integration platform would largely decrease the manual effort for the solution approach.

8.2.3 Variability of *Engineering Processes*

The proposed variability model for engineering processes is a very simplistic approach, that takes only the most important types of process variability identified in Schnieders [62] and Schnieders and Puhlmann [63] into account (see also Section 6.1.2). Several other sources of variability can be found in engineering and business processes, which come up in real-world scenarios. This issue impedes the applicability of the solution approach to a broader field of adaptations of the engineering processes. The proposed solution approach would benefit from a more sophisticated variability model based on *BPMN*. Second, the manual combination of the engineering process templates, as it is performed in this work, can be tedious and is also a source of error. At least a mechanism to better assure the quality of the resulting engineering process variants would help the approach to gain relevance for the industry. If parts of the adaptation could be executed based on predefined rules, it would be even better for consistent results of this step of the *CCP*.

8.2.4 Engineering Process Catalog

For the engineering process templates the creation and usage of a structured catalog was motivated in Section 6.1 - 'Process Variability Model'. For the evaluation, described in Section 7.1.2 the catalog was utterly created using the file system. In a real-world scenario, this would have several limitations, like the accessibility in a distributed setting, the traceability of changes and the structured revisioning of process templates, to name a few. However, several technologies and methods exist to overcome such issues and create a better system for a structured catalog of process blueprints that is properly addressable.

8.2.5 Restrictions due to limited Workflow Engine integration

A goal of the thesis was to improve the *Engineering Process Selection and Configuration* phase to generate fully configured engineering processes. As mentioned before, fully configured in this case means, that the engineering processes are correctly configured with service calls of the same services, which are deployed in the integrated tool application. Ideally, these engineering processes would have been deployed to the integrated tool application, in this case, the *AML.hub*, and executed in the workflow engine within this application. Due to the limited integration of the workflow engine in the *AML.hub*, the configured engineering processes were not be executed within the customized environment. This would contribute further to show, that the improved configuration process is superior to the traditional approach. An execution of the generated and configured engineering processes would additionally increase the quality of the resulting engineering process variants.

CHAPTER 9

Conclusion and Future Work

In the previous chapter, the evaluation from Chapter 7 was discussed in consideration of the research issues introduced in Chapter 3 and the findings presented. Furthermore, in the previous chapter, the author described the limitations of the solution approach and the prototype, that still exist.

This chapter draws a conclusion out of the discussion and findings from Chapter 8 and presents, based on this thesis, future work that needs to be done.

9.1 Conclusion

Large-scale projects in *production systems engineering (PSE)*, like the planning of hydropower plants, are set in a multidisciplinary tool environment. In such projects, engineers of diverse domains work together in a combined effort to perform their engineering tasks. In such environments, various specialized tools deeply-seated in the different domains determine the engineering processes [11]. Unfortunately, these domain-specific tools often have quite limited data connectivity capabilities. Project consistency, in such settings, therefore, requires a continuous artifact exchange [75]. While, the *Engineering Service Bus (EngSB)* [5] provides a generic approach to integrate engineering tools and processes seamlessly, this integration platform requires tailoring to customer specific needs. However, this *Customization and Configuration Process (CCP)* is tedious and error-prone manual work that has to be performed by experts.

This work proposed an approach to improve the traditional and manual approach of *Engineering Process Selection and Configuration* by utilizing concepts of *Variability Modeling*, especially *Feature Modeling (FM)*, and *Process Modeling*, especially *Business Process Model and Notation (BPMN)*. These concepts were utilized for engineering processes and the software services of tool integration platforms to define a *Process Variability Model* and a *Platform Variability Model*. The introduced models were then

used to propose an approach, which enables a (semi)automated selection of engineering process templates to build engineering process variants, their mapping to software service variants and, finally, the configuration of these engineering process variants to deployable process descriptions. The approach now allows to map engineering process blueprints in *BPMN* to service variants, that referenced in a feature model.

In an evaluation based on a real-world use case from an industry partner and using a prototype developed for this purpose, the solution approach was examined. The tested use case is one variant of the *Signal Change Management Process (SCMP)* [74], which is part of *round-trip engineering (RTE)* [64], a cyclic development process for iterative refinement, that is common practice in engineering. The evaluation in this thesis showed that the proposed method for engineering process and service mapping is *feasible*. Furthermore, based on the evaluation results, which were measured for the above-mentioned sample, the method showed *significant better* results than the traditional approach of customization and configuration. This mainly concerns the decreased complexity and amount of working effort for the *Engineering Process Selection and Conclusion* phase, that kicks in after the first iteration of the configuration process. The first iteration is, however, burdened by the preparation of the engineering process templates and the creation of the feature model. The second improvement concerns the far better rate between automation and the increased number of extension points for quality assurance mechanisms, which further proofs the increase of efficiency and the quality due to the solution approach. It is expected, that for larger and more complex examples, the solution approach works even better than for the already evaluated ones.

Nevertheless, due to the limited scope of the work, several topics remained as future work, which will be outlined in the next section.

9.2 Future Work

The following sections discuss future work, that result from the limitations outlined in Section 8.2.

9.2.1 Case Study with the Solution Approach

In Section 8.2, the author mentioned the limitation of the evaluation procedure, due to the examination of the solution approach with the prototype just for a single engineering process and a single customized integrated tool application. Furthermore, the author mentioned, that the manual measurement of the raw data for the *Key Performance Indicators (KPIs)* is a threat to the validity of the thesis. Future work, for this limitation, would be a case study with a larger sample of engineering processes as well as a greater sample of customized applications. Also, a strict evaluation protocol should be developed that makes the traditional and the manual approach better comparable.

9.2.2 (Semi)automated Creation of the *Feature Model*

One limitation mentioned in Section 8.2 in the previous chapter, concerned the manual creation of the feature model for the tool integration platform. The discussion in the previous chapter showed that this task is very labor intensive and prone to errors. The partially automated creation of the feature model or parts of feature model would further support the solution approach as a great part of the manual work could be omitted. In Section 6.3 - ‘ Platform Variability Model’ the author mentioned that the *EngSB* uses Maven as build tool, that provides a sophisticated dependency management. Based on the dependencies between the services organized in Maven modules and Maven as a tool, an automated creation of the feature model, that is afterward completed by an engineer familiar with the tool integration platform to omit manual work is in the scope of future work.

9.2.3 Variability of *Engineering Processes*

The thesis, in its restricted scope, provided a simple variability model for engineering processes. A limitation is the manual aggregation of the sub-process templates to a combined sub-process which can be called with a single reference from the call activities. The threat is, that, because of missing aggregation rules, engineering processes are created very differently. A survey by Rosa et al. [58] investigated the topic of *Business Process Families*, which introduces different ways to model sets of processes that similar to each other. Future work would also be, to improve the proposed variability model for engineering processes by using concepts of the survey.

9.2.4 Execution of Engineering Processes

The limitations discussed in Chapter 8 mentioned that, while the engineering processes resulting from the improved approach are fully configured in the sense of the thesis, the engineering processes were not executed in the integration tool application due to the limited implementation of a *BPMN* workflow engine in the *AutomationML Hub (AML.hub)*. An implementation of, for example, the Camunda workflow engine into the *AML.hub* and the execution of the engineering processes resulting from customization, further proof the validity of the solution approach.

9.2.5 Storage of the Process Blueprints in Repositories

The evaluation described in Chapter 7 used the file system to store the engineering process template catalog. This raised some issues for the usage in a production environment such as traceability of changes. Modern software engineering used source code repositories, on the one hand, and binary artifact repositories, on the other hand. The utilization of a binary artifact repository such as Artifactory[1] as engineering process variant catalog would be an interesting topic to investigate.

[1] JFrog Artifactory – `https://jfrog.com/artifactory/`

9.2.6 Implementation of Quality Assurance Mechanisms

In the evaluation of the solution approach and the investigation of *KI-3.2*, this thesis shows possible extension points for quality assurance mechanisms. In the future, it would be possible to investigate these extension points and implement quality assurance mechanisms that fit these extension points to improve the improved approach further.

List of Figures

List of Tables

Listings

Acronyms

AML AutomationML. 15, 46, 47, 49, 52, 62, 66, 74

AML.hub AutomationML Hub. 15, 45–47, 52, 55, 62–64, 74, 80, 81, 87, 96, 97, 101

API Application Programming Interface. 14, 15

ASB Automation Service Bus. 13

BMNS Business Process Management System. 9

BPEL Business Process Execution Language. 19

BPM Business Process Management. 8, 9, 40, 41

BPMI Business Process Management Initiative. 19

BPMN Business Process Model and Notation. 9, 15, 19–23, 55–60, 67–71, 75–78, 80, 94, 95, 97, 99–101, 103

CCP Customization and Configuration Process. 3–8, 24, 31–38, 43, 50, 52, 54, 55, 57, 59–61, 66, 69–73, 75, 77, 80, 83, 88, 89, 94, 96, 97, 99

CID Continuous Integration and Deployment. 33

CSV Comma Separated Value. 2, 47, 48

CVL Common Variability Language. 24

DM Decision Modeling. 24, 27–29, 61, 62

DOPLER Decision-Oriented Product Line Engineering for effective Reuse. 29

EMF Eclipse Modeling Framework. 47

EngSB Engineering Service Bus. 2, 3, 5, 7, 13–15, 31, 32, 45, 53, 54, 61, 62, 64, 67, 73, 93, 99, 101, 103

EPC Event-driven Process Chain. 17, 18, 55

ESB Enterprise Service Bus. 2, 13, 14, 53

FM Feature Modeling. 9, 24, 27, 29, 61, 62, 71, 93, 99

FODA Feature-Oriented Domain Analysis. 24, 25

FOP Feature-Oriented Programming. 26

FOSD Feature-Oriented Software Development. 26

JAR JAVA archive. 63

KKS Kraftwerk-Kennzeichensystem. 45–47, 49

KPI Key Performance Indicator. 36, 37, 43, 80, 82, 83, 85, 86, 88–90, 92, 96, 100

MDSE Model Driven Software Engineering. 8, 32

MOF Meta Object Facility. 18

OASIS Organization for the Advancement of Structured Information Standards. 19

OCL Object Constraint Language. 18

OESB Open Engineering Service Bus. 2, 15, 46, 62

OMG Object Management Group. 18, 19, 24

OSGi OSGi. 2, 15, 46, 55, 63, 64, 67, 74, 93, 104

PLE Product Line Engineering. 23

PSE production systems engineering. 1, 2, 11–13, 15, 31, 43–45, 51, 53, 59, 99

PTP point-to-point. 11–13, 103

RTE round-trip engineering. 44, 45, 47, 48, 51–53, 73, 93, 100

SCMP Signal Change Management Process. 48, 50, 58, 59, 73, 74, 80, 81, 84, 87, 90, 93, 96, 100, 104

SPF Software Product Family. 7

SPL Software Product Line. 7–9, 23, 24, 26, 27

SPLE Software Product Line Engineering. 23, 24

UML Unified Modeling Language. 18, 55

VM Variability Modeling. 7–9, 23, 24, 35, 40, 41

WfMC Workflow Management Coalition. 19

XMI XML Metadata Interchange. 18, 47

XPDL XML Process Definition Language. 19

Bibliography

[1] S. Adam, N. Riegel, T. Jeswein, M. Koch, and S. Imal. Studie-BPM Suites 2013. *Fraunhofer IESE, SP Consulting GmbH*, 2013.

[2] C. Ayora, V. Torres, V. Pelechano, and G. H. Alférez. Applying CVL to Business Process Variability Management. In *Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone*, VARY '12, pages 26–31, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1809-9. doi: 10.1145/2425415.2425421. URL http://doi.acm.org/10.1145/2425415.2425421.

[3] S. Biffl and A. Schatten. A Platform for Service-Oriented Integration of Software Engineering Environments. In *Proceedings of the 2009 conference on New Trends in Software Methodologies, Tools and Techniques (SoMeT 09)*, pages 75–92, Pargue, CZE, 2009. IOS Press. URL http://dl.acm.org/citation.cfm?id=1659308.1659316.

[4] S. Biffl, R. Mordinyi, and A. Schatten. A Model-Driven Architecture Approach Using Explicit Stakeholder Quality Requirement Models for Building Dependable Information Systems. In *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*, Los Alamitos, CA, USA, 2007. IEEE Computer Society. ISBN 0-7695-2959-3. doi: 10.1109/WOSQ.2007.1.

[5] S. Biffl, A. Schatten, and A. Zoitl. Integration of heterogeneous engineering environments for the automation systems lifecycle. In *2009 7th IEEE International Conference on Industrial Informatics*, pages 576–581. IEEE, June 2009. ISBN 978-1-4244-3759-7. doi: 10.1109/INDIN.2009.5195867. URL http://ieeexplore.ieee.org/ielx5/5175248/5191764/05195867.pdf?tp=&arnumber=5195867&isnumber=5191764http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5195867.

[6] S. Biffl, R. Mordinyi, and T. Moser. Automated Derivation of Configurations for the Integration of Software(+) Engineering Environments. In *Proceedings of the 1st International Workshop on Automated Configuration and Tailoring of Applications (ACoTA 2010)*, pages 6–13, 2010. URL http://publik.tuwien.ac.at/files/PubDat_187568.pdf.

[7] S. Biffl, R. Mordinyi, and T. Moser. Integriertes Engineering mit Automation Service Bus - Paralleles Engineering mit heterogenen Werkzeugen. *atp edition*, 12:36–43, 2012.

[8] S. Biffl, R. Mordinyi, and T. Moser. Anforderungsanalyse für das integrierte Engineering - Mechanismen und Bedarfe aus der Praxis. *atp edition*, 5:28–35, 2012.

[9] J. Bosch. Software Product Line Engineering. In R. Capilla, J. Bosch, and K.-C. Kang, editors, *Systems and Software Variability Management SE - 1*, pages 3–24. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-36582-9. doi: 10.1007/978-3-642-36583-6_1. URL http://dx.doi.org/10.1007/978-3-642-36583-6_1.

[10] M. Broy. Software Quality: From Requirements to Architecture. *Software Quality. Increasing Value in Software and ...*, pages 1–2, 2013. ISSN 18651348. doi: 10.1007/978-3-642-35702-2_1. URL http://link.springer.com/chapter/10.1007/978-3-642-35702-2_1.

[11] M. Broy, M. Feilkas, M. Herrmannsdoerfer, S. Merenda, and D. Ratiu. Seamless model-based development: From isolated tools to integrated model engineering environments. *Proceedings of the IEEE*, 98(4):526–545, 2010. ISSN 00189219. doi: 10.1109/JPROC.2009.2037771.

[12] S. N. Cant, D. R. Jeffery, and B. Henderson-Sellers. A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology*, 37(7):351–362, 1995. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/0950-5849(95)91491-H. URL http://www.sciencedirect.com/science/article/pii/095058499591491H.

[13] J. Cardoso. Approaches to Compute Workflow Complexity. In F. Leymann, W. Reisig, S. R. Thatte, and W. van der Aalst, editors, *The Role of Business Processes in Service Oriented Architectures*, volume 06291 of *Dagstuhl Seminar Proceedings*, pages 16–21, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum f{ü}r Informatik (IBFI), Schloss Dagstuhl, Germany. ISBN 1862-4405. URL http://drops.dagstuhl.de/opus/volltexte/2006/821/.

[14] J. S. Cardoso. Business Process Control-Flow Complexity: Metric, Evaluation, and Validation. *Int. J. Web Service Res.*, 5(2):49–76, 2008. doi: 10.4018/jwsr.2008040103. URL https://doi.org/10.4018/jwsr.2008040103.

[15] J. S. Cardoso, J. Mendling, G. Neumann, and H. A. Reijers. A Discourse on Complexity of Process Models. In J. Eder and S. Dustdar, editors, *Business Process Management Workshops, {BPM} 2006 International Workshops, BPD, BPI, ENEI, GPWW, DPM, semantics4ws, Vienna, Austria, September 4-7, 2006, Proceedings*, volume 4103 of *Lecture Notes in Computer Science*, pages 117–128. Springer, 2006. ISBN 3-540-38444-8. doi: 10.1007/11837862_13. URL https://doi.org/10.1007/11837862{_}13.

[16] D. A. Chappell. *Enterprise Service Bus*. O'Reilly, 2004. ISBN 978-0-596-00675-4.

[17] M. Chinosi and A. Trombetta. BPMN: An introduction to the standard. *Computer Standards & Interfaces*, 34(1):124–134, 2012. ISSN 0920-5489. doi: https://doi.org/10.1016/j.csi.2011.06.002. URL http://www.sciencedirect.com/science/article/pii/S0920548911000766.

[18] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski. Cool Features and Tough Decisions : A Comparison of Variability Modeling Approaches. *6th Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMos'12)*, pages 173–182, 2012. doi: 10.1145/2110147.2110167.

[19] D. Dhungana, P. Grünbacher, and R. Rabiser. The DOPLER meta-tool for decision-oriented variability modeling: A multiple case study. *Automated Software Engineering*, 18(1):77–114, 2011. ISSN 09288910. doi: 10.1007/s10515-010-0076-6.

[20] R. Drath. *Datenaustausch in der Anlagenplanung mit AutomationML*. Springer-Verlag Berlin Heidelberg, 1 edition, 2010. ISBN 9783642046735. doi: 10.1007/978-3-642-04674-2.

[21] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[22] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[23] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 391–400, sep 2014. doi: 10.1109/ICSME.2014.61.

[24] M. Fowler and M. Foemmel. Continuous integration. *Thought-Works) http://www.thoughtworks.com/Continuous Integration.pdf*, 122, 2006.

[25] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. In R. Capilla, J. Bosch, and K.-C. Kang, editors, *Systems and Software Variability Management SE - 1*, pages 253–260. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-36582-9. doi: 10.1007/978-3-642-36583-6. URL http://link.springer.com/10.1007/978-3-642-36583-6.

[26] R. Heidel. Industrie 4.0: Ohne Normung geht es nicht. IEC TC 65: Industrial-process measurement, control and automation. *Blomberg*, oct 2014.

[27] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design Science in Information Systems Research. *Design Science in IS Research MIS Quarterly*, 28(1):75–105, 2004. ISSN 02767783. doi: 10.2307/25148625.

[28] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321200683.

[29] IEC. IEC 62714-1:2014 - Engineering data exchange format for use in industrial automation systems engineering - Automation markup language - Part 1: Architecture and general requirements, 2014. URL https://webstore.iec.ch/publication/7388. [Online; accessed 2017-10-05].

[30] IEC. IEC 62714-2:2015 - Engineering data exchange format for use in industrial automation systems engineering - Automation markup language - Part 2: Role class libraries, 2015. URL https://webstore.iec.ch/publication/22030. [Online; accessed 2017-10-05].

[31] IEC. IEC 62714-3:2017 - Engineering data exchange format for use in industrial automation systems engineering - Automation markup language - Part 3: Geometry and kinematics, 2017. URL https://webstore.iec.ch/publication/34158. [Online; accessed 2017-10-05].

[32] ISO. ISO 9000:2015 - Quality management systems - Fundamentals and vocabulary, 2015. URL https://www.iso.org/obp/ui/#iso:std:iso:9000:ed-4:v1:en. [Online; accessed 2017-10-05].

[33] ISO. ISO 9001:2015 - Quality management systems - Requirements, 2015. URL https://www.iso.org/obp/ui/#iso:std:iso:9001:ed-5:v1:en. [Online; accessed 2017-10-05].

[34] K. C. . Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report November, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

[35] K. C. Kang and H. Lee. Variability modeling. In *Systems and Software Variability Management*, pages 25–42. Springer, 2013.

[36] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: A Tool Framework for Feature-oriented Software Development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 611–614, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070568. URL http://dx.doi.org/10.1109/ICSE.2009.5070568.

[37] K. Kruczynski. Business process modelling in the context of soa–an empirical study of the acceptance between epc and bpmn. *World review of science, technology and sustainable development*, 7(1-2):161–168, 2010.

[38] S. T. March and G. F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251–266, 1995. ISSN 01679236. doi: 10.1016/0167-9236(94)00041-2.

[39] R. McCabe, G. Campbell, N. Burkhard, S. Wartik, J. O'Connor, J. Valent, and J. Facemire. Reuse-driven software processes guidebook. techreport SPC-92019-CMC, Version 02.00.03, Software Productivity Consortium, Nov. 1993.

[40] J. Mendling, G. Neumann, and M. Nüttgens. A Comparison of XML Interchange Formats for Business Process Modelling. In *EMISA*, volume 56, pages 129–140, 2004.

[41] R. Mordinyi, T. Moser, E. Kühn, S. Biffl, and A. Mikula. Foundations for a Model-Driven Integration of Business Services in a Safety-critical Application Domain. In *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009)*, pages 267–274. IEEE Computer Society, 2009. ISBN 978-0-7695-3784-9. doi: 10.1109/SEAA.2009.19. URL http://publik.tuwien.ac.at/files/PubDat_179464.pdf.

[42] T. Moser and S. Biffl. Semantic Integration of Software and Systems Engineering Environments. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(1):38–50, jan 2012. ISSN 1094-6977. URL http://dblp.uni-trier.de/db/journals/tsmc/tsmcc42.html{#}MoserB12.

[43] T. Moser, R. Mordinyi, A. Mikula, and S. Biffl. Efficient Integration of Complex Information Systems in the ATM Domain with Explicit Expert Knowledge Models. In *Complex Intelligent Systems and Their Applications*, pages 1–19. Springer-Verlag, New York, 2010. ISBN 978-1-4419-1635-8. doi: 10.1007/978-1-4419-1636-5\{_\}1. URL http://publik.tuwien.ac.at/files/PubDat_192639.pdf.

[44] OMG. Business Process Model and Notation (BPMN), Version 2.0, Jan. 2011. URL http://www.omg.org/spec/BPMN/2.0/. [Online; accessed 2017-10-05].

[45] OMG. Common Variability Language (CVL), OMG Revised Submission, Aug. 2012. URL http://www.omgwiki.org/variability/lib/exe/fetch.php?media=cvl-revised-submission.pdf. [Online; accessed 2017-10-05].

[46] OMG. Object constraint language (ocl), version 2.4, Feb. 2014. URL http://www.omg.org/spec/OCL/2.4/. [Online; accessed 2017-10-11].

[47] OMG. Unified modeling language (uml), version 2.5, Mar. 2015. URL http://www.omg.org/spec/UML/2.5/. [Online; accessed 2017-10-11].

[48] OMG. Xml metadata interchange (xmi), version 2.5.1, June 2015. URL http://www.omg.org/spec/XMI/2.5.1/. [Online; accessed 2017-10-11].

[49] OMG. Meta object facility (mof), version 2.5.1, Nov. 2016. URL http://www.omg.org/spec/MOF/2.5.1/. [Online; accessed 2017-10-11].

[50] OMG. Decision Model and Notation (DMN), Version 1.1, June 2016. URL http://www.omg.org/spec/DMN/1.1/. [Online; accessed 2017-10-05].

[51] OMG. A uml action language: Action language for foundational uml (alf), version 1.1, July 2017. URL http://www.omg.org/spec/ALF/1.1/. [Online; accessed 2017-10-11].

[52] OMG. Semantics of a foundational subset for executable uml models (fuml), version 1.3, July 2017. URL http://www.omg.org/spec/FUML/1.3/. [Online; accessed 2017-10-11].

[53] J. A. Pereira, S. Krieter, J. Meinicke, R. Schröter, G. Saake, and T. Leich. FeatureIDE: Scalable Product Configuration of Variable Systems. In *Proceedings of the 15th International Conference on Software Reuse: Bridging with Social-Awareness - Volume 9679*, ICSR 2016, pages 397–401, New York, NY, USA, 2016. Springer-Verlag New York, Inc. ISBN 978-3-319-35121-6. doi: 10.1007/978-3-319-35122-3_27. URL https://doi.org/10.1007/978-3-319-35122-3{_}27.

[54] M. Petritsch. Process and product analysis in multidisciplinary, heterogeneous engineering environments. Master's thesis, TU Wien, Mar. 2016.

[55] C. Prehofer. *Feature-oriented programming: A fresh look at objects*, pages 419–443. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. ISBN 978-3-540-69127-3. doi: 10.1007/BFb0053389. URL https://doi.org/10.1007/BFb0053389.

[56] J. Recker, J. Mendling, W. Van Der Aalst, and M. Rosemann. Model-driven enterprise systems configuration. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4001 LNCS:369–383, 2006. ISSN 03029743. doi: 10.1007/11767138_25.

[57] A. Reeve. *Managing data in motion: Data integration best practice techniques and technologies*. Morgan Kaufmann, 2013. ISBN 9780123971678.

[58] M. L. Rosa, W. M. P. Van Der Aalst, M. Dumas, and F. P. Milani. Business Process Variability Modeling: A Survey. *ACM Comput. Surv.*, 50(1):2:1—-2:45, mar 2017. ISSN 0360-0300. doi: 10.1145/3041957. URL http://doi.acm.org/10.1145/3041957.

[59] M. Rosemann. Potential pitfalls of process modeling: part A. *Business Proc. Manag. Journal*, 12(2):249–254, 2006. doi: 10.1108/14637150610657567. URL https://doi.org/10.1108/14637150610657567.

[60] A.-W. Scheer, O. Thomas, and O. Adam. Process Modeling Using Event-Driven Process Chains. In M. Dumas, W. M. P. Van Der Aalst, and A. H. M. Ter Hofstede, editors, *Process-Aware Information Systems: Bridging People and Software Through Process Technology*. Wiley, 2005. ISBN 978-0-471-66306-5.

[61] K. Schmid and I. John. A customizable approach to full lifecycle variability management. *Science of Computer Programming*, 53(3):259–284, 2004. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2003.04.002. URL http://www.sciencedirect.com/science/article/pii/S0167642304000929.

[62] A. Schnieders. Variability mechanism centric process family architectures. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06)*, pages 10 pp.–298, mar 2006. doi: 10.1109/ECBS.2006.72.

[63] A. Schnieders and F. Puhlmann. Variability Mechanisms in E-Business Process Families. In W. Abramowicz and H. C. Mayr, editors, *Business Information Systems, 9th International Conference on Business Information Systems, {BIS} 2006, May 31 - June 2, 2006, Klagenfurt, Austria*, volume 85 of *LNI*, pages 583–601, Klagenfurt, 2006. GI. ISBN 3-88579-179-X. URL `http://subs.emis.de/LNI/Proceedings/Proceedings85/article4299.html`.

[64] D. Schreiner. Component Based Communication Middleware for AUTOSAR. *Framework*, (9026735), 2009.

[65] Software Engineering Institute. Software product lines, May 2014. URL `https://www.sei.cmu.edu/productlines/`. [Online; accessed 2017-11-28].

[66] T. Thum, C. Kastner, S. Erdweg, and N. Siegmund. Abstract features in feature modeling. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 191–200. IEEE, 2011.

[67] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-oriented Software Development. *Sci. Comput. Program.*, 79:70–85, jan 2014. ISSN 0167-6423. doi: 10.1016/j.scico.2012.06.002. URL `http://dx.doi.org/10.1016/j.scico.2012.06.002`.

[68] W. M. P. Van Der Aalst and K. M. Van Hee. Business process redesign: A Petri-net-based approach. *Computers in industry*, 29(1):15–26, 1996.

[69] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distrib. Parallel Databases*, 14(1):5–51, jul 2003. ISSN 0926-8782. doi: 10.1023/A:1022883727209. URL `https://doi.org/10.1023/A:1022883727209`.

[70] VGB. Kks kraftwerk-kennzeichensystem, Jan. 2010. URL `https://www.vgb.org/shop/b105.html`. [Online; accessed 2017-10-11].

[71] S. Walraven and P. Verbaeten. AO middleware supporting variability and dynamic customization of security extensions in the ORB layer. *Companion '08 Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, pages 121 – 123, 2008. doi: 10.1145/1462735.1462771. URL `http://dl.acm.org/citation.cfm?id=1462771`.

[72] F. Waltersdorfer, T. Moser, A. Zoitl, and S. Biffl. Version management and conflict detection across heterogeneous engineering data models. In *2010 8th IEEE International Conference on Industrial Informatics*, pages 928–935, July 2010. doi: 10.1109/INDIN.2010.5549617.

[73] S. A. White. Process modeling notations and workflow patterns. *Workflow handbook*, pages 265–294, 2004.

[74] D. Winkler, T. Moser, R. Mordinyi, S. W. D. Sunindyo, and S. Biffl. Engineering object change management process observation in distributed automation systems projects. In *18th EuroSPI Conference*, 2011. URL http://www.ifs.tuwien.ac.at/~mordinyi/papers/2011eurospi.pdf.

[75] D. Winkler, S. Biffl, and H. Steininger. Integration von heterogenen Engineering Daten mit AutomationML und dem AML.hub: Konsistente Datenüber Fachbereichsgrenzen hinweg. *develop3 systems engineering*, (3):62–64, 2015.

[76] D. Winkler, R. Mordinyi, and S. Biffl. *Qualitätssicherung in heterogenen und verteilten Entwicklungsumgebungen für industrielle Produktionssysteme*, pages 259–278. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017. ISBN 978-3-662-53248-5. doi: 10.1007/978-3-662-53248-5_89. URL https://doi.org/10.1007/978-3-662-53248-5{_}89.