universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

Software Variant Management Through Process Modeling And Enactment Integration

verfasst von / submitted by

Tobias Pfaller, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2021 / Vienna 2021

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 926
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Wirtschaftsinformatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Math. oec. Dr. Stefanie Rinderle-Ma

Declaration of Authorship

I, Tobias Pfaller, BSc, declare that this thesis titled, "Software Variant Management Through Process Modeling And Enactment Integration" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I tried to be as clear as possible to describe what was done by others and what I have contributed myself.
- Where the thesis is based on work done by myself jointly with others, the concepts and ideas always are my contribution. Implementation details are sometimes contributed by others, if so, this fact is clearly stated and their contribution is acknowledged.

Signed: _____

Date: _____

UNIVERSITY OF VIENNA

Abstract

Faculty of Computer Science
Workflow Systems and Technology Group

Software Variant Management Through Process Modeling And Enactment Integration

by Tobias Pfaller, BSc

Software vendors are confronted by their customers with high customizing requirements. The widely used practice of cloning and modifying software leads to an increasing number of variants and thus to enormous maintenance difficulties in the long term. Therefore, this master thesis "Software Variant Management Through Process Modeling And Enactment Integration" is about exploring architectural changes to monolithic software in order to enable process-based model-driven configuration. For this purpose, a framework *Process-Based Mediation Configuration* (PBMC) was developed, consisting of the following stages: (1) Identification of variable components, (2) extraction of variable components, (3) augmentation with process technology, (4) use of a new pattern *Configurable Process Start* to choose the process variants suitable for different customers during instantiation.

To evaluate PBMC, (1) a sample process has been implemented in a real-world scenario (in cooperation with a company partner), (2) an independent evaluation with existing approaches has been conducted, and (3) a prototypical implementation has been developed.

UNIVERSITÄT WIEN

Kurzbeschreibung

Faculty of Computer Science
Workflow Systems and Technology Group

Software Variant Management Through Process Modeling And Enactment Integration

von Tobias Pfaller, BSc

Software Anbieter sehen sich mit hohen Anpassungsanforderungen ihrer Kundinnen und Kunden konfrontiert. Die viel genutzte Praxis, Software zu klonen und zu modifizieren, führt aufgrund einer steigenden Anzahl an Varianten langristig zu enormen Wartungsschwierigkeiten. Diese Masterarbeit mit dem Titel "Software Variant Management Through Process Modeling And Enactment Integration" beschäftigt sich mit möglichen Architekturveränderungen monolithischer Software, um prozessbasierte modellgesteuerte Konfiguration zu ermöglichen. Dazu wurde ein Framework *Process-Based Mediation Configuration* (PBMC) entwickelt, das zu folgender Vorgehensweise anleitet: (1) die Identifizierung von variablen Komponenten, (2) die Extrahierung der variablen Komponenten, (3) die Erweiterung mit Prozess-Technologie und (4) die Nutzung eines neuen Patterns *Configurable Process Start*, um die für unterschiedliche Kundinnen und Kunden passenden Prozessvarianten bei der Instanziierung zu wählen.

Zur Evaluierung von PBMC wurde (1) ein Beispielprozess in einem realem Szenario (in Kooperation mit einer Partnerfirma) umgesetzt, (2) eine unabhängige Evaluation mit existierenden Ansätzen durchgeführt, und (3) eine prototypische Implementierung entwickelt.

Acknowledgements

I want to thank everyone who has supported me during the long, hard, and strenuous times (more than one and a half years is quite a long time) working on this thesis. Whether support directly related to the thesis, or any other support in this challenging time, I am enormously grateful. Special thanks go to my girlfriend, who relieved me of many tasks in order to get me working on this thesis as well as motivated me again and again. Thanks also go to J. Mangler, who always took the time to help me with problems and gave me tips bringing me on the right track (must have been so exhausting that you fled to Munich). Finally, I want to thank my fellow students Denise, Nikola and Kevin, looking back on six great years together at the University of Vienna.

Contents

List of Figures

To my parents and Lisa and me.

Chapter 1

Introduction

1.1 Motivation

Economy is in a challenging time due to ever-changing demands and globalization. Companies are confronted with great requirements for flexibility and must constantly adapt their processes dynamically to the economic situation. Instead of being central and static, business processes nowadays are affected by a dynamically changing environment including [5]

- regulatory adaptations,
- market evolution,
- changes in customer behaviour,
- process improvement,
- policy shifts and more.

In order to face these challenges, companies must dissolve their static structures and processes and replace them with dynamic ones. Processes are the core of every business and companies are constantly executing processes in every product and service they deliver. For this reason, processes must have top priority and cannot be modified in order to e.g. fit into a particular software or information system. Therefore, the interaction between business processes and information systems became an important research area [6] since the 1990s. Since then, the focus has been on process orientation [7] [8] [9]. The effects of process orientation can already be clearly seen in the market as Kohlbacher [7] outlines companies that have introduced a process-oriented structure as well as the resulting positive effects. Among them are well-known companies like Bosch [10], Volvo car [11], Crédit Suisse [12] and more.

These changes naturally also lead to a change in existing information technology systems. Gong and Janssen [13] define the installed base of existing systems in companies as a cause for flexibility limitations, since they do not use open standards, making the implementation of new systems very difficult. In addition, existing legacy software is mostly not process-oriented which significantly limits the dynamic adaptation of processes. Therefore, functional-oriented information systems are replaced by process-oriented information systems, realizing a clear separation of business process logic and application functions [14].

The change from application orientation to process orientation naturally also has an impact on software vendors. Since each customer is unique and software needs to fit into individual processes, a standardized offering is often not sufficient. Therefore, customization and configuration capability is a key success factor of software vendors [15]. Since customization represents the

altering of source code, it results in different variants of a software that must also be maintained. Many companies perform customizing by cloning and modifying existing software, because the entrance barrier is very low and any freedom and independence for adjustments is given [16]. However, in the long run, especially with an increasing number of created variants, cloning results in maintenance difficulties [16]. Configuration in contrast, contains different variants of a software and allows to switch between them based on configuration or scripting mechanisms.

While customizing will always result in software variants that need to be individually maintained, configuration has the potential to provide low-maintenance software adaptation, depending on the quality of the implemented configuration mechanisms. Therefore, configuration became an important research area for academia and industry [17].

This thesis deals with an example of a software vendor who, so far, customized software by cloning and modifying and is now facing serious maintenance problems due to a large number of software variants to be maintained. The objective of this scenario is to extend the existing software with process technology in a way that variants can be implemented by model-driven configuration, without editing source code. Therefore, a process-oriented mindset should already be applied in software development, in order to allow users to enact their individual process variants. For this purpose, a Business Process Management System (BPMS) can be used. A BPMS allows the modeling and enactment of business processes, as well as their analysis, monitoring and more (depending on the BPMS used). Dumas et al. [1] identified four broad categories of advantages making the integration of a BPMS attractive for companies:

- workload reduction
- flexible system integration
- execution transparency
- rule enforcement

If a BPMS could be used for supporting a model-driven configuration of software variants, these advantages would affect both, the software vendor and the user. While the software vendor could benefit from only having to maintain one software (workload reduction) and could flexibly integrate additional systems, the user could benefit from the accessible data stream generated during execution. This increased transparency will allow him to implement rule-based compliance checking [18], and to implement process changes on-the-fly.

Therefore, the objective of this thesis is to develop an approach in order to transform / extend the architecture of a software existing in multiple variants in a way, so that these variants can be easily implemented through model-driven configuration.

1.2 Research Questions

The motivation stated in section 1.1 leads to following three research questions addressed within this thesis:

1. Highly customized software requires either that (1) different versions of the software are maintained, or that (2) extensive configuration / scripting mechanisms are included in the software. Which possibilities exist to transform / extend existing legacy software to solve the maintenance problems when dealing with software variants?
2. Existing concepts regarding model-driven configuration, assume a clear separation between process / business logic, functionalities, and user interaction. Which additional considerations and modeling elements have to be taken into account, when transforming / extending existing legacy software, in order to define a simple / yet comprehensive interface between the legacy software and model-based extensions?

3. What are relevant metrics and characteristics, to determine the complexity and feasibility of legacy software transformations / extensions, in comparison to existing approaches?

1.3 Contributions

This thesis describes a framework named *Process-Based Mediation Configuration* (PBMC), which helps in transforming / extending monolithic software, existing in multiple variants, in order to obtain a single service-oriented software that can be adapted to individual user requirements through process-based model-driven configuration. For this purpose, PBMC consists of four stages. The first three stages describe the identification and extraction of variability as well as an augmentation with process technology. The fourth stage describes the management of resulting process variants and therefore introduces a new pattern *Configurable Process Start*, providing a new perspective on process variant management.

In order to evaluate and demonstrate the developments, (1) the application of PBMC is demonstrated in a real-world scenario, (2) a detailed comparison to existing model-driven configuration concepts is performed, and (3) a prototypical implementation demonstrating the presented concepts (especially the architecture of a software after applying PBMC and the pattern Configurable Process Start) is developed.

1.4 Methodology

In order to develop the solutions to the problems stated in this thesis, a science-based research methodology called design science is used. Design science [19] consists of the following six stages:

1. Identify the problem.
2. Define the objectives of a solution to the problems.
3. Design the solution.
4. Demonstrate the solution (through implementation).
5. Evaluate the solution (against the objectives).
6. Communicate the solution.

The main research - in how to best handle software variants through the use of process technology - has been tackled by modularizing a monolithic software and augmenting it with process technology. As a result, two stakeholders are involved:

- *Software Vendor:* develops and maintains the software and delivers it to the users.
- *Software User:* uses the software.

Hence, both stakeholders are affected by the solutions presented in this thesis. Therefore there is a special focus in highlighting the results of both perspectives throughout this thesis.

Additionally, since various (conceptual) methods for handling process variants already exist, there is a special research focus throughout this thesis to point out a different perspective on process variants using the approach developed in this thesis in comparison to the existing approaches.

1.5 Structure of the Thesis

So far, the motivation (section 1.1) of this thesis has been stated and the resulting research questions (section 1.2) have been presented. This first chapter is then continued with section 1.3 "Contributions", where a sneak preview of the newly developed concepts and the resulting added value for current science is presented. Moreover, the methodology applied in order to generate the research results has been justified (section 1.4). The chapter is closed with section 1.6 "Definition of terms" in order to get an understanding of how terms used in this thesis can be interpreted and therefore to avoid misunderstandings due to terminology.

In the next chapter (chapter 2 "Scenario"), the practical problem that should be solved in the course of this thesis is illustrated. For this purpose, the current architecture of a monolithic software is analysed and the resulting problems are shown. In addition, requirements for the solution are presented.

In chapter 3 "Related Work", the way to possible solutions should be introduced on the basis of existing, current research literature. In addition, other similar works and concepts used for a comparison in the later course of the work will be briefly explained.

Chapter 4 "Solution Design" contains the solution approach solving the problems presented in the scenario (chapter 2). This includes the development of a framework "Process-Based Mediation Configuration", which consists of the transformation of a monolithic into a service-oriented software and its augmentation with process technology. In addition, a pattern "Configurable Process Start" is developed, which provides a new perspective on process variant management. The pattern's effectiveness is demonstrated by explaining its application in the course of the practical scenario.

Since the developed solution approach foresees the use of a process engine, which will also be implemented in the prototypical implementation of this thesis, chapter 5 "Industry Case Study" presents a real world case study that illustrates the selection process of a suitable process engine. For this purpose, five process engines were evaluated on the basis of 23 main criteria, and the process of selecting an appropriate engine is illustrated.

In chapter 6 "Prototypical Implementation", a sample implementation is presented, which is intended to reflect the practical software architecture arising from the solution approach. For this purpose, the exact architecture of the implementation is explained and process flows and results are described.

Chapter 7 "Discussion" summarizes and discusses all developments presented in this thesis. This includes the Framework *Process-Based Mediation Configuration* and its pattern *Configurable Process Start*, as well as the comparison to existing approaches. Thus, the research questions stated in section 1.2 will be answered and resulting contributions will be defined.

The thesis is concluded with chapter 8 "Conclusio and Future Work", where the most important aspects are briefly covered and a prospect for future work is given.

1.6 Definitions of Terms

1.6.1 Process Engine, BPMS and WfMS

Dumas et al. [1] defines a Business Process Management System (BPMS) as "a system that supports the design, analysis, execution and monitoring of business processes on the basis of explicit process models". Based on this definition, a BPMS accompanies a process-oriented organization through the BPM lifecycle (see section 3.1.1.1). Thus, the BPMS is the central point of a process-oriented organization, supporting the process participants from process design to execution and its monitoring.

In order to support the BPM of users through all phases, the BPMS naturally requires different components. Dumas et al. [1] divides the architecture of a BPMS into different components:

- Execution engine
- Process modeling tool
- Process model repository
- Administration and monitoring tools
- Worklist handler
- Execution logs

Apart from the term BPMS, Workflow Management Systems (WfMS) are also often mentioned. The difference between these two systems is, that the focus of a WfMS is only on process modeling and its execution, while a BPMS also supports analysis, monitoring and other features [1].

In the course of this thesis, the term **process engine** is used a lot. It is meant to be a generic term representing all kind of systems (BPMS and WfMS).

1.6.2 Process, Business Process and Workflow

Weske [20] defines *"A business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal."*

However, Dumas et al. [1] define a business process as *"a collection of inter-related events, activities, and decision points that involve a number of actors and objects which collectively lead to an outcome that is of value to at least one customer"*.

The term "Workflow" is often used as a synonym for "Business Process". Nevertheless, according to the Workflow Management Coalition [21] a workflow is "the automation of a business process ..." and therefore represents a more technical perspective.

However, in this thesis the terms "process" and "business process" are used as general terms referring to the execution of several activities in order to achieve a certain outcome.

1.6.3 Process Family and Process Variant

The terms process family and process variant occur many times during this thesis. **Process family** is intended to represent the generic term for a process that can exist in several variants. A process family does not represent an executable process, but only the border for underlying variants. The term **process variant** then represents a variant of a process family.

For example, if a process *Print Report* exists in three different variants (due to different parameters that should be on it), the process family is the generic term "Print Report", and a process variant is one of the three available variants.

Chapter 2

Scenario

2.1 Company Partner

The scenario of this thesis is a practical use case of a company partner (further called "CP"). CP is an Austrian automation company and its offer covers many different hard- and software products such as

- switchgear cabinets,
- interface software,
- warehouse management systems, or
- measurement engineering systems.

The customers of CP are companies operating in a wide range of industries. These range includes industries from the agribusiness to machine and plant manufacture, the chemical industry, water supply installations and many others.

2.2 Software X

One of the offered products is a software called Software X (further called "SX") developed by CP, which represents the main focus of this thesis. The goal of SX is to offer operators of a plant an optimal interface to control the plant and related systems. In order to fulfill this purpose, SX consists of several composite modules like:

- *Programmable logic controller:* The programmable control unit and interface representing the core of a plant.
- *Database:* Storage of current and historical information about the plant and products.
- *Order management:* Input of production orders and preparation for production.
- *Maintenance module:* Overview of necessary maintenance.
- *Inventory control:* Overview of the current inventory and batch history.

2.2.1 Architecture of SX

The architecture described in this section corresponds to a generic architecture that is used to illustrate the problem. Components that are not necessary in order to demonstrate the problem are omitted. Thus, this architecture represents a generic example and does not correspond 100 percent to the true architecture of SX.

Figure 2.1 shows that generic architecture. SX is a monolithic system running on the costumers's own IT infrastructure. The central part of SX is its application logic, which has integrated interfaces to many different functions and services (which are either running inside of SX, or running outside of SX).

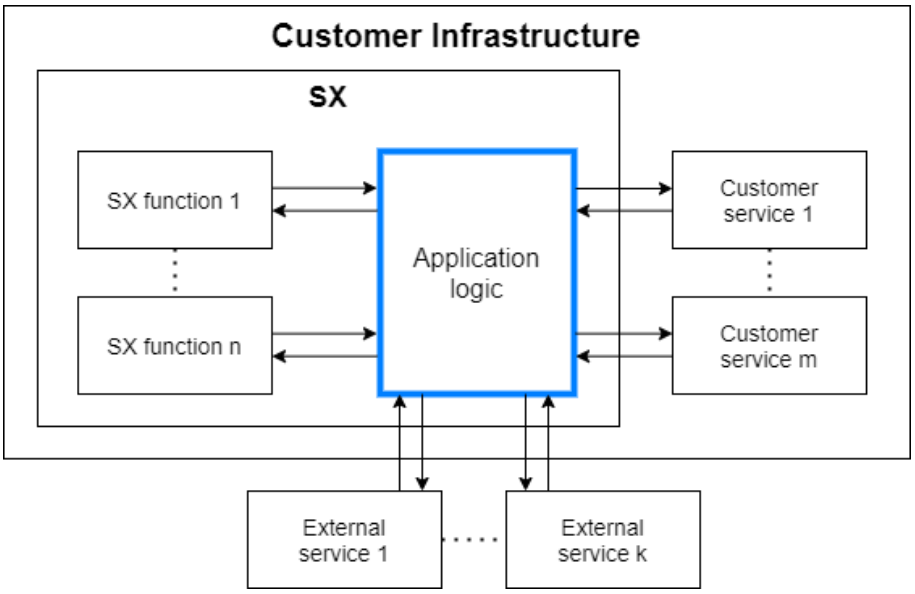


FIGURE 2.1: Initial architecture of SX

However, there are three components within SX's architecture, which represent each an important issue in this scenario and therefore are explained in more detail: services, application logic and procedures.

2.2.1.1 Services

The services shown in figure 2.1 can support any functionality (for example: database calls, web services, plant processes, user interfaces, etc.) and could also run on different (remote) infrastructures. Services communicate via the application logic, therefore, they need to offer an interface to it. Services could also contain other interfaces, but these are not relevant in the course of this problem.

Note: Since SX represents a monolithic software, services running inside of SX are called functions.

2.2.1.2 Application Logic

The application logic is more or less a service itself. However, it is the central unit and contains SX's logic for interaction and information sequences. In addition, it is a mediator between all services and functions. Any sequential call of several services or functions (e.g. in executing procedures) is controlled via the application logic and data is exchanged via it.

However, procedures and communication behaviours defined in the application logic are monolithic and hard-wired into the source code of SX. It is therefore delivered to customers as a part of SX and runs within SX on the customer's infrastructure.

2.2.1.3 Procedures

Procedures contain a control flow for the execution of one or more services (e.g. starting the plant, cleaning the plant, query stock level, etc.).

Therefore, procedures are processes that are defined in SX's application logic. They can be started by an user (e.g. starting a system or querying the stock level) or a machine (e.g. send error messages or success messages to the user interface, etc.).

However, procedures are fixed components in the application logic and are therefore defined in its source code. Changes to procedures are only accompanied by reprogramming the application logic and redelivering the entire SX software.

2.2.1.4 Interaction of Services, Application Logic and Procedures

Figure 2.2 is intended to show how communication between services, the application logic and procedures works. The figure shows a user starting the procedure "start Plant" via the user interface. As can be seen, all participating objects communicate only via the application logic.

After the user has started the procedure, the application logic starts the associated procedure, which determines the further call sequences. In this case, it is a simple service call to the plant, transmitting the information to be started.

After the plant has started, it needs data. To get necessary data, it calls the application logic, which again starts the corresponding procedure. Since in this case data from a customer service is necessary, the application logic makes the call, retrieves the data and forwards it back to the plant.

In this case, two simple procedures were shown. Procedures can of course also be more complicated and require the call of several services. These entire procedures are nevertheless always defined in the application logic.

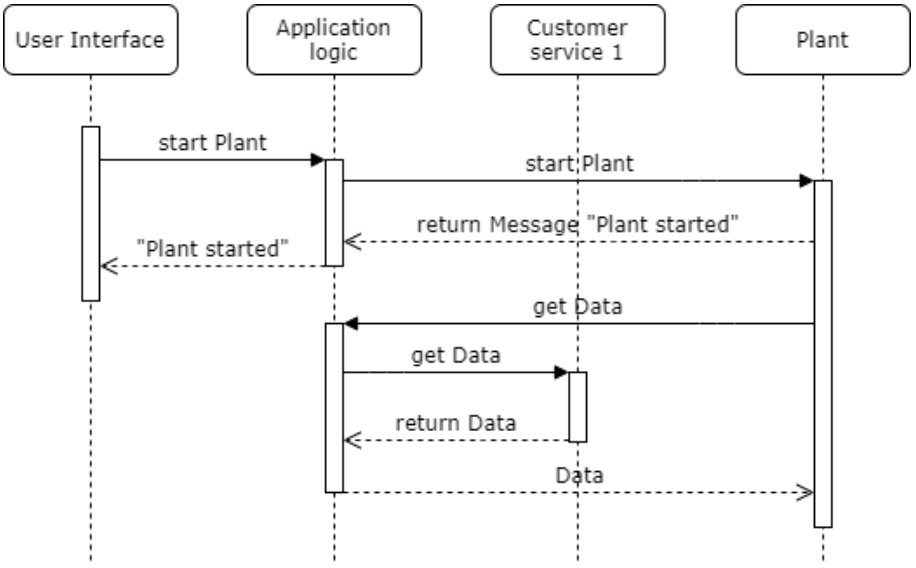


Figure 2.2: Services, application logic and procedures

2.2.2 Variation of the Application Logic

Basically, all customers use the same SX software. However, there are differences in their application logic. Different customers use different procedures depending on their individual needs. So, the application logic is an individual component in SX, that differs for customers.

Figure 2.3 shows a generic example of how the same procedure can differ between different customers. For example, the call sequence differs between Customer A and Customer B, but the same services are used. Customer C, on the other hand, has a totally different procedure, that has no similarity with those of Customer A and B. There could also be a Customer D who has not implemented this procedure at all, because it is not needed. So, there can be serious differences in the executing procedures between customers.

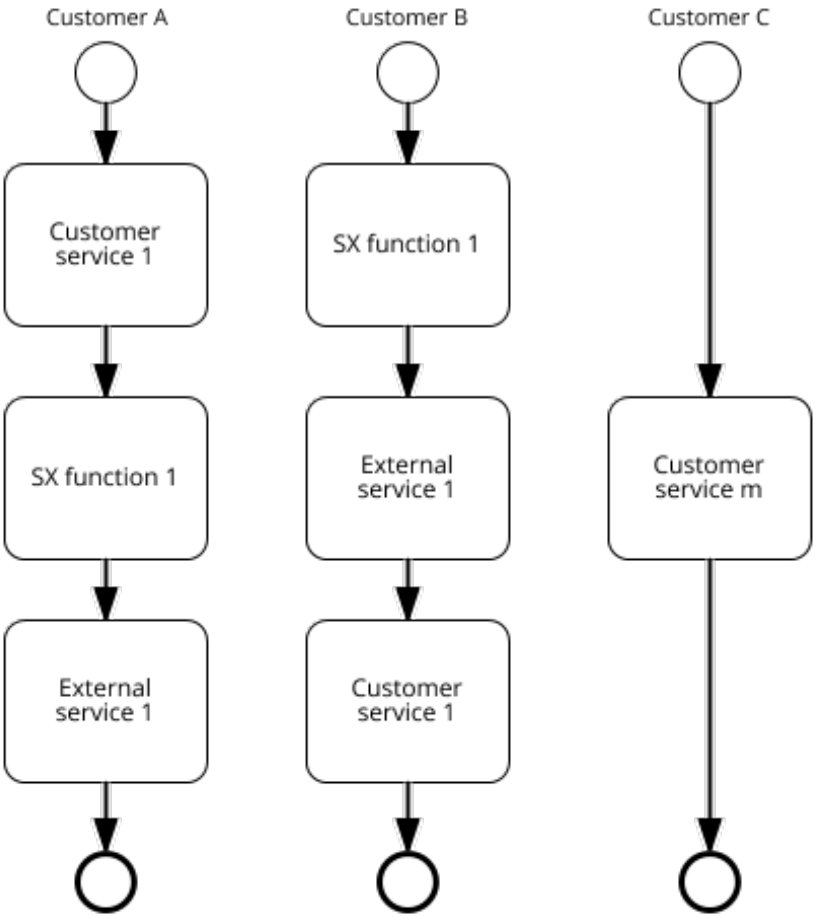


Figure 2.3: Process variants in SX

In addition, these procedures must be regularly changed. So, they are not implemented once and then run for several years, but may need to be adapted to changing circumstances at shorter intervals. Therefore, they must be constantly dynamically adaptable.

2.2.2.1 Practical Illustration

To illustrate this with an practical example, figure 2.4 shows a sample process modeled in BPMN [22] that is executed in SX from customers in the agribusiness industry. The process called

"Approval" represents a decision-making process that should determine if an incoming grain can be forwarded through the gutter into the silo. Therefore, several parameters must be checked. In the case of the sample process: the order status, if the article is blocked, if the response is okay, and the quantity of the order with an user confirmation if it's bigger than 5,000. In other scenarios much more parameters could be checked (e.g. what kind of grain was running through the gutter before, local restrictions, and several more). If the approval is completed successfully the grain will be forwarded, if it is terminated with a termination message the process will be stopped.

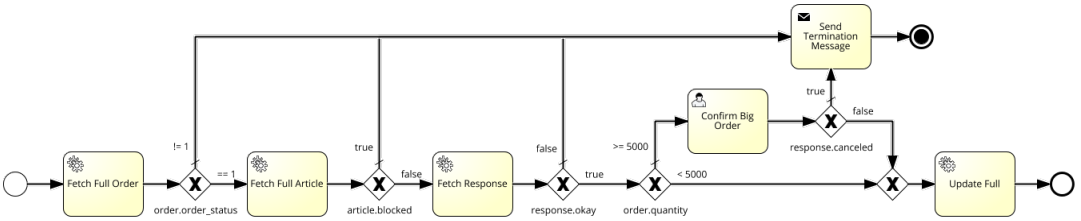


FIGURE 2.4: Sample process "Approval"

Mapping this sample process with the current architecture of section 2.2.1, the following interconnections can be identified:

- The whole process is a procedure, that can be started in SX. Thus, the control flow is defined in the source code of the application logic.
- Each task of the process (e.g. Fetch Full Order, Confirm Big Dialog, Send Termination Message, etc.) represents a service (or a function).

However, depending on the customer requirements, this process may or may not be integrated in the customers SX software. Between customers who have integrated this process, the parameters to be checked may vary. Therefore, this process is defined several times in many different variants (depending upon the customer and its requirements). This scenario applies not only to this sample process, but to several different processes.

2.3 Problem Definition

Finally summarized, there are several factors leading to problems. Among them are

- the monolithic architecture,
- the hard wiring of procedures in the source code of SX's application logic, and
- the variation of the application logic for different customers (due to variation in their procedures).

These problems lead to a high maintenance effort as well as a high susceptibility to errors in various scenarios:

1. *Changing procedures:* Since the individual procedures are wired in the source code of the application logic, which in turn is wired in the overall SX system, (1) the procedure must be redeveloped in the source code of SX and (2) the entire software must be redelivered to the customer.

2. *Delivery of an SX update:* Customers basically use the same SX software, but with different procedures in their application logic. Since these procedures are wired into the overall software, a software update requires a new build for each customer with their specific application logic. Additionally each software build, must be delivered to the associated customer.
3. *Changing services:* Changes to services (internal or external) lead to redevelopment and redelivery (again for each customer) as the interfaces need to be updated in the application.

The requirements of CP to a revising SX, coincide with the solution of the problems just mentioned:

1. Delivery of a uniform SX software to customers (despite variation in the application logic).
2. Changing the application logic (the procedures) without redelivering the entire SX software (configuration instead of customization).

In summary, software variants (affecting only a part of the software, namely the procedures in the application logic) lead, in combination with the current system architecture, to serious maintenance difficulties. Therefore, in this thesis a framework is developed to efficiently enable this software variant management.

Chapter 3

Related Work

3.1 Process Orientation

Globalization and with it the speed of developments, progresses and market changes, are putting companies under increasing pressure. Processes must become increasingly flexible and adapt to current developments and conditions. As a result, they should be constantly adaptable at little costs and with little risk of errors. Therefore, individual processes must be replaced by more current ones while normal operation is maintained and related processes continue running smoothly. These enormous requirements on process flexibility led to a focus on process orientation instead of functional orientation [14].

Process-oriented information systems are characterized by a clear separation between process logic and application functions [14]. Therefore, processes will be modeled, stored and can then be executed by process engines. Depending on the running process, a process engine assigns the individual tasks to the responsible units (human or machine). Once a task has been finished, the process engine moves on and assigns the subsequent tasks. Thus, the total control flow information is kept in a process model, enacted by a process engine. This increases the flexibility of processes enormously, as they can be constantly replaced by new variants without the need to make fundamental changes to existing (IT)-structures. In contrast to process-oriented information systems, the process logic in functionally-oriented information systems is usually hardwired into the source code [23]. Adjustments to processes therefore are extremely error-prone, time-consuming and the processes are rigid and inflexible.

Therefore, there exists a lot of literature, illustrating a wide range of benefits in being process oriented [7] [8] [9] [24] [25]. Additionally, Kohlbacher [7] present effects of process orientation through case studies [10] [11] [12] of companies switching to a process-oriented structure.

3.1.1 Business Process Management (BPM)

The daily work of any company consists of executing processes. Whether it is the delivery of a service or a product, or the manufacturing of products, there are always processes that are executed in the background. The term "Business Process Management" (BPM) comprises methods, techniques and tools supporting the design, enactment, management and analysis of business processes [26]. In contrast, process-oriented organizations think in terms of processes and therefore perform Business Process Management (BPM) [7]. For this purpose, processes are modeled, executed, analyzed and optimized supported by Business Process Management Systems (BPMS). This allows the processes to be flexibly adapted to latest conditions. Additionally, there is full transparency in the execution of the processes. Through monitoring tools included in the BPMS, processes can constantly be monitored and analyzed, problems or opportunities for improvements can be identified and afterwards the processes can be optimized.

3.1.1.1 BPM Lifecycle

BPM is not something that is introduced once and then no longer need to be considered, but a constantly running process. Organizations that perform BPM manage a variety of processes. These processes are constantly monitored, analyzed and improved in order to always work at maximum efficiency. This continuous process improvement was defined by Dumas et al. in the BPM Lifecycle [1] and consists of six phases: process identification, process discovery, process analysis, process redesign, process implementation and process monitoring.

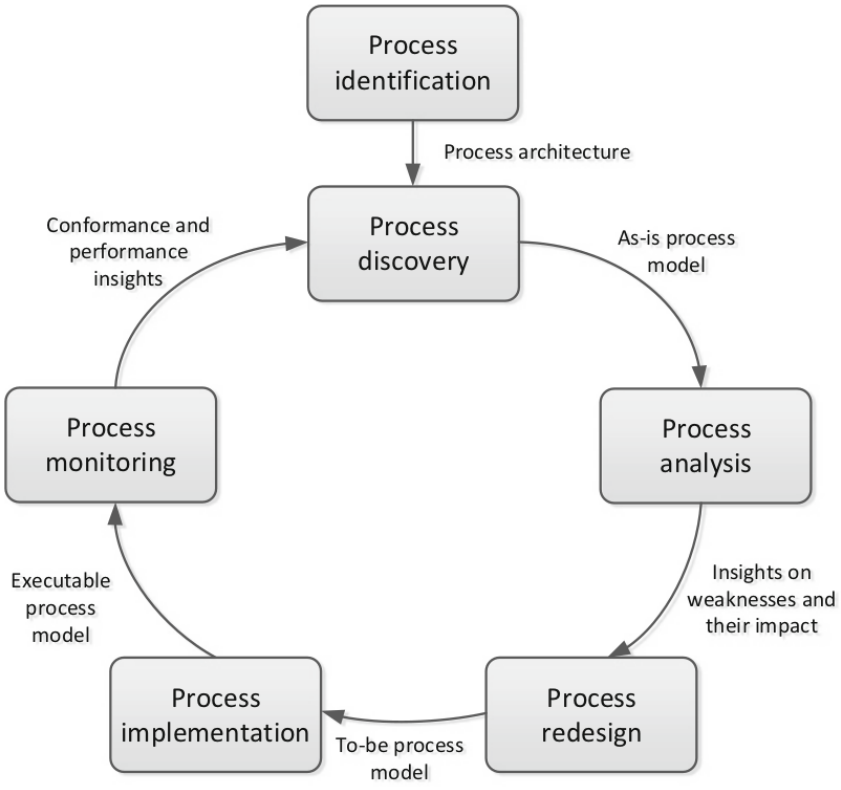


Figure 3.1: BPM lifecycle by Dumas et al. [1]

Process identification. In this phase, the processes of the organization are identified, delimited and inter-related. The result of this phase is the process architecture of the organization.

Process discovery. In this phase, the processes of the process architecture are discovered in detail. For this purpose, the processes are modeled. The result of this phase are the as-is process models. The as-is process models represent the process in its current state. Therefore, this phase is also called as-is process modeling.

Process analysis. In this phase, the as-is process models will be analyzed. Therefore, the processes are observed and documented. It is also important to talk to current employees involved in the process in order to obtain their opinion. Additionally, certain key figures will be used to measure current process performances. Ideally,the result of this phase are insights on weaknesses and their impact in the currently running processes.

Process redesign. In this phase, which can also be called process improvement, the insights on weaknesses and their impact are used to improve the processes. Therefore, the as-is process models are redesigned, in order to eliminate weaknesses and introduce improvements. The result of the process redesign phase are the newly designed process models, so-called to-be process models.

Process implementation. In this phase, the old as-is process models will be replaced by the new to-be process models. Replacing the old processes naturally brings challenges, as both the organizational structure (e.g. employees involved in the process) and the technical structure (information systems involved in the process) must be adapted to the new processes. The result of this phase are the implemented and therefore executable process models.

Process monitoring. In this phase, the newly implemented processes are monitored. After the new processes have been implemented in the organization, they must of course be monitored and data must be collected to define how well the new processes are running. Any problems that may arise should be identified and eliminated here. The result of this phase are conformance and performance insights on the new process models.

Business Process Management Systems (BPMS) are process-oriented information systems that support the execution of BPM. BPMS and their difference to Workflow Management Systems (WfMS) have already been described in chapter 1, section 1.6.1.

3.2 Integration of a BPMS

As stated so far, a solution to the problem of CP is to think in terms of process orientation. The executed processes and their process logic must be brought to the front and separated from the functional application properties. This enables efficient business process management - the processes can be modeled in explicit process models and executed by a process engine. By running through the BPM lifecycle, these can be constantly optimized and improved.

The basic prerequisite for this process-oriented way of thinking and working is the introduction of a process engine. With its support, processes can be modeled and executed (and - in the case of a BPMS - monitored, optimized and more). Conversely, the degree of process orientation has an impact on the positive implementation of a process engine [24].

The introduction of a process engine brings many advantages for company, but also poses challenges. This section will therefore list both, advantages of implementing a BPMS, followed by challenges.

3.2.1 Advantages of Introducing a BPMS

In addition to the general advantages of a process-oriented view, which were stated in section 3.1 "Process Orientation", the introduction of a BPMS leads to further advantages. Dumas et al. [1] identified four broad categories of advantages that make it attractive for organizations to use a BPMS (the source refers only to BPMS, however these advantages also refer to WfMS):

1. *Workload Reduction:* A BPMS distributes the tasks and thus no employee has to worry about the "control flow" of a running process. When a task is completed, it is automatically forwarded to the next position. There is also no need for coordination - the BPMS decides what is done, when, by whom and assigns these tasks directly.
2. *Flexible System Integration:* Process logic is kept separate from functional applications. Since information systems then only contain functional application code and no business process logic, they work independently and are much easier to maintain or even to replace. New information technology systems do not have to be adapted to process logic, because process logic runs independently of any other system in a BPMS.
3. *Execution Transparency:* Since the BPMS manages all processes, the management department and process participants have full transparency over its execution. They get insights, they might not have without a BPMS. In addition, both, running and already completed processes can be analyzed. This is very important in order to identify problems as well as opportunities for improvements.

4. *Rule Enforcement:* By executing processes through a BPMS using explicit process models, people can be sure that processes are executed exactly as they were modeled. Rules that have been modeled in the process are executed with certainty. If the process is coordinated by people instead of a BPMS, there is a certain freedom that could lead to irregularities and to certain rules not being observed. This cannot happen when processes are executed using a BPMS.

3.2.2 Challenges of Introducing a BPMS

In addition to many advantages of introducing a BPMS, there are of course also some challenges to be handled in order to guarantee an efficient implementation. Dumas et al. [1] distinguishes between technical and organizational challenges:

1. *Technical Challenges:* Old, static systems, which partly are still entangled in the IT infrastructure of organizations, were not designed to interact with other systems (especially BPMS) at the time of their creation. It is often a great challenge to integrate the necessary interfaces in order to trigger these systems from a BPMS. The fact, that the developers of such systems are often no longer part of the company and that documentation is rarely available often makes the integration of a BPMS even more difficult.
2. *Organizational Challenges:* The introduction of a BPMS has a massive impact on an organizational level. It can lead to massive changes in processes, which are accompanied by changes in the employee's daily work. Resistance may be encountered and certain people may not be happy with such changes. Therefore, the introduction of a BPMS can be a very challenging task, as interests of all stakeholders must be balanced.

3.3 Process Variant Management in Common Modeling Tools

In process-oriented organizations, every process is stored in an explicit process model which will be executed by a process execution engine. Hence, each process type (e.g. sales process, production process) consists of its own process model. However, a process can take on different variants (e.g. production of different car models). Consider an internationally operating online store as an example. The purchase completion process will basically be the same in all countries over the world. However, different laws in different countries may require different processes. This problem results in the process differing slightly from country to country, although the basic process is the same. Visualizing different process variants, like these, clearly and consistently, while remaining flexible, is a challenging task. Hallerbach et al. [27] identified two approaches of modeling process variants in common modeling tools, frequently applied in practice: the *single-model* and the *multi-model* approach.

Single-Model Approach: Using the single-model approach, multiple variants are captured in a single model using conditional branchings (XOR/OR splits). Depending on the number of different variants, the model becomes very huge and confusing because of the high number of XOR/OR splits. Additionally, it will be difficult to distinguish between the different variants and identify variant-specific paths. If a variant needs to be changed, this leads to enormous maintenance costs and a high error rate. Moreover, Mendling et al. [28] define to use as few elements as possible in a model, to decompose models with more than 50 elements, to minimize routing paths and to avoid OR routings.

Multi-Model Approach: Using the multi-model approach, each variant is captured in its own process model. In contrast to the single-model approach, variant-specific changes can be easily implemented here, since the variants are separated into different models. However, applying this approach leads to multiple (more or less) redundant models, which may only differ slightly from

each other. If the basic process is changed, each variant must be adapted individually, which, as with the single-model approach, leads to enormous maintenance costs and a high error rate.

3.4 Existing Approaches of Process Model Configuration

In the course of this related work, three existing approaches (PROVOP [29], C-EPC [2] and REMUS [3]), allowing process configuration will be explained in detail, as these approaches will also be used for an evaluation later in the thesis.

3.4.1 Process Variants by Options (PROVOP)

The basic idea behind PROVOP [29] [30] [31] is to represent all process variants of a basic process in a single process model. Three basic concepts for representing process variants with PROVOP are briefly explained below (information taken from Hallerbach et al. [29]):

1. *Basic Process.* Based on this process, different variants will be derived. The basic process itself can be a variant defined for a specific use case, but can also be defined without specific use case.
2. *Change Operations.* Change operations represent differences between the basic process and a process variant. The PROVOP approach allows the change operations INSERT, DELETE, MOVE and MODIFY. Each of these operations has a related symbol and needs a set of input parameters to guarantee a correct execution.
3. *Options.* To define complex variants which need multiple adjustments from the basic process, change operations can be grouped to an option. Therefore an option consists of a set of change operations.

3.4.2 C-EPC

In order to be able to adapt/configure process models to different needs, Van der Aalst et al. [2] identified nine workflow patterns that allow all possible configurations of a process model and derived a new process variant modeling language "Configurable-EPC" (C-EPC) based on Event-Driven Process Chains (EPC). Therefore, based on these nine configuration patterns, the modeling language EPC was extended. This allows a process to be adapted to individual needs by built-in configuration before it is implemented in an organization. The identified configuration patterns are briefly explained below (information taken from Van der Aalst et al. [2]):

Optionality. Tasks within a process model can be enabled and disabled at configuration time, or this decision can be postponed to runtime.

Parallel Split. The parallel split pattern represents an AND. With the configurable AND, the number of paths executed in parallel can be reduced. It is important to note that a configurable AND must remain an AND even after configuration (at least 2 paths executed in parallel). The only decision the modeler has, is to reduce the number of executed paths (e.g. from five to three).

Synchronization. The synchronization pattern merges all paths of a parallel split, therefore it represents the end of an AND.

Exclusive Choice. The exclusive choice pattern represents an XOR. Using the configurable XOR pattern, either one of the paths can be already selected at configuration time, or the decision between desired paths can be postponed to runtime.

Simple Merge. The simple merge pattern merges an exclusive choice and therefore represents the end of an XOR.

Multi Choice. The multi choice pattern represents an OR. One or more paths can be selected using an OR pattern. Therefore, at configuration time, this pattern has the potential to get either an OR, an AND, or an XOR.

Synchronising Merge. The synchronising merge pattern merges a multi choice and therefore represents the end of an XOR.

Interleaved Parallel Routing. The interleaved parallel routing pattern allows tasks to be moved sequentially. Within the configurable interleaved parallel routing pattern, tasks are not arranged sequentially, the sequence of tasks is either arranged at configuration time or is postponed to runtime.

Sequence Inter-relationships. The sequence inter-relationships pattern represents the dependency of tasks in different processes. It allows to set tasks of a related process on, off or optional (pattern "Optionality"), depending on tasks and results of the actual process.

Figure 3.2 gives an overview of the configuration patterns "Parallel Split", "Exclusive Choice" and "Multi Choice" with an example each, demonstrating how they are modeled in a configurable model and models that can result after configuration (build) time.

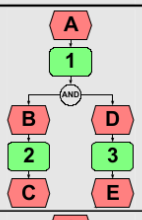
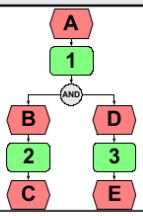
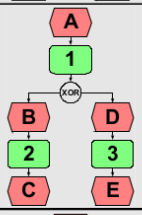
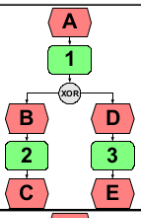
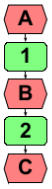
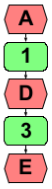
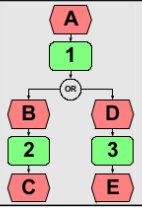
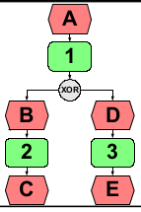
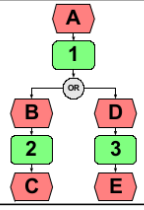
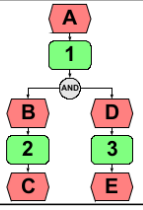
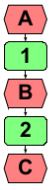
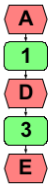
Configuration Pattern	Depiction of Configuration Pattern	Build Time Configuration Possibilities				
		XOR	OR	AND	Sequence	
					Sequence 1	*Sequence 2*
Parallel Split		*n.a.*	*n.a.*		*n.a.*	*n.a.*
Exclusive Choice			*n.a.*	*n.a.*		
Multi Choice						

FIGURE 3.2: Configuration pattern by Van der Aalst et al. [2]

Configurable Event-Driven Process Chains (C-EPCs) [2] are an extension of EPCs by configuration nodes and configuration attributes. The extension by such nodes and attributes allows to derive an individual EPC process model by configuration decisions from a C-EPC model.

3.4.3 REMUS

Vigne [3] created a REstful Marketplace for Unified Services (REMUS), allowing, to integrate different services in an actively running process. For this purpose, the BPMN was extended with a resucue-ring (and more concepts, but they are not important in the course of this thesis), representing injection calls, meaning that the control flow will be extended with any subservice (some different subservices may be available).

At first glance, this approach may not look like it could be used in order to create process variants. However, if a basic process is defined and can be manipulated by extending it with different services, it's possible to handle process variants by means of these varying services. Figure 3.3 represents a process flow using REMUS. The initial process consists of three tasks: *Perform 'Search & Book'*, *Manipulate from 'Search & Book'* and *Remove Objects of 'Search & Book'*. While the second and third tasks are normal tasks, the first one has an rescue-ring and therefore represents an injection call. Any subservice can be called and as can be seen, subservices can again contain injection calls (e.g. "Call Find" and "Perform Search").

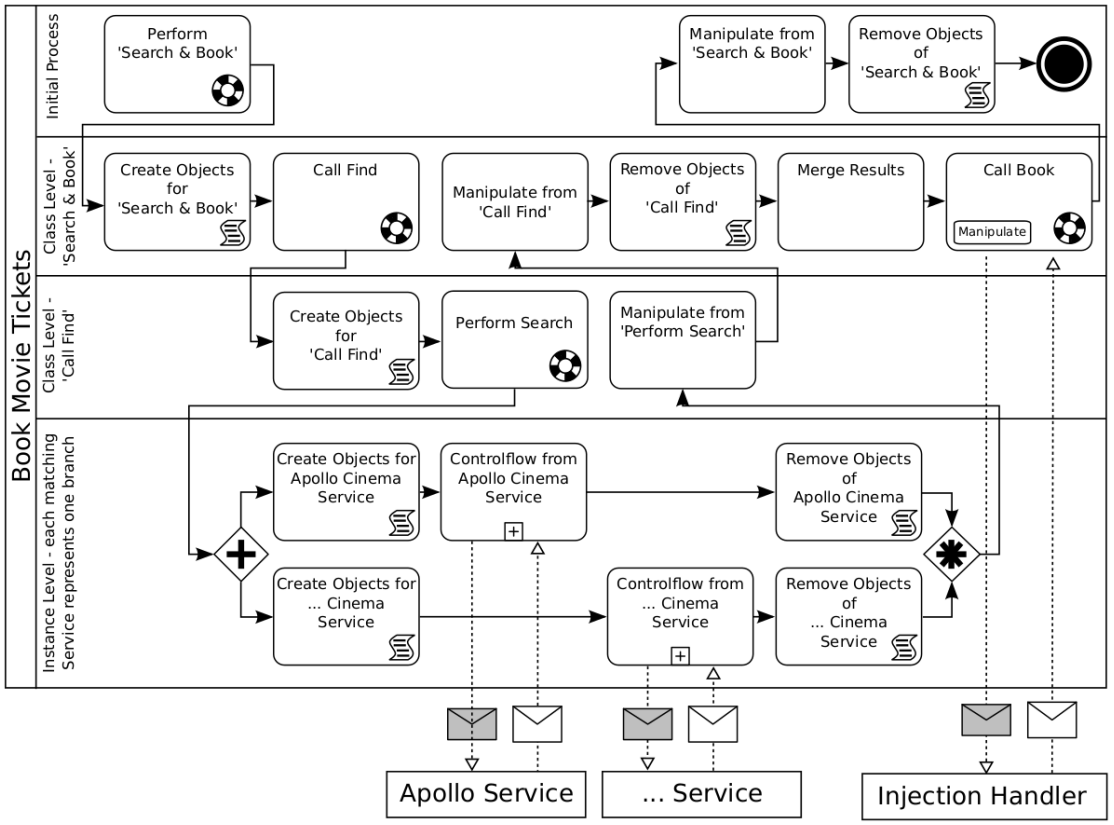


FIGURE 3.3: REMUS by Vigne [3]

In the end, the conceptual idea behind REMUS is not to handle process variants, but the approach can be used to do so. Based on a minimal initial process, there is the possibility to create desired process variants by extending the initial process with a variety of subservices through injection calls.

Chapter 4

Solution Design

The scenario introduced in chapter 2 requires a solution approach allowing a flexible variation of SX's application logic for different users. Customers should be able to use their different procedures in SX without CP having to slightly change the overall software construct for each of them. For this purpose, a framework "Process-Based Mediation Configuration" is presented in this chapter, whose application leads to an extraction of process logic from a monolithic software and its extension by process technology in order to enable a dynamic management of process variants. Additionally, the framework introduces a new pattern "Configurable Process Start", which is intended to give a new perspective on process variant management.

Furthermore, the framework and its pattern "Configurable Process Start" is compared to existing approaches of process variant management in order to determine similarities and differences as well as advantages and disadvantages. The objective of the developed solution approach is not to limit itself to the use case of CP, but to represent a framework that can be used universally. Thus, the framework should also be applicable to other scenarios, as carried out in an implementation demonstration in chapter 6.

4.1 Process-Based Mediation Configuration

Process-Based Mediation Configuration (PBMC) forms a guideline for architecturally modifying a standard monolithic software and augmenting it with process technology in order to allow process variations for different users. Thus, as a result after application, a uniform standard software can be used and managed for all users, and nevertheless, the software can still be variably adapted to their individual processes. Costly customizing for each individual user and the associated maintenance difficulties are eliminated and replaced by flexible process modeling and enactment in a process engine.

To this end, PBMC (figure 4.1) consists of four stages:

1. *Variation Identification.* The variable component inside of the monolithic software is identified and delimited.
2. *Variation Extraction.* The variable component is extracted from the monolithic software. Solely an instantiation instance (called "worker") remains in the software, containing an interface to the process engine for instantiating and starting processes. Internal interfaces to and from the variable component are replaced by external interfaces.
3. *Augmenting with Process Technology.* The software is extended by an externally acting process engine. The processes defined in the source code of the variable component are lifted to the modeling layer and therefore replaced by explicit, executable process models.

The multitude of resulting process models are structured in individual or grouped process repositories.

4. *Process Variant Management.* In order to manage the process variants emerging in the previous stage, a new pattern "Configurable Process Start" is introduced, representing a new view on process variant management. Nevertheless, existing process variant management approaches can be used instead, or in addition, in this stage.

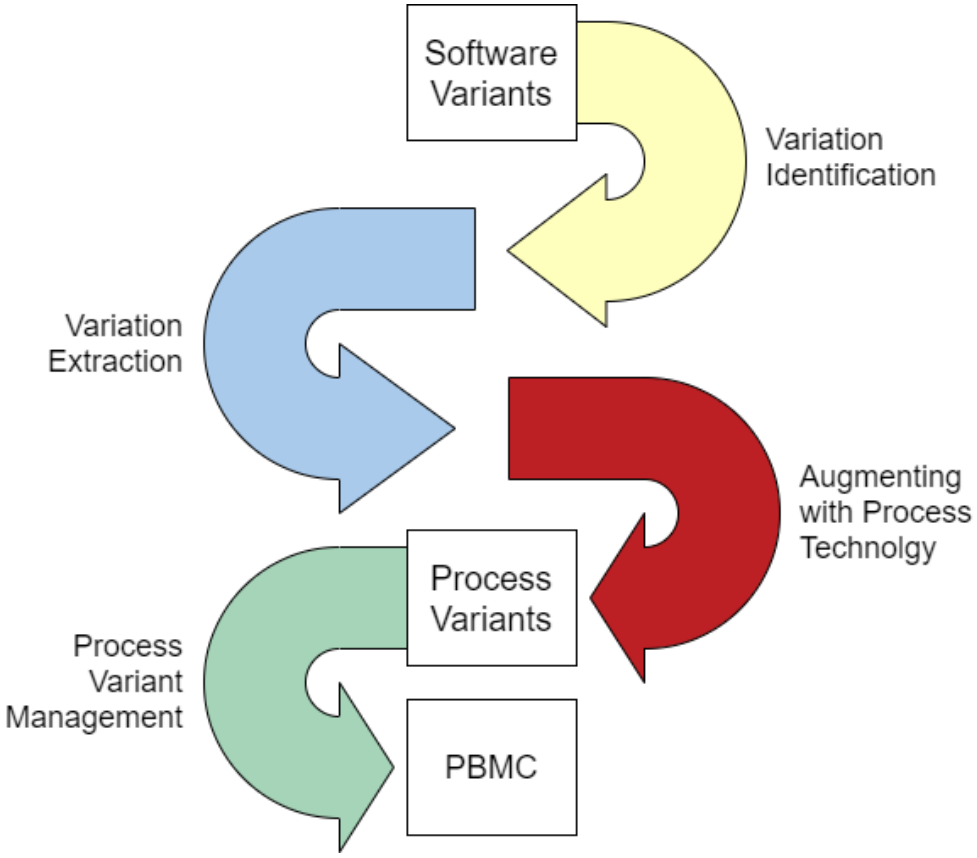


FIGURE 4.1: Framework "Process-Based Mediation Configuration"

The initial situation of the framework is a software that exists in different variants in order to allow different users to execute their individual process variants. Therefore, the first three stages take place entirely in parallel for all users of the software. The last stage "Process Variant Management" allows a simplified and clear management for the variance and redundancy created by the parallel creation of process models.

PBMC will be applied to the scenario of CP and its four stages will be run through. After the initial architecture was presented in chapter 2, figure 4.2 represents the final architecture of CP's software SX after the application of PBMC. The process engine and its related process models represent a mediator mediating between the worker and all services to be called. All sequences and communication is handled by the process engine and its process models.

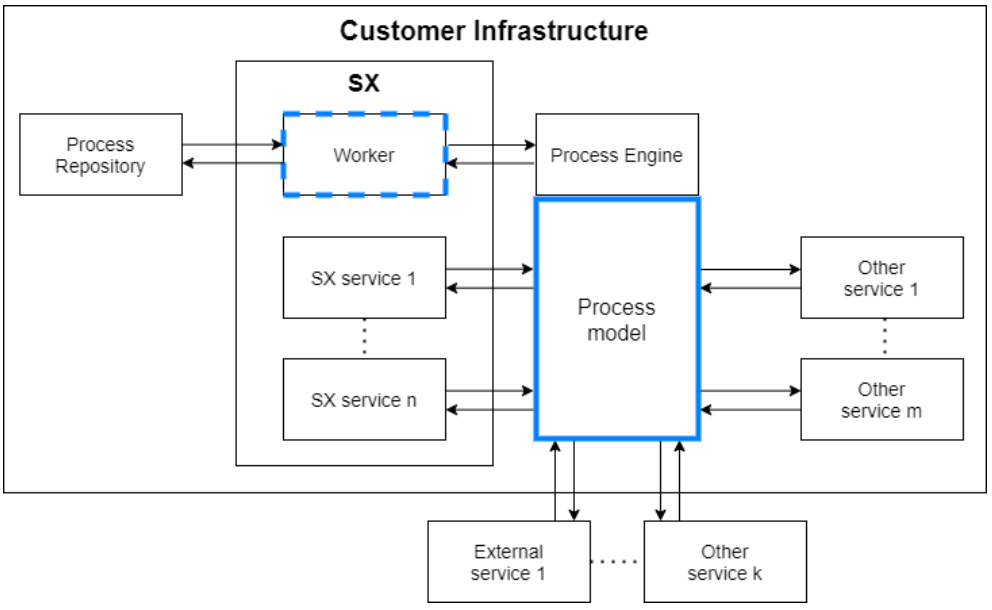


FIGURE 4.2: Architecture of SX after the application of PBMC

The term "Mediating" as a part of the name in PBMC refers to the mediating function of the process engine in the final architecture. As can be seen in figure 4.2, services do not communicate directly with each other, but only via process models. Therefore, the process engine can be compared to the software design pattern "Mediator". According to Gamma et al. [32] the pattern "Mediator" is defined as *"Define an object that encapsulates how a set of objects interact"*. In our case, this definition clearly applies to the process engine, since interaction is only possible via its processes and the forms and sequences of communication are defined there. Figure 4.3 demostrates the equality between the process model (as part of the process engine - see figure 4.2) and the pattern "Mediator".

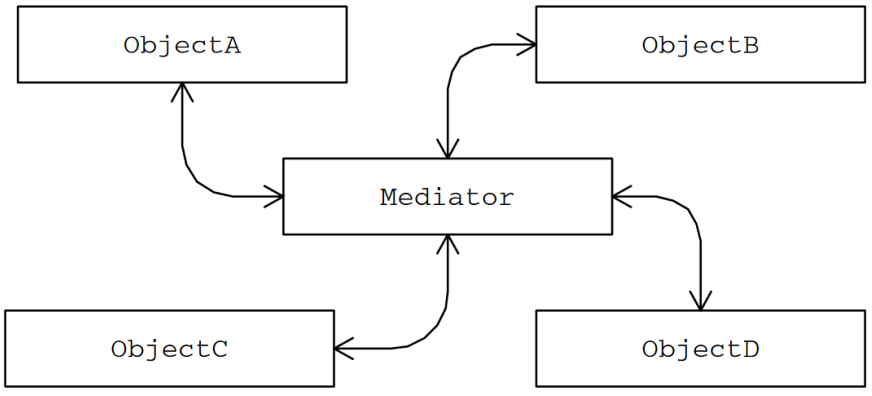


FIGURE 4.3: Mediator demonstration [4]

Before explicitly discussing the application of PBMC in CP's scenario, the following subsections provide a more detailed conceptual explanation of PBMC's four stages.

4.1.1 Variation Identification

The identification of the variable component and its delimitation from other components in the initial software is conducted in this stage. The initial software is characterized by a monolithic

architecture in which one (or more) component(s) contain variability for different users. The variable component must therefore be identified and clearly delimited from static components. Thus, this first stage represents a kind of definition of the initial problem.

4.1.2 Variation Extraction

4.1.2.1 Extraction of the Variable Component

The variable component is extracted from the monolithic software. Solely a minimal part (worker) of the extracted component remains in the system, responsible for instantiating and starting processes. In order to keep no variability in the system, the remaining worker is identical for all users. Thus, all user individuality is outside of the software and the remaining part is the same for all users.

4.1.2.2 Interface Adaptation

Since the variable component has been removed from the system, the interfaces must also be adjusted. All interfaces called by the variable component must now have a call option from outside in order to be called by the process engine, which acts outside of the software. That means, that functions inside of the monolithic software will be transformed to services, callable from outside. Functions used to start procedures, will be redirected to the worker. Using this worker, the process engine is called and desired procedures are instantiated and started in form of executable process models.

After completing this stage of PBMC for all users, all variability is banished from the software (since it is now outside of the system) and thus only a single, uniform variant of the software exists for all users.

4.1.3 Augmenting with Process Technology

4.1.3.1 Implementation Process Engine

The software is extended by an external process engine in this stage. Therefore, as a first step, an appropriate process engine, that meets the requirements of the organization, is selected and implemented. The location of the process engine implementation can either be on the same infrastructure where the actual software is running, or on an external infrastructure that is accessed via cloud. The worker, remaining in the system, must have an interface to the process engine in order to instantiate and start processes.

4.1.3.2 Code to Processes

As a next step, the source code of the extracted variable component will be restructured and therefore modeled in explicit process models executable by the process engine. Hence, the source code is lifted to the modeling layer.

4.1.3.3 Structure Process Models in Repositories

After the transformation of the source code into process models, each user has a multitude of process models that differ more, less or not at all from those of other users. Therefore, there are two possible scenarios of how these processes are structured:

(1) *Individual repository for each user.* Each user has his own process repository, independent of other users, which is likely also managed on his own infrastructure, resulting in:

- *Secret processes.* Processes remain secret, since every user has an own process repository where no one else has access.
- *Easy maintenance of "single" processes.* The maintenance of a single process is simple, since no other user is affected.

An individual repository for each user is a reasonable solution for users acting absolutely independent to other users.

(2) *Grouping the processes of multiple users in one repository.* There is a grouped repository for several, or even all, users. It is located and managed in a central place and is either directly called for the instantiation of processes, or is always delivered redundantly to all users in order to run on their own infrastructure. Grouped repositories result in:

- *Re-usability and therefore simpler maintenance of redundant processes.* Redundancy could be avoided if users run the same process. Fundamental process changes affecting processes executed by many users only need to be adjusted in a single process model, rather than individually for each user.
- *Process models of other users can be called.* Users have the possibility to execute process variants of other users at any time (e.g. for testing purposes).

A grouped repository is a reasonable solution for interdependent users (e.g. same plants at different locations of the same company). Since many similar, or even identical processes of different users are handled in one repository, this solution requires process variant management in order to handle these processes.

There can also be a combination of the two scenarios (partly grouped, partly individual repositories).

4.1.4 Process Variant Management

This last stage of PBMC is a very general and optional part, which can be implemented very differently, or can even be omitted.

Whether this stage is applied or not depends on the structure of the process repositories, designed in stage 3 "Augmenting with Process Technology" subsection 3 "Structure Process Models in Repositories". If separate process repositories for each user are utilized, the framework is finished since each user handles its processes in its own repository. So, there is no need to manage process variants of many users in a single repository. However, if a grouped repository for multiple users was created, process variant management is necessary and appropriate approaches will be applied.

Since in grouped repositories different users apply the processes of the same process family in different forms, different variants of processes are stored in the repository. In order to avoid redundancy, reduce maintainability and in general to manage the occurring process variants as efficiently as possible, appropriate modeling languages or other approaches for process variant management can be used. Existing approaches for the management of process variants [29] [2] [3] could be integrated to the process engine in this stage.

Van der Aalst et. al [2] identified, based on the workflow patterns [33], configurable pattern that are needed in order to derive variants of a process model from a reference model and thus to create all variants of a process from a single model. However, these patterns aim to represent process variants in a single process model and thus only allow the derivation of variants within a process. The process instantiation, in order to allow a configurable process entry, is not included. Hence, in the following section a new configurable pattern "Configurable Process Start" will be presented, creating a new view on process variant management.

4.1.4.1 Configurable Process Start

Configurable Process Start represents a pattern (Riele and Züllighoven [34] define a pattern as *"the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts"*) allowing to choose the process variant to be executed at the start of the process. The variant is not derived from a single reference model (as in existing approaches), but consists in a separate process model. So, for each process variant of a process family there exists a separate process variant model and a configuration file allows to decide which variant is initialized at process start.

Figure 4.4 demonstrates the typical application of the pattern. An user contacts a worker in order to start a process family (without explicitly deciding for a specific variant). The worker gets the necessary parameters from the configuration file and then instantiates the appropriate process variant.

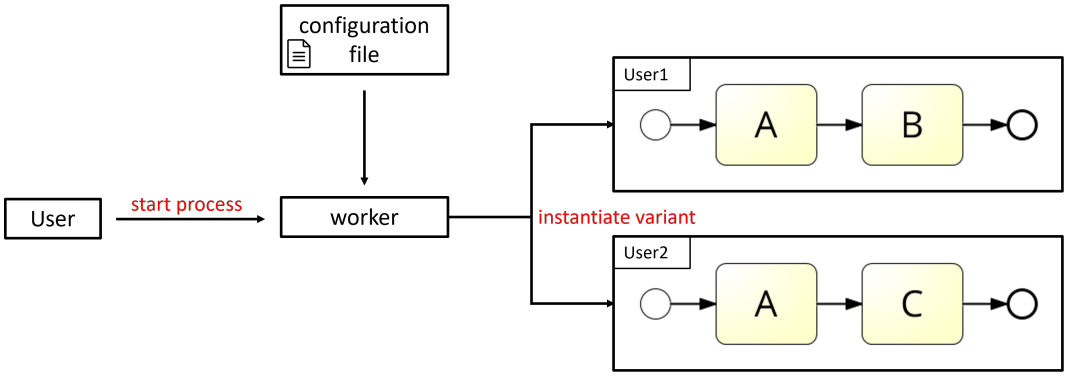


FIGURE 4.4: Pattern "Configurable Process Start"

4.2 Application of PBMC at CP

The scenario of CP is intended to demonstrate the application of PBMC using a real-world scenario. Therefore, the following four sections illustrate the steps of the individual PBMC stages in the use case of CP.

4.2.1 Variation Identification

Since the stage *Variation Identification* represents the identification and delimitation of the variable component and thus the demonstration of the problem definition, this stage was already dealt with in detail in chapter 2 "Scenario". To summarize briefly: All users of SX are basically using the same software, but need different processes (and thus have variation) in their application logic. Hence, the application logic represents the variable component in CP's scenario. A representation of SX with the variable component framed in red can be seen in figure 4.5.

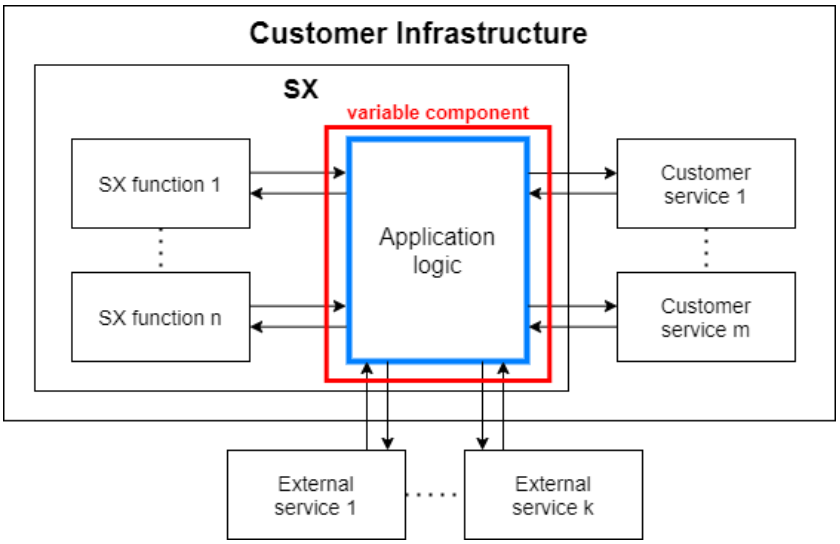


FIGURE 4.5: Stage 1 - Variation Identification

4.2.2 Variation Extraction

As identified in stage 1, in CP's scenario the application logic represents the variable component of SX. This component is therefore extracted from the software and the interfaces are adapted. Functions become services and get an interface to the outside. Figure 4.6 shows SX after the application logic is extracted (without adapted interfaces) and the functions got services. Figure 4.7 shows the final architecture after stage two, with already adapted interfaces.

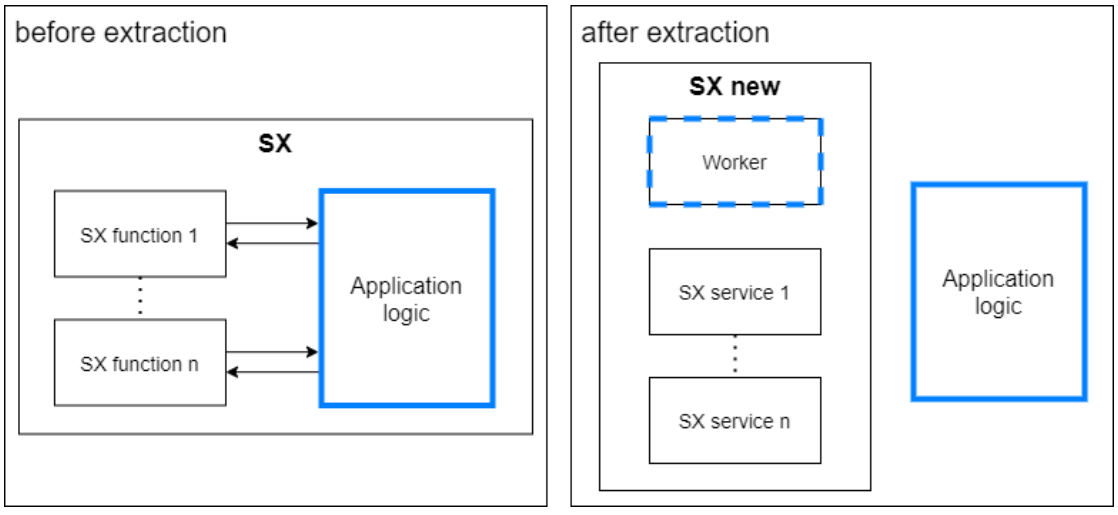


FIGURE 4.6: Stage 2 - Variation Extraction (SX inside view)

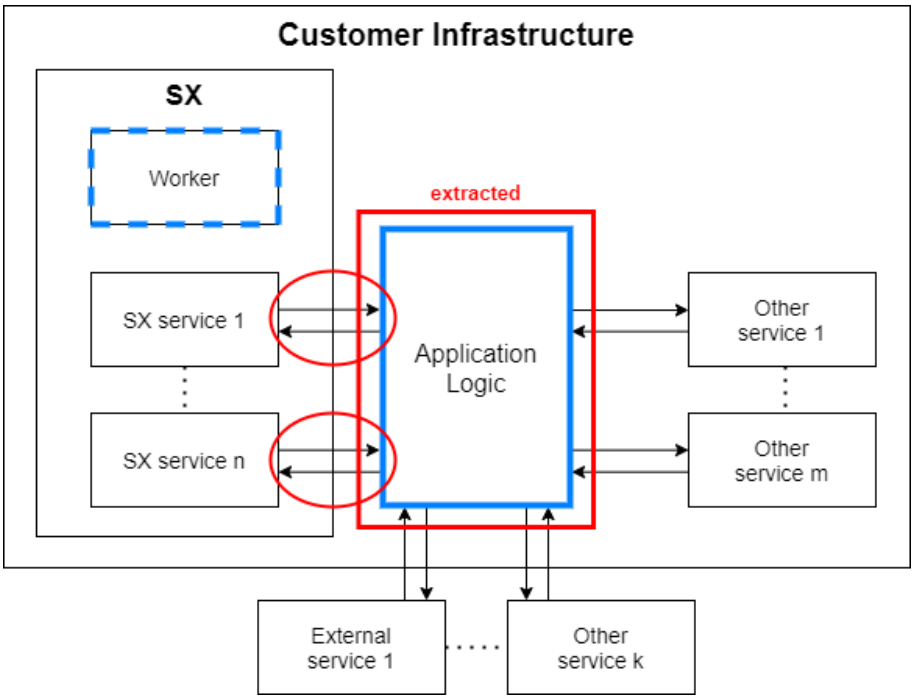


FIGURE 4.7: Stage 2 - Variation Extraction

4.2.3 Augmenting with Process Technology

4.2.3.1 Implementation Process Engine

In the case of CP, a detailed analysis of appropriate process engines was conducted in order to make a clear, justifiable decision which process engine to implement. This selection process represents an important foundation for the successful implementation of PBMC and was therefore described in detail in chapter 5. In the scenario of CP, the Cloud Process Execution Engine (CPEE) [35] was chosen. Therefore, the CPEE was implemented on the respective infrastructures of SX users and the interface for the worker was created.

4.2.3.2 Code to Processes

Afterwards, the previously (in stage 2) extracted application logic is converted to process models. The conversion of source code into process models can be seen in algorithm 1 (source code) and figure 4.8 (associated process model). Algorithm 1 represents a partial sequence of a process (from the sample process in chapter 2) that was defined in the application logic. The process model in figure 4.8 can be used to represent this code. Function calls are replaced by tasks (calling the functions which were transformed to services in stage 2) and the sequence (ifs, decisions, parallelisms, etc.) is replaced by the control flow.

Algorithm 1 Pseudo code of the sample process

```
1:  order ← FetchFull("Order")
2:  if order.status != 1 then
3:      Message("False Order Status")
4:      return
5:  end if
6:  article ← FetchFull(order.article, "Article")
7:  if article.blocked == true then
8:      Message("Article blocked")
9:      return
10: end if
11: UpdateFull(order)
```

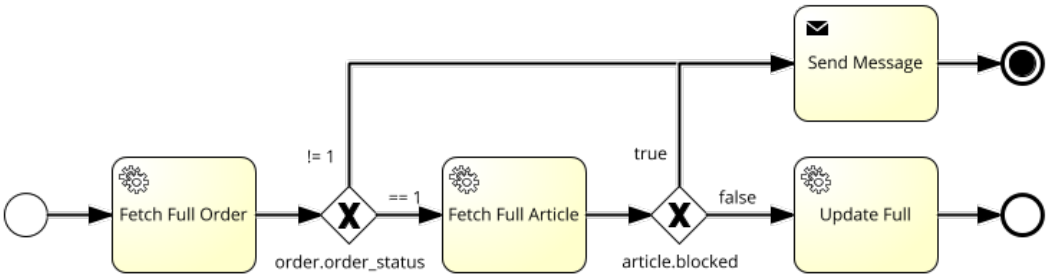


FIGURE 4.8: Resulting BPMN model of the sample process

As a next step, this model is replaced by an executable CPEE model (figure 4.9).

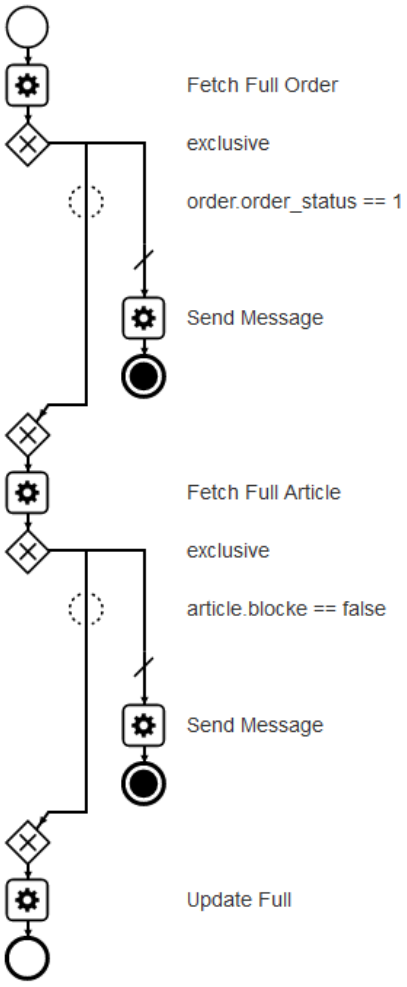


FIGURE 4.9: Resulting executable CPEE Model of the sample process

Transforming each procedure of the application logic into process models turns the entire application logic into a modular element consisting of individual executable process models. This step, like the previous ones, takes place in parallel for all users. Each user naturally has a variety of different process models to transform.

4.2.3.3 Structure Process Models in Repositories

In this last step of the third stage, the process models will be stored in process repositories. CP decided to implement a separate process repository for each user on their respective infrastructure. So, each customer has their own processes in a repository on their own infrastructure.

Since in the last stage "Process Variant Management" no more architectural software changes take place, but only conceptual changes occur if process variant management is needed, figure 4.10 illustrates the final architecture of SX at customers infrastructures, after application of PBMC.

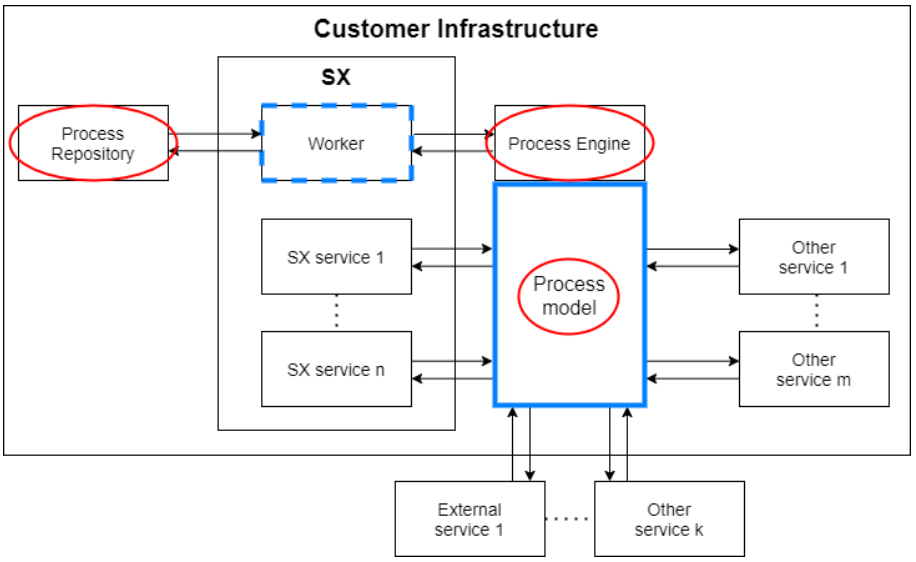


FIGURE 4.10: Stage 3 - Augmenting with Process Technology

4.2.4 Process Variant Management

Since the scenario of CP uses a separate process repository for each user, this stage for handling process variants is not necessary and therefore omitted. Nevertheless, it could still be applied, for example if an user itself has several variants of a process. However, SX users do not make use of this.

Since this stage is an important part of the framework, there will still be illustrated how this stage would have been applied if CP had used a shared repository for all customers. Therefore, figure 4.11 demonstrates how different customers initialize their different processes from a central, grouped repository. Both customers use the exact same SX software and start processes of the same process family (e.g. A or B). The included config file determines which of the variants is instantiated and with which parameters. Thus, it is also possible that processes exist in a single form, used by both customers. By simply changing the config file, other variants can be used and parameters can be changed at any time.

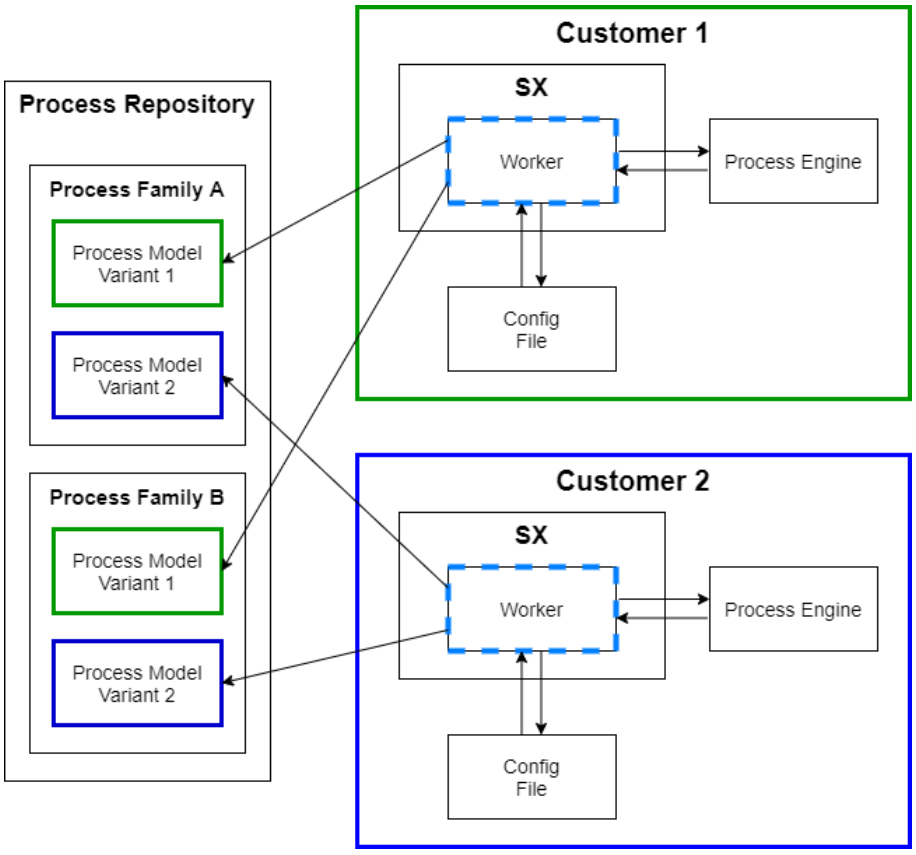


FIGURE 4.11: Stage 4 - Process Variant Management

4.3 PBMC - Results after Application

By applying PBMC, a modern, process-oriented software architecture is formed, having advantages in the production and maintenance of the software as well as in their usage.

The software producer, can use a single standard software for all users without high customization efforts. This fact leads to following advantages:

- **Simple Maintenance:** The maintenance of the software is only done once - in the standard software delivered to all users. There is no need to deal with individualities of different users in the source code of the software.
- **Simple Delivery:** The software only needs to be compiled once and can then be delivered to all users. Different software builds for different users are no longer necessary.
- **Simple Customizing:** No completely new software needs to be customized for new users. The standard software can be delivered to them as well. Customizing is only done in the process level, by configuration of process variants and the associated configuration file.

Beside these advantages in the development and maintenance of the software, the application of process technology and the associated process orientation has also advantages for the users of the software.

- **Process Overview:** Users have a clearer overview of the executed processes by using process models instead of code. Process models can be better interpreted by people and

are therefore understandable to more people than code - which is only understood by programmers. Thus, there is a better awareness of the processes.

- **Fast Response Time to Changes:** If there are any reasons to change processes, this is done very quickly and easily and does not require any changes to the program code, but only to the process model. This means that not even a programmer is needed but only a person who has the know-how to model the processes. Process changes are simple, without much effort and very fast (if necessary even at runtime of an active process).
- **Process Monitoring:** Processes executed in Process Engines can be monitored, providing insights into processes that may not have been available before. This opens up scope for improvements and optimization.

4.4 Classification of PBMC in General Process Variant Management

The final architecture of a software, after the application of PBMC, is characterized by the flexible use of external process models instead of internally wired source code. The process repository can contain different process variants, which are initialized correctly by the pattern "Configurable Process Start", meaning that a certain variant is selected at the process start. Other approaches of process variant management such as PROVOP [29], C-EPC [2] and REMUS [3] could also be used.

In the following sections a comparison between PBMC and already existing approaches for the management of process variants is presented.

4.4.1 Comparison to Existing Approaches

PBMC is compared to PROVOP [29], C-EPC [2] and REMUS [3] in table 4.1.

Table 4.1: Differentiation of variation concepts

Properties	PBMC	C-EPC	PROVOP	REMUS
Reference Model	no reference model	comprehensive	freely selectable	minimal
Type of Configuration	start	restriction	restriction and extension	extension
Software Design Pattern	Mediator pattern	no pattern	no pattern	no pattern
Type	executable	conceptual	conceptual	executable
Realizability	easy	easy	difficult	moderate
Application Scenario	yes	no	no	no
Model Representation	multiple models	single model	single model	multiple models
Modification of Single Variants	easy	moderate	difficult	moderate
Initial Situation	software variants	process variants	process variants	process variants

Reference Model & Type of Configuration: The reference model and the type of configuration are strongly interlinked. The reference model is the central model from which all

variants are derived. There are different possibilities for the extent to which this model is represented. A distinction is made in comprehensive, minimal and freely selectable reference models. Additionally, not having a reference model is also possible (e.g. in PBMC).

The type of configuration is connected to the extent of this reference model. Basically there exist the possibilities *restriction* and *extension*. While comprehensive reference models derive variants by restriction, minimal reference models derive them by extension. Freely selectable reference models offer both, restriction as well as extension. In the case of PBMC, since no reference model exists, neither restriction nor extension is used. The configuration is done using a configurable process start.

Comprehensive & Restriction: The reference model contains all possible variants as different paths. Through configuration options, variants can be derived by restricting paths. Figure 4.12 represents a comprehensive process model. The round gateway with green background represents a configurable exclusive gateway. Through configuration different variants can be derived (as shown in figure 4.14 - using either B, C, or neither).

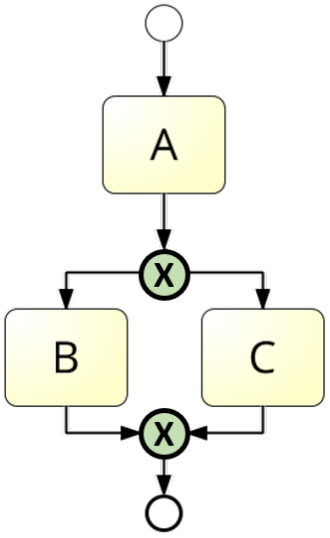


FIGURE 4.12: Comprehensive process model with restriction

Minimal & Extension: Minimal reference models contain only basic sequences that are the same for all variants. The creation of variants is possible by extension as seen in figure 4.13. The round gateway with green background represents an extension point. Here, tasks, sequences, or even subprocesses can be added (according to defined rules) in order to create variants.

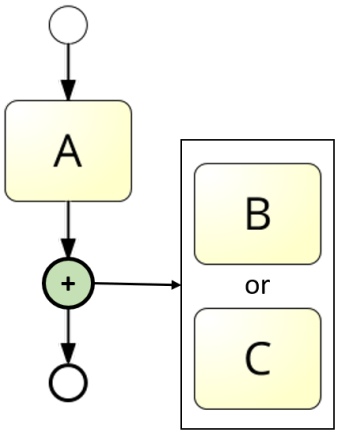


FIGURE 4.13: Minimal process model with extension

Freely selectable (Restriction & Extension): The modeler can freely choose how minimal or comprehensive the reference model should look like. Freely selectable process models allow restriction as well as extension.

No reference model: In the case of PBMC, there is no reference model, since there is a separate model for each variant. Every possible process variant exists in a seperate model and the selection is done directly before execution.

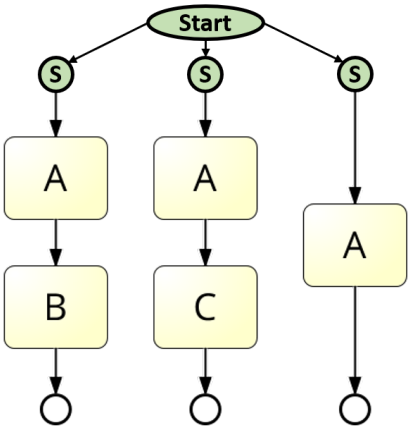


FIGURE 4.14: Multiple process models with the pattern "Configurable Process Start"

Software Design Pattern: This property determines which software design pattern is used in order to enable the process variant approach. If the approach represents just a model configuration property that is enabled as functionality within the modeling tool (or process engine), no software design pattern is used. In the case of PBMC, the process engine can be classified as a mediator.

Type: The type determines whether the approach is merely a concept or whether it actually exists in an executable form (for example, through an sample implementation or whether a certain process engine supports this approach for executable process models). A distinction can be made between the types executable and conceptual.

Realizability: The realizability determines how difficult it is to implement/realize the respective approach (e.g. implement the approach in a real world scenario). A distinction can be made between the difficulty levels easy, moderate and difficult.

Application Scenario: The property "Application Scenario" determines whether the approach has been applied and evaluated in a real world application scenario.

Model Representation: The property "Model Representation" determines how many models are required for the representation of a process family consisting of several process variants in the respective approach. The variants can be represented in single model, or in multiple models.

Modification of Single Variants: The property "Modification of Single Variants" determines how difficult it is to modify a single variant independently of other variants in the process family. A distinction is made between the difficulties easy, moderate and difficult.

Initial Scenario: The property "initial scenario" determines how the initial architecture must look like in order to the apply the respective approach. A distinction is made between process variants and software variants. In the case of process variants, the existence of different process models and thus process orientation is needed. Software variants, on the other hand, must firstly be converted into process variants by restructuring the architecture and thus introducing process orientation.

4.4.2 Perspective of Process Variants

Existing approaches for the management of process variants create variants by extending or restricting a reference model. However, for deriving variants from a reference model, they must have a certain degree of similarity, otherwise the representation of variants by means of a comprehensive or minimal reference model becomes confusing. Figure 4.15 represents three variants of a process and its corresponding reference model. Each of the three variants can be derived from the reference model.

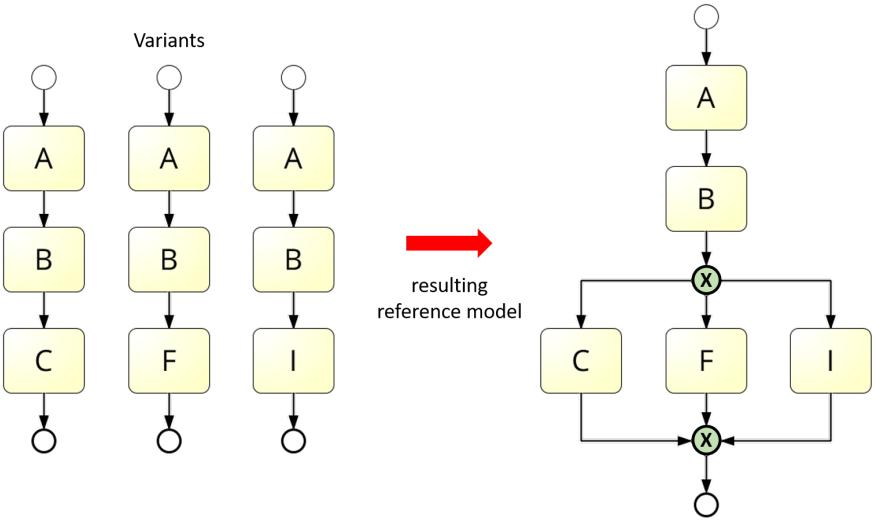


FIGURE 4.15: Example for meaningful process variant management

In the case of a fundamental change to the process, that affects several variants, instead of changing each individual process model, only the reference model needs to be adjusted. Thus, the different variants can be derived. Figure 4.16 represents such a fundamental change.

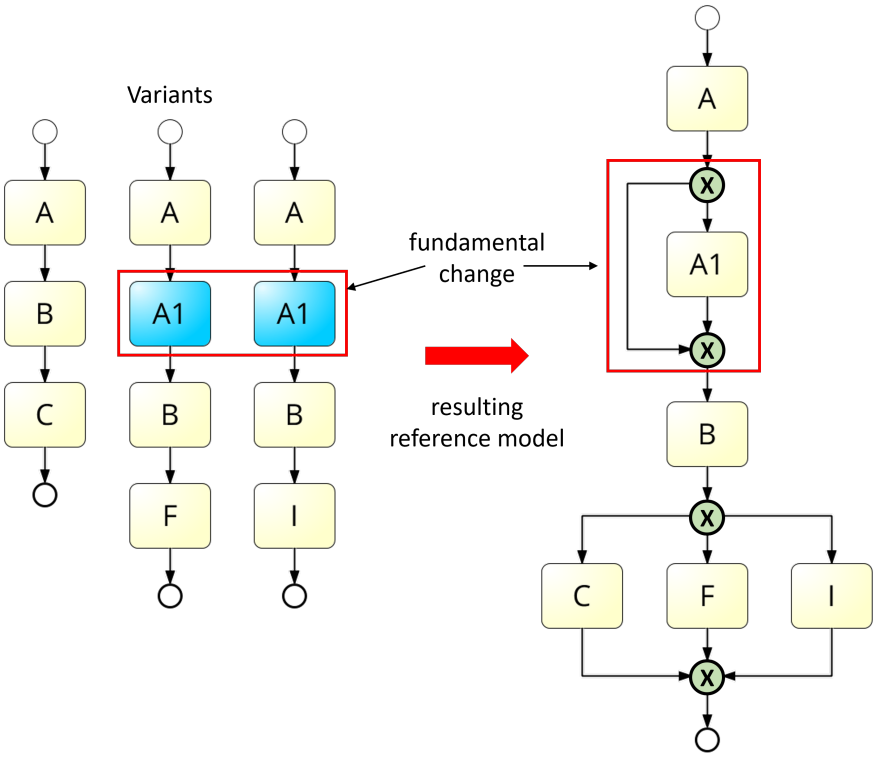


FIGURE 4.16: Fundamental process change at meaningful process variant management

Figure 4.17, in contrast, represents variants that have no similarity and thus are not fundamentally changed together. It can be seen that no advantages are generated by modeling them in a central reference model.

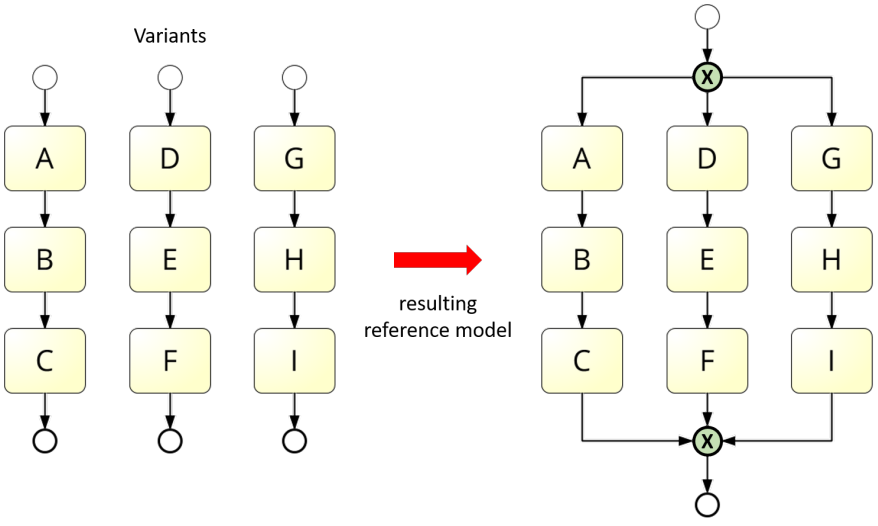


FIGURE 4.17: Example for meaningless process variant management

So, on the one hand, the similarity of the models is an essential point in the management of process variants. On the other hand, the fundamental changes of variants represent an essential point. If processes are regularly fundamentally changed (affecting several variants), an existing approach for the management of variants makes sense. However, if variants affect, for example,

different companies that make completely independent decisions and the variants are never fundamentally changed together, then an approach for deriving variants from a single model does not make sense.

In the case of CP, both points are fulfilled. Firstly, the variants have little or no similarity and secondly, different customers act completely differently and independently of each other. Therefore, an existing approach in which variants are derived from a central reference model is not purposeful. However, the different processes are variants of the same process family and should therefore be conceptually "connected". The *Configurable Process Start* pattern enables exactly this connection - processes of the same family are managed as variants and the correct variant can be selected through the configuration at the process start, but nevertheless they are managed separately from each another.

Chapter 5

Industry Case Study

After the decision for the integration of a BPMS has been made at CP, the necessary information has to be collected in order to find a suitable process engine. Therefore, a detailed evaluation of different process engines was carried out. This chapter starts with the evaluation of five different process engines and then follows with an explanation of the most important decision criteria for CP.

5.1 Evaluation Process Engines

There exist several different process engines which substantially differ in their functionalities. In order to find the most appropriate process engine to handle the variant management of the CP, a detailed evaluation of five process engines based on many different criteria (technical as well as organizational) was done. The evaluated engines are the Cloud Process Execution Engine (CPEE) [35], Camunda[1], jBPM[2], Imixs[3] and Wexflow[4]. In this section the criteria considered for the evaluation will be introduced and explained, the results will be presented and the most important decision criteria for the CP will be demonstrated.

5.1.1 Criteria

Architecture
The architecture of process engines can either be monolithic or service-oriented. As Stuermer et al. [36] showed, current process engines are mostly based on monolithic systems covering different aspects like workflow enactment, control flow, data, exception handling, resources, logging, security and others. In contrast to monolithic process engines, an engine based on a modular service-oriented architecture separates the Workflow Execution Engine, handling the control flow, from the other aspects, allowing dynamic workflows and increased flexibility.

Language
The programming language a process engine is implemented in plays an important role when it comes to customizing. Especially in monolithic systems, which do not offer the necessary interfaces, customizing is often just possible by editing the source code in the respective language.

Interfaces
Interfaces represent the possibilities to communicate with the process engine and its tasks. Interfaces are differentiated into a workflow specific engine API and a business specific API [36]. The

[1] camunda.com
[2] jbpm.org
[3] imixs.org
[4] wexflow.github.io

workflow specific engine API allows communication with the process engine itself. This includes for example the starting and stopping of process instances. The business specific API provides the interfaces which are used by tasks to communicate with external or embedded services. A process engine providing the necessary interfaces for the specific user can reduce the adjustment effort during implementation.

Logging
Process engines could use monolithic logging-frameworks or can also provide logging-interfaces. Using a framework or an interface which allows to create log files in standardized logging-formats like Extensible Event Stream (XES[1]) is an advantage. Logs in standardized formats can directly be used by process mining tools to obtain process analyses without the need of converting the log files.

Rule Engine
A rule engine is a controlling instance working passively in the background. It observes running process instances and holds information the instance may not have. If the process holds wrong information or runs danger of making a wrong decision based on non-existent or incorrect information, the rule engine can intervene the process and make adjustments. Therefore, the rule engine defines business rules that are constantly checked. If a rule condition occurs the resulting consequence is executed. Salatino and Aliverti introduced the Drools Rule Engine [37] and characterized it by supporting the following four patterns:

Decision Points Pattern: The decision of the process engine at a decision point (exclusive gateway) can be taken or overruled by the rule engine. This is a stateless pattern.

Data Validations / Enrichment Task Pattern: The data quality at a specific task can be validated and data can be edited or even enriched by the rule engine. Like the Decision Points Pattern, this is a stateless pattern.

Contextual Based Decisions Pattern: Like in the Decision Points Pattern, the rule engine has the possibility to take or overrule a decision made by the process engine. However, in this pattern the rule engine can include facts of the session context. It is therefore a stateful pattern, offering flexible conditions not only based on the specific process instance information but also in conjunction with the available context.

Multi Process Instance Evaluations Pattern: This pattern allows the rule engine to collect information of multiple running instances and use that information for taking and overruling decisions or changing data in single instances. This allows a coordination of multiple running instances which are related to each other. It is a stateful pattern providing a high-level view of the processes (focus on more than one running process instance).

Compliance Checking
Compliance checking offers the possibility to comply business processes with relevant regulations, laws, or guidelines [38]. This functionality can be executed using a rule engine or external tools and allows the definition of superordinate compliance rules. The running process instance does not know about the existence of these rules. If the process instance breaks a compliance rule, the rule engine can intervene the process and make adaptions (Data Validations / Enrichment Task Pattern - see in criteria "Rule Engine"). Compliance checking allows a separate handling of process rules and compliance rules. Process modelers therefore do not need to know anything about the compliance rules as they can be managed at a higher level.

Additional Engine-Features
These are engine features which are included in addition to the core process engine. For example in some cases there is a decision engine included which allows the visualization and execution of decisions. A feature like this makes the execution of complicated or nested decisions clearer. Desired features that are already contained in the process engine result in less customizing effort.

[1] xes-standard.org

Standardized Runtime Data Stream Analysis
In addition to data that is important for a running process instance, a multitude of other data is generated. Due to performance and other reasons, the process instance handles just data which is necessary for the further operation of the instance. Other data generated (e.g. log data of an invoked microservice) will not be further used but discarded. Standardized Runtime Data Stream Analysis allows to handle these two different data streams separated from each other without discarding the data not necessary for the running process instance.

Automated Runtime Data Visualization
As explained in the previous section Standardized Runtime Data Stream Analysis allows a separate handling of data that is necessary for the current process instance and other data generated (e.g. log data of running plants) . Through Automated Runtime Data Visualization the other data generated can also be visualized in a specific format. This visualization format can be defined in the process engine through an annotation in the process model.

Runtime Modifications
Runtime modifications allow to modify a process instance at its runtime. A differentiation between modifying the structure of an instance and modifying endpoints of tasks of an instance can be done:

Structure: Modifying the structure of a process at runtime, means to make fundamental control flow modifications while the process instance is already running. Therefore, control flow elements [33] supported by the respective process engine can be added, deleted or changed at runtime.

Endpoints: Modifying the endpoints of a process at runtime, means to change the endpoint of one or more tasks whithout changing the structure, while the process instance is already running.

Process Evolution
Process evolution is a standard-mechanism changing the structure of multiple running process instances. If a structure modification is applied, the process evolution proofs all running instances of that process model and applies the structure modification to all instances where it is possible (e.g. where the section of the structure change is not already executed).

Customizing
The difficulty of customizing a process engine to the own requirements is an important criteria. The customizing can be decomposed into the following four characteristics:

Modeling: The adaption of the modeling tools allows the user to modify e.g. symbols of the modeling language.

Protocols: The use of own protocols is important for a smooth integration of the process engine into the current infrastructure.

Logging: In order to evaluate log data with currently used tools, the log files should have a certain format. Therefore, it is important to adapt the way of logging to its own requirements.

Worklists: Since every user of process engines has different needs for worklists it is important that an adaption is possible.

License
The license under which a process engine is published plays an important role in embedding it in current systems as well as in the extensibility of the engine.

Additional UI-Features
Additional user interface features (e.g. worklists or analysing cockpits) delivered with the process engine could reduce the workload of adaption if they are suitable and can be used.

Community, Documentation
A large community and a good documentation can make it easier for users of process engines to integrate independently as well as help to solve future problems.

Professional Support
Professional support can greatly assist and facilitate the integration as well as the maintainability of a process engine but is usually associated with high costs.

Employees
The administration of a process engine leads to increased know-how requirements and can therefore make it necessary to hire new employees. This requires to know about how to find employees with the appropriate knowledge.

Operating System
In order to run the process engine on the corresponding servers without making fundamental changes, the operating system the process engine is executable on plays of course a crucial role.

Memory Utilization
Since many companies only have limited memory, the memory utilization can be a decisive criteria.

Scalability
While monolithic systems mainly run single-core, service-oriented approaches offer the possibility to run multi-core. Multi-core systems guarantee a higher flexibility as well as performance improvements.

Frequency of New Versions
An active development is a prerequisite for the use of a process engine. Therefore, the frequency of new versions is an important indicator.

Costs
The direct costs arising from the integration of a specific process engine are of course essential. A differentiation into costs for the licensing and costs for the customizing can be done:

Licensing: While licensing of open source engines is free, commercially distributed engines can cause high costs.

Customizing: Under some licenses, customizing is only possible if certain criteria are met, which can lead to additional costs.

5.1.2 Results

Table 5.1: Evaluation Results of CPEE, Camunda and jBPM

Criteria	CPEE	Camunda	jBPM
Architecture	service-oriented architecture	monolithic	monolithic
Language	Ruby	Java	Java
Interfaces			
Business API	REST	REST, Java	REST, Java
Engine API	REST	REST, Java	REST, Java
Logging	XES, Elasticsearch[1], external services	SLF4J[2], proprietary	SLF4J[2], XES export, proprietary
Rule Engine	any external via REST API (e.g. Drools)	No, but integration via Java possible	Drools (DRL)
Decision Points Pattern	Yes	No	Yes

Continued on next page

[1] elastic.co
[2] slf4j.org

Table 5.1 – *Continued from previous page*

Criteria	CPEE	Camunda	jBPM
Data Validations /Enrichments Task Pattern	Yes	No	Yes
Contextual Based Decisions Pattern	Yes	No	Yes
Multi Process Instance Evaluations Pattern	Yes	No	Yes
Additional Engine-Features	None	DMN Decision Engine	DMN Decision Engine
Compliance Checking	external Tools	No	via Rule Engine
Standardized Runtime Data Stream Analysis	Yes	No	No
Automated Runtime Data Visualizing	Yes	No	No
Process Evolution	Yes	can be retrofitted	can be retrofitted
Runtime Modifications			
Structure	Yes	Yes	Yes
Endpoints	Yes	No	No
Customizing	easy	difficult	very difficult
Modeling	XML-based Themes, Javascript	Modeler-Plugin, Javascript	Java, Javascript
Protocols	Webservices in any language	Java	Java
Logging	Webservices in any language	Java	Java
Worklists	Webservices in any language	Java	Java
Lines of Code			
Engine	CPEE: 2.859 (*.rb) WEEL: 1.084 (*.rb)	267,117 (*.java)	398,291 (*.java)
User Interface	2,371 (*.js)	20,546 (*.java) 104,382 (*.js)	172,802 (*.java) 47,901 (*.js)
License	LGPL[1]	Apache 2.0[2]	Apache 2.0[2]
Additional UI-Features	None	Tasklist, Cockpit	Tasklist, Cockpit
Community, Documentation	Community CDP/ University of Vienna, Examples	Online Documentation, Videos	Online Documentation, jBPM Developer Guide
Professional Support	CDP	Camunda (Enterprise Edition)	Red Hat (Enterprise Edition)
Employees	Cooperation with University of Vienna - graduates	Recruiting	Recruiting
Operating System	Windows, MacOSX, Linux	Windows, MacOSX, Linux	Windows, MacOSX, Linux
Memory Utilization			
after Startup	35 MiB	1.1 GiB	4 GiB
per Instance	1 MiB	3 MiB	5 MiB

Continued on next page

[1] www.gnu.org/licenses/lgpl-3.0.html
[2] apache.org/licenses/LICENSE-2.0

Table 5.1 – *Continued from previous page*

Criteria	CPEE	Camunda	jBPM
Scalability	Multi-Core	Single-Core	Single-Core
Frequency of New Versions	every 2-3 weeks	every 6 months	every 3 weeks
Costs			
Licensing	free	Community: free Enterprise: request	Community: free Enterprise: request
Customizing	free	free	free

Table 5.2: Evaluation Results of Imixs and Wexflow

Criteria	Imixs	Wexflow
Architecture	monolithic	monolithic
Language	Java	C#
Interfaces		
Business API	Java	C# (*.dll files)
Engine API	REST, Java	REST, C#
Logging	Java Logging[1], proprietary	log4net[2], proprietary
Rule Engine	No	No
Decision Points Pattern	No	No
Data Validations /Enrichments Task Pattern	No	No
Contextual Based Decisions Pattern	No	No
Multi Process Instance Evaluations Pattern	No	No
Additional Engine-Features	BPMN Decision Engine	None
Compliance Checking	No	No
Standardized Runtime Data Stream Analysis	No	No
Automated Runtime Data Visualizing	No	No
Process Evolution	No	No
Runtime Modifications		
Structure	No	No
Endpoints	No	No
Customizing	difficult	medium
Modeling	Java	C#
Protocols	Java	C#
Logging	Java	C#
Worklists	Java	C#
Lines of Code		
Engine	52,121 (*.java)	10,302 (*.cs)
User Interface	1,253 (*.java) 21,770 (*.js)	6,599 (*.js)

Continued on next page

[1] imixs.org/doc/logging.html
[2] logging.apache.org/log4net

Table 5.2 – *Continued from previous page*

Criteria	Imixs	Wexflow
License	GPL v2[1]	MIT License[2]
Additional UI-Features	None	None
Community, Documentation	Online Documentation	Online Documentation
Professional Support	Imixs Software Solutions	Akram El Assas (Project Manager), Issue Board
Employees	Recruiting	Recruiting
Operating System	Windows, MacOSX, Linux	Windows, MacOSX, Linux, Android, iOS
Memory Utilization		
after Startup	0.6 GiB	60 MiB
per Instance	10 MiB	1 MiB, depending of individual tasks (used libraries)
Scalability	Single-Core	Single-Core
Frequency of New Versions	regular updates (several times a month)	every few months
Costs		
Licensing	free	free
Customizing	only with GPL licensed software under the GPL license	free

5.2 Decision Criteria for CP

Since the CP works partially on virtual machines with limited resources, a low memory utilization would be desirable. Therefore, the CPEE and Wexflow look very promising because of their resource-saving approach.

Since the current infrastructure of CP was developed in C# and therefore the greatest know-how is available in this programming language, it would be preferable not to deviate from it. Hence, Wexflow would be very interesting, because it can be integrated into the current infrastructure with existing know-how. The service-oriented approach of the CPEE would also be appropriate, since that process engine can be customized through self-written webservices in any desired language (e.g. C#).

Additionally, support would also be desireable for the CP in order to get assistance during and also after the integration. Since the necessary know-how for process engines and their application is not available, this is also an important decision criteria. Therefore, the CPEE would be well suited, because it is in active development by the project partner "Austrian Center of Digital Production"[3] who could provide access to professional support.

In the end the CP decided to integrate the CPEE. This decision was based on the decision criteria given in this section.

Note: In the course of this case study, criteria for the evaluation of different business process management systems were developed. The business process management systems mentioned in this chapter were evaluated on the basis of these criteria. This evaluation served as a basis for CP to choose an appropriate process engine for their implementation. None of the five evaluated process engines is "better" or "worse" than another, but they merely support different concepts and are therefore differently suited for different use cases. However, the decision for an appropriate process engine was made by CP and was neither supported nor confirmed nor evaluated in the course of this thesis.

[1] www.gnu.de/documents/gpl-2.0.de.html
[2] opensource.org/licenses/MIT
[3] acdp.at

5.3 CP using CPEE

CP uses the selected process engine CPEE as specified in the solution design (see chapter 4). Currently, the implementation of PBMC using the CPEE is in a test phase where a single process (sample process "Approval" from chapter 2) is implemented at a single customer. In the further course, more processes will be implemented as well as more customers included. The objective of CP is to configure any variability of all SX users by executable process models. Upon successful completion of the implementation, the CPEE can be used to enable further configurations of other software or to map internal company processes.

Chapter 6

Protoypical Implementation

In order to demonstrate the final architecture of a software (like SX) after the application of the framework PBMC with its pattern "Configurable Process Start", a sample implementation called "Inkscape-Manager" is presented in this chapter. The architectural change of SX through the application of PBMC is shown in chapter 4 "Solution Design". The initial scenario, in contrast, is presented in chapter 2 "Scenario". Instead of plants being managed by SX, as shown in the case of CP, this implementation demonstration uses the software Inkscape, which will be managed by the developed software "Inkscape-Manager". Via executable CPEE process models, which can exist in different variants and are instantiated by the pattern "Configurable Process Start", objects (QR and Aztec codes) are inserted into Inkscape in the form of an SVG (Scalable Vector Graphics) file. The sequence that is enacted while object insertion is defined in executable process models. Different variants of these process models lead to different process sequences and to different objects. The results and process sequences can be easily and quickly adapted in the process models or different process variants can be selected by changing the configuration file.

FIGURE 6.1: QR code, Inkscape, Aztec code

This chapter therefore presents the entire architecture and its individual components in detail. Furthermore, process flows and results are presented and compared with those of the solution design.

6.1 Architecture

This section represents the entire architecture of the Inkscape-Manager and thus a final sample architecture of a software after the application of PBMC with its Configurable Process Start. The main goal of the architecture is to enable a uniform software that nevertheless allows dynamically changing process variants for different users. In this sample implementation, the

uniform, service-oriented software is called "Inkscape-Manager". It includes all necessary functionalities, in order to enable the dynamic and configurable instantiation of processes using the pattern "Configurable Process Start". Regardless of the user and its specific process variants, the Inkscape-Manager remains consistent and has no need to be modified or adapted. Modifications are only implemented by adapting the process models.

Figure 6.2 shows the whole architecture of this sample implementation. The central component is the Inkscape-Manager, which is divided into worker and other functionalities (including the Graphical User Interface and other services). The worker provides an interface to Inkscape and accesses the interfaces of the config file, the process repository and the CPEE. The functionalities provide interfaces that are tangible from the currently executed process models. The process model uses an interface provided by the CPEE. More detailed explanations of each component are provided in the following subsections.

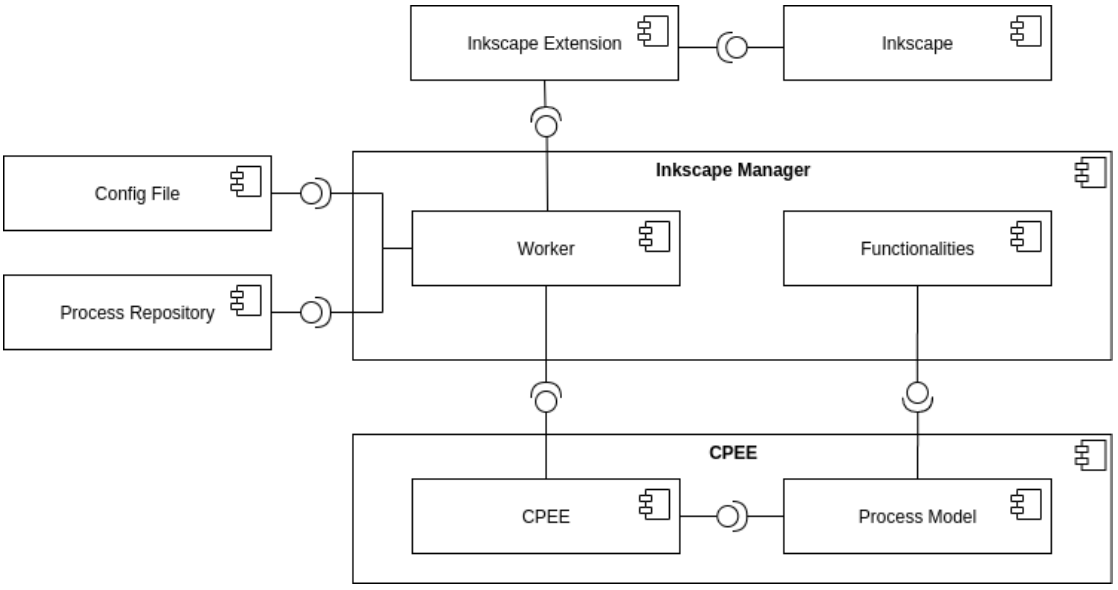


FIGURE 6.2: Component diagram - Inkscape-Manager

6.1.1 Inkscape-Manager

The Inkscape-Manager is a central, service-oriented operation unit, which consists of services (called functionalities) instead of functions, using a new architecture (after application of PBMC). It is a Python3 Flask application running on *https://cpee.org/inkscape-manager*. It contains all functionalities that can be called in the course of a running process - this includes, for example, the worker and the GUI (both are described in detail in the following subsections), but also other functionalities that are called as endpoints by running processes. The worker was specifically mentioned in the component diagram (figure 6.2), since it has a special function with initiating the (configurable) process start. Other services have been combined under the component "functionalities". In order to get a brief overview, some of them are briefly described below:

User Message: This endpoint can be called by the CPEE in order to leave a message for the user in the GUI. It receives the request parameter "message" as data type "string" (e.g. "hello world") and sends it to the GUI, which displays it to the user. The executed process continues without stopping - therefore, an user message does not require any confirmation by the user in order to proceed the process.

Table 6.1: Service: User Message

Service Name	User Message
Request Method	POST
Request URL	cpee.org/inkscape-manager/user_message
Request Parameter	message = "hello world"

Change Color: This endpoint can be called from the CPEE in order to change the color of the created object. It receives the object, that is already stored in the CPEE data elements as a "string" (this is, however, an SVG XML string) and adds the variable "color" which was also passed (the default color would be black). The color can be passed either as a string ("red", "blue", "orange", etc.) or as a hex-code (#777777, #bbbbbb, etc.). After the functionality is processed, the updated SVG object is returned and the process continues.

Table 6.2: Service: Change Color

Service Name	Change Color
Request Method	GET
Request URL	cpee.org/inkscape-manager/changecolor/
Request Parameter	object = "<svg> ... </svg>" color = "red"
Request Method	GET
Response Body	object = "<svg> ... </svg>"

User Confirmation: The endpoint "user confirmation" receives a string "confirmation", which represents the question or text the user should confirm (e.g. Abort process?). This text is forwarded to the GUI, where a confirmation window opens. A response with the header "CPEE-CALLBACK" = "true" is sent to the CPEE, which stops the CPEE until an user answer is sent to the callback url.

Table 6.3: Service: User Confirmation

Service Name	User Confirmation
Request Method	POST
Request URL	cpee.org/inkscape-manager/user_confirmation/
Request Parameter	confirmation = "Stop process?"
Response Body	headers: "CPEE-CALLBACK" = "true"

Inkscape: The endpoint "Inkscape" receives a string named "object" that contains the SVG object created and manipulated within the running process. Within this functionality, the transmitted SVG object will be stored on the file system in a file called "object.xml". The worker constantly checks the file system for exactly this file and immediately passes it to Inkscape as soon as it exists.

Table 6.4: Service: Send to Inkscape

Service Name	Send to Inkscape
Request Method	POST
Request URL	cpee.org/inkscape-manager/inkscape/
Request Parameter	object = "<svg> ... </svg>"

6.1.1.1 Worker

The worker represents a conceptual very important role in this sample implementation. It is responsible for the realization of the developed concept "Configurable Process Start". The

worker can be started by receiving an impulse in form of a GET call (see table 6.5).

Table 6.5: Call to Worker

Request Method	GET
Request URL	https://cpee.org/inkscape-manager/worker/
Request Body	process = "pbmc"
Response Body	entweder: object = svg ... oder: object = "terminated"

After the worker has been contacted, it starts the desired process family. Which variant and which parameters will be used for this process start, is regulated by the worker under use of the pattern "Configurable Process Start". Therefore, the logic of the Configurable process start is implemented in the course of the worker. It searches the necessary information in the configuration file. Using that information, it can fetch the correct process model from the process repository and send it to the CPEE in order to start the process. The sequence diagram in figure 6.3 shows the flow of the worker when starting a process.

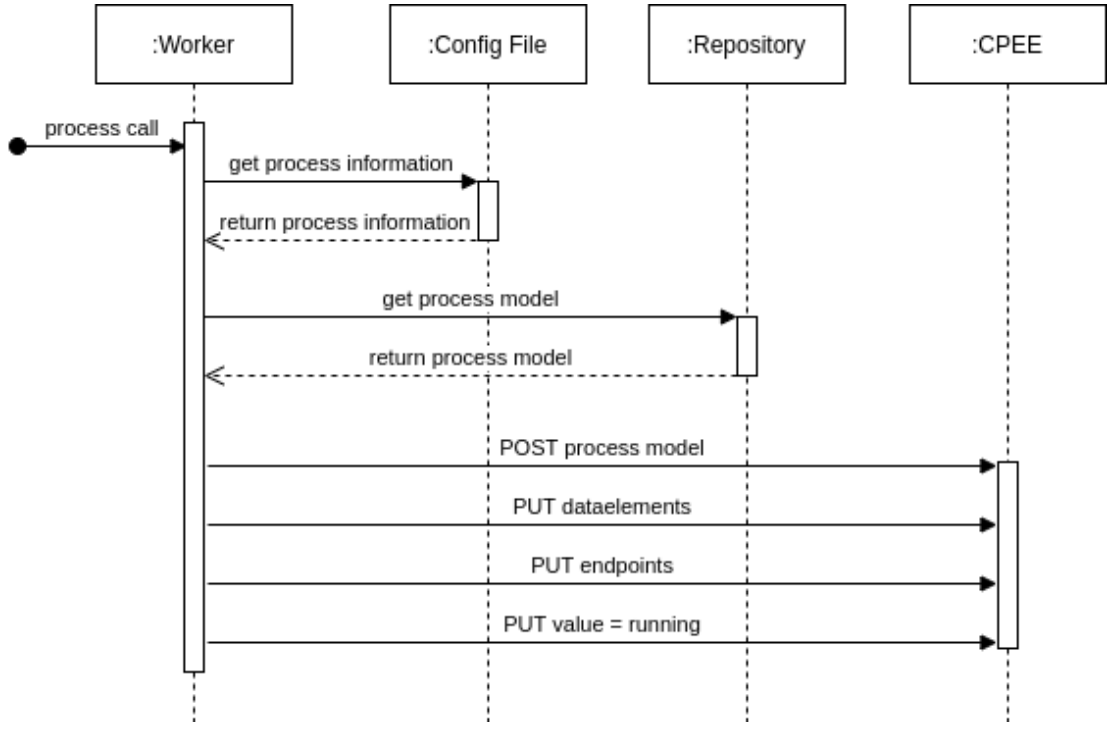


Figure 6.3: Sequence diagram - worker starting a process

First a process call comes from outside - it does not matter where exactly the call comes from. In this sample implementation, the call comes directly from an Inkscape extension (see section 6.1.2.1 "Inkscape Extension"). This call only contains the process family that should be started (e.g. "insert object").

In order to determine the variant to be selected, as well as its data elements and endpoints, the worker calls the config file where this process information is stored (for more detailed information on the config file, see section 6.1.5 "Configuration File").

After the worker successfully found the necessary process information, it can fetch the variant to be started (the respective process model) from the process repository (for more detailed information on the process repository, see 6.1.4 "Process Repository").

POST Process Model: The POST call to the CPEE instantiation service contains the process model in the body, as well as information about whether the process should start directly after instantiation or not. Passing the parameter *behavior = fork_ready* means that the process is instantiated but not yet started. In the response body, information about the instantiated instance (ID, UUID, etc.) is received. This information can be used for the further interaction with the running process instance.

Table 6.6: POST process model

Request Method	POST
Request URL	https://cpee.org/flow/start/xml/
Request Body	behavior = fork_ready xml = process.xml
Response Body	{"CPEE-INSTANCE": "1001", "CPEE-INSTANCE-URL": "https://cpee.org/flow/engine/1001", "CPEE-INSTANCE-UUID": "2dffe5fc-06eb-4558-b8b5-1c35a06b2916", "CPEE-BEHAVIOR":"fork_ready"}

PUT data elements / endpoints: The CPEE-INSTANCE-URL received from "POST process model" can now be used to address the process on the server. In this call, the data elements and endpoints (for more detailed information on data elements and endpoints see section 6.1.3.1) are added to the already instantiated CPEE process model, in separate calls.

Table 6.7: PUT data elements / endpoints

Request Method	PUT
Request URL	https://cpee.org/flow/engine/1001
Request Body	<dataelements xmlns="http://cpee.org/ns/properties/2.0"> <key1>value1</key1> <key2>value2</key2> </dataelements>

PUT *value = running*: This PUT call transmits the information that the process should be started to the CPEE.

Table 6.8: PUT value = running

Request Method	PUT
Request URL	https://cpee.org/flow/engine/1001
Request Body	value = running

After the entire process has been instantiated with the desired parameters and is finally started, the worker is waiting for news from the process. Therefore, it checks the following things in a loop:

1. It checks every 0.5 seconds whether a file "object.xml" has been created on the file system. (This would be the ideal case, because then the process is finished and the object can be transferred to Inkscape.)
2. It checks every 5 seconds whether the process instance has already been finished or terminated, by a direct request to the running process instance (see table 6.9). (In this case the process was terminated without creating a file "object.xml". This can happen, if the process is terminated - intentionally or if an error occurred.)

Table 6.9: GET state

Request Method	GET
Request URL	https://cpee.org/flow/engine/1001/properties/state
Response Body	ready *or* running *or* stopped *or* finished *or* abandoned

Depending on which of these two cases has occurred, the worker returns either (1) the SVG code it found in the file "object.xml", or (2) a message that the process has ended without being able to pass an object.

Listing 1 represents the whole source code of the worker with detailed comments. The function call "funcs.start_process()" in line 3 includes the whole process start presented in figure 6.3. Afterwards, the loop checking for process updates starts and checks if "object.xml" exists or the process was finished. Thereafter, the GUI is updated and the object is returned to Inkscape (if no object exists, a string "terminated" is returned).

```
1  @app.route('/worker', methods=['GET'])
2  def worker():
3      instance_header = funcs.start_process(request.args.get('process'))
4
5      #check if process is still running
6      i = 0
7      while(True):
8          #update counter
9          i = i+1
10
11         #check if object.xml file already exists (object.xml contains the svg
       -code after the task /inkscape)
12         if(os.path.exists("object.xml")):
13             break
14
15         #every tenth time check the CPEE Engine if process is already
       finished or abandoned (in case that the process was terminated or
       abandoned)
16         if(i%10 == 0):
17             state = rest.get(instance_header["CPEE-INSTANCE-URL"]+'/
       properties/state').text
18             if(state == "finished" or state == "abandoned"):
19                 break
20
21         #wait 0.5 seconds before rerunning the loop
22         time.sleep(0.5)
23
24     #update GUI to "FINISH"
25     with open('gui_information.json') as json_file:
26         data = json.load(json_file)
27     data['CPEE-STATE'] = " - finished"
28     with open('gui_information.json', 'w') as outfile:
29         json.dump(data, outfile)
30
31     #return the svg-object as string from object.xml (in case object.xml
       doesn't exist, return "terminated")
32     _object = "terminated"
33     if(os.path.exists("object.xml")):
34         with open('object.xml', 'r') as file:
35             code = file.read()
36         os.remove('object.xml')
37     return _object
```

Listing 1: Source code of the worker

6.1.1.2 Graphical User Interface

In this sample implementation, the Graphical User Interface (GUI) is part of the functionalities of the Inkscape-Manager. It is responsible for mediating interactions between the running process model and the user. In addition, it provides the user with an overview of the currently running process (or last running process, if no process is currently running).

Figure 6.4 shows the GUI that is displayed while a process is running or after a process run through (in this case, the process has already finished). In the upper blue window, messages for the user can be seen. In the lower green window, the process history of the process is shown. In the lowest small yellow window, the currently selected variant is stated.

The linked heading at the top opens the CPEE-GUI and monitors the current process.

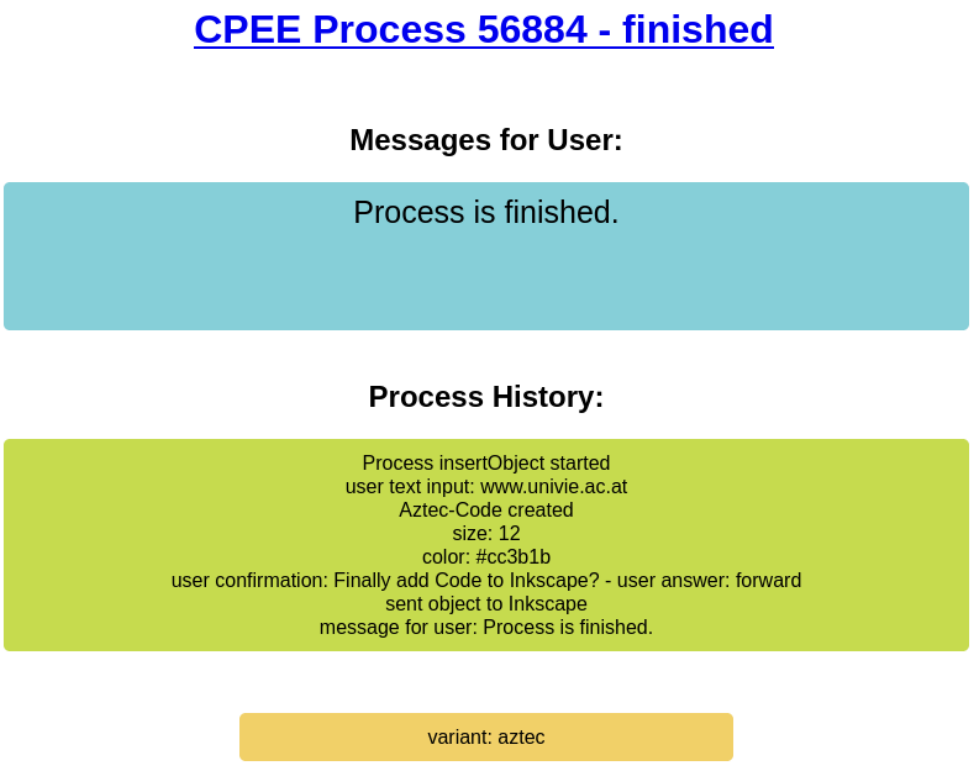


FIGURE 6.4: Inkscape-Manager GUI

The GUI for this sample implementation allows following user interactions:

User Messages: User messages are simple messages to the user that are shown directly to the display (see figure 6.4, blue window). The process does not pause and does not require any information or confirmation by the user. User messages, therefore, only serve to inform the user about a particular state.

Text inputs: Text inputs are opened in a pop-up and require a response from the user. The process stops and holds until the user returns a response. Figure 6.5 shows the GUI when a text input is required. Text inputs are used, for example, at the beginning of a process when the text for a QR code is not to be taken from default values, but should be manually added.

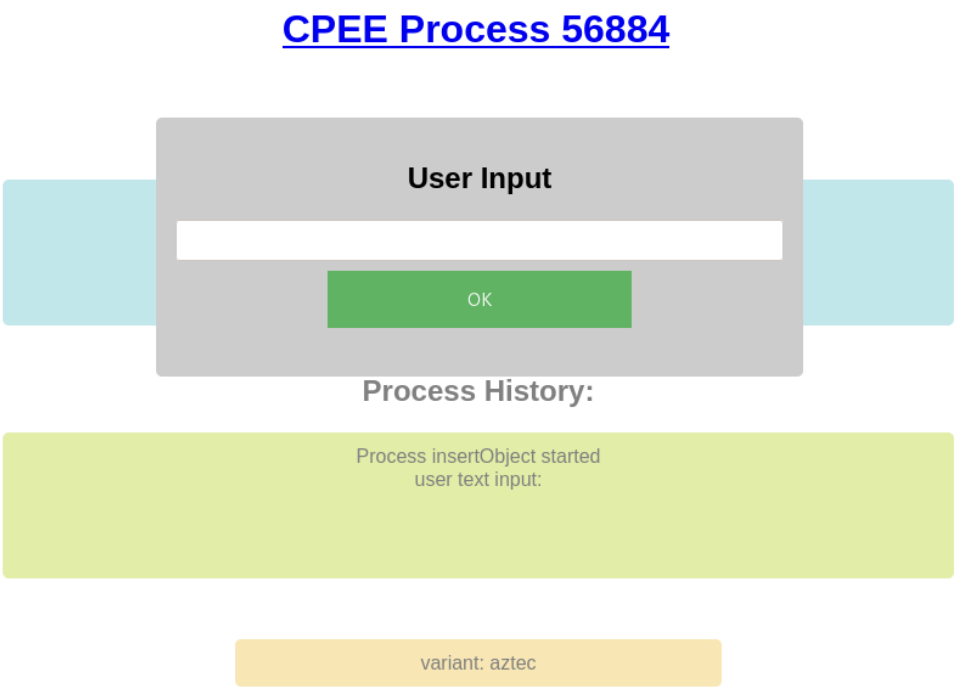


FIGURE 6.5: GUI - text input

Confirmations: Confirmations are opened in a pop-up (like text inputs) and pass a text or question to the user, who must either confirm or reject it (e.g. "Finally add Code to Inkscape?"). The answer is then returned to the process, which may lead to variations in the further process execution. Figure 6.6 shows the GUI when a user confirmation is displayed.

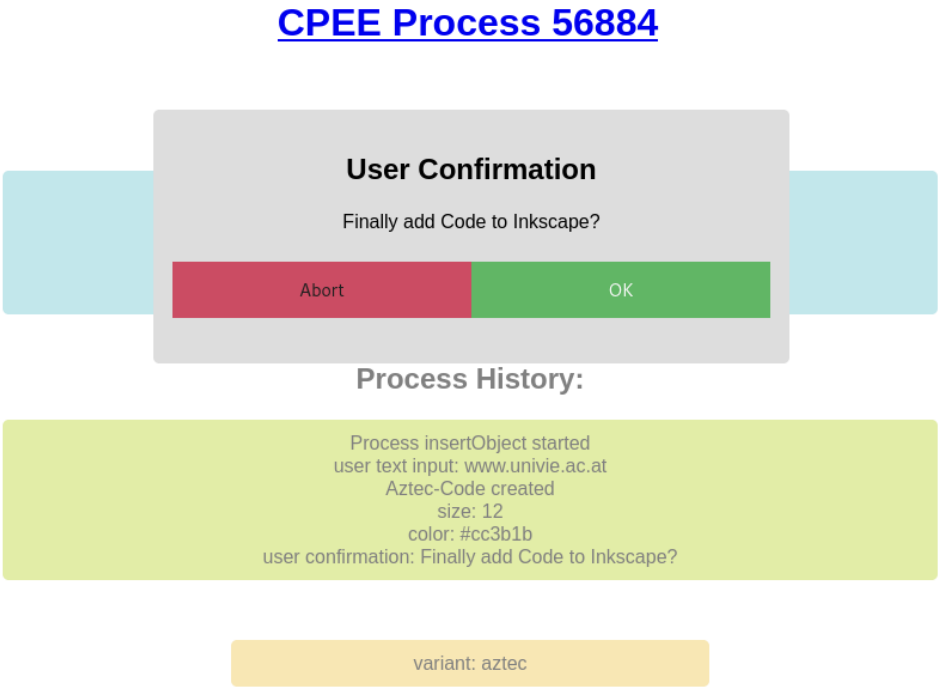


FIGURE 6.6: GUI - user confirmation

6.1.2 Inkscape

Inkscape[1] is a professional drawing software for editing vector graphics and is executable on the operating systems Windows, Mac OS X and GNU/Linux. It is a free and open source software which is licensed under the GPL[2] and uses the free standard SVG (Scalable Vector Graphics).

In this sample implementation, Inkscape should represent a plant that is "controlled" by process technology by means of the Inkscape-Manager with its associated functionalities. In order to state a direct comparison with the use case of the company CP (see chapter 2 "Scenario"), Inkscape represents the "plant" which is controlled by the central, monolithic software (SX in the case of CP). The goal of an executed process in this implementation is therefore always to achieve desired effects in Inkscape (e.g. insertion of an object). In comparison, the goal of the sample process of CP is to achieve effects in the plant (e.g. storage of grain).

6.1.2.1 Inkscape Extension

Inkscape provides an interface to attach self-developed extensions directly to the software. This possibility was used in the course of this implementation demonstration.

Inkscape extensions consist of two files. An .INX file which extends the Inkscape-GUI with the new functionality and a .PY file which contains the source code to be executed.

The .INX file consists of XML code and determines the category and name of the extension. Furthermore, the .INX file allows to pass input parameters via the Inkscape GUI (however, this functionality was not necessary in our case and therefore is not used).

Listing 2 shows the .INX file and figure 6.7 the corresponding Inkscape-GUI entry.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <inkscape-extension xmlns="http://www.inkscape.org/namespace/inkscape/
      extension">
3   <name>Insert Object</name>
4   <id>org.inkscape.template.effect</id>
5 <label name="description" xml:space="preserve">
6 This extension allows inserting an object (QR or Aztec) on basis of process
      technology and process variant concepts.
7
8 Please open https://cpee.org/inkscape-manager for navigating through the
      running process (current variant can be seen in the yello window).
9
10 Have fun!
11 </label>
12   <effect needs-live-preview="false">
13     <effects-menu>
14       <submenu name="Process"/>
15     </effects-menu>
16   </effect>
17   <script>
18     <command location="inx" interpreter="python">insert_object.py</command>
19   </script>
20 </inkscape-extension>
```

Listing 2: Inkscape extension - .INX file

[1] https://inkscape.org/
[2] https://www.gnu.org/licenses/gpl-3.0.en.html

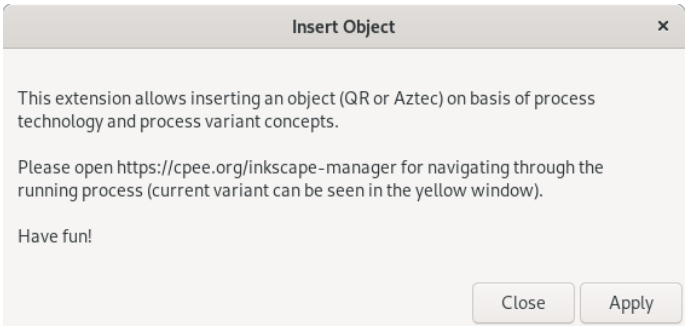


FIGURE 6.7: Inkscape extension GUI entry

To start the extension, the menu "Extensions" can be used. The whole path is "Extensions/Process/Insert Object (see figure 6.8).

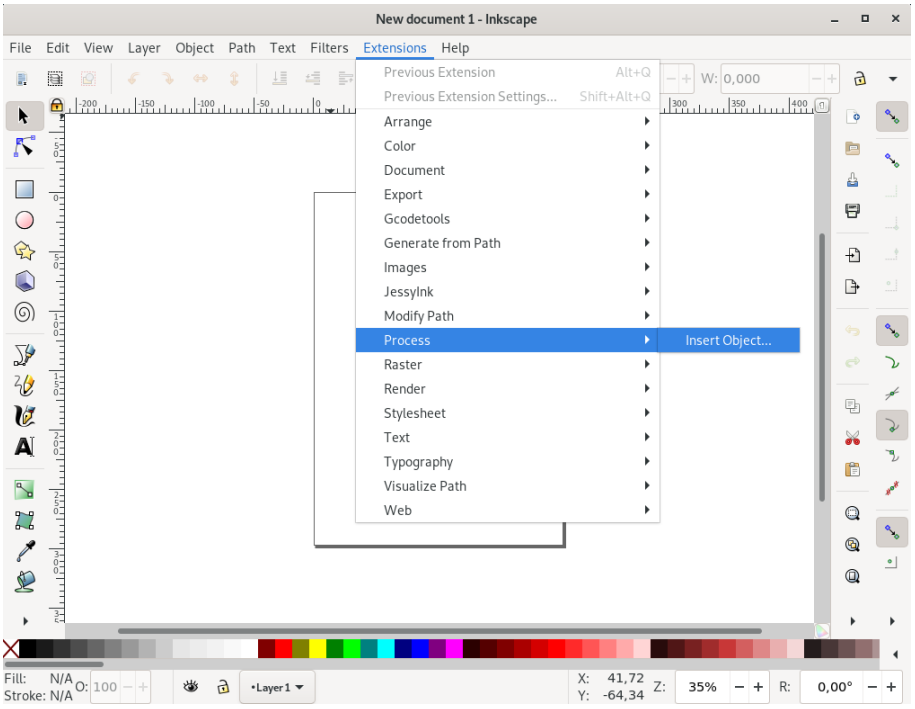


FIGURE 6.8: Inkscape - Path for starting Inkscape-Manager

The .PY file consists of Python code that allows communication with the Inkscape SVG file using the library "inkex"[1]. Listing 3 shows the entire source code of the Inkscape extension. Since all logic is located in the CPEE process models, the Inkscape extension code is very simple. It is a call to the worker (line 11), which returns either the SVG object to be inserted or information that no object exists. If an SVG object is returned, it is inserted into the current layer of the currently opened SVG file (lines 16 and 18). If no SVG object is returned, an exception is thrown (lines 13 and 14) informing the user that no SVG object was passed by the process.

```
1 import inkex
2 import requests as rest
3 from lxml import etree
4
5 class InsertObject(inkex.EffectExtension):
6
7     def effect(self):
```

[1]https://pypi.org/project/inkex/

```
8
9         data = {'process':'insertObject'}
10
11        code = rest.get('http://cpee.org/inkscape-manager/worker',data).text
12
13        if(code == 'terminated'):
14           raise Exception('Process got terminated!')
15
16        layer = self.svg.get_current_layer()
17
18        layer.add(etree.fromstring(code))
19
20 if __name__ == '__main__':
21     InsertObject().run()
```

Listing 3: Inkscape extension - .PY file

6.1.3 Process Engine - CPEE

The process engine used is the Cloud Process Execution Engine (CPEE). It enables the execution of process models and is therefore an important cornerstone of this implementation. Generally, any process engine can be used. However, since the CPEE is also used in the use case of CP and this implementation should be as close as possible to that of CP, the decision was also made in favour of the CPEE here.

As already described in chapter 4 "Solution Design", the Process Engine (more precise the process model it executes) acts as a mediator between the functionalities. Any sequence of operations and communication between the functionalities only takes place via it.

The CPEE consists of three parts, each running on different addresses:

- The *Engine* runs on https://cpee.org/flow/engine/. It is the central unit and responsible for the flow of processes.
- The *Instantiation Service* runs on https://cpee.org/flow/start/. It is used to instantiate process models. After instantiating a process model via */flow/start/*, the model is executable and accessible in the engine on */flow/engine/*.
- The *CPEE-GUI* runs on https://cpee.org/flow/. It provides a user interface to instantiate, manipulate and visualize processes during execution. In order to open a specific instance in the CPEE-GUI, it can be passed within the variable "monitor". The entire link (for e.g. open instance number 1) would look like this:
  https://cpee.org/flow/?monitor=https://cpee.org/flow/engine/1/

The CPEE provides an extensive REST interface that enables the entire communication with the engine and its instantiation service. Therefore, the CPEE-GUI is only a visual overview of process instances, but however is not necessary in its actual enactments.

6.1.3.1 Process Models

After the process models have been sent from the worker to the CPEE, they are instantiated by the CPEE, afterwards extended (again by the worker) with the necessary parameters (data elements, endpoints) and then started. After the process is started, the defined control flow gets enacted and the Inkscape-Manager functionalities (endpoints) are called with appropriate process variables (data elements), executed, and then will return one or more parameters to the process model (updating data elements).

The storage format of CPEE process models is of type XML. The following figure 6.9 and the associated listing 4 show a comparison of a simple process model in XML and its counterpart in the CPEE-GUI.

The data elements represent the variables - in this example x = 5 and y = "abc". They can be passed to services, and/or used in the process (e.g. for decisions). The endpoints represent the request URLs of the functionalities and can be seen in the CPEE-GUI when switching to the tab "Endpoints". As can be seen in the XML under "endpoints" (listing lines 7 to 9), there is an endpoint my_service = http://localhost:1234/my_service. The process flow can be found in the CPEE-GUI under "Graph" and its counterpart in the XML under "description" (listing lines 15 to 24). In this example, there is one service call called "Service Call" with ID "a1" that controls the endpoint "my_service" with a get call.

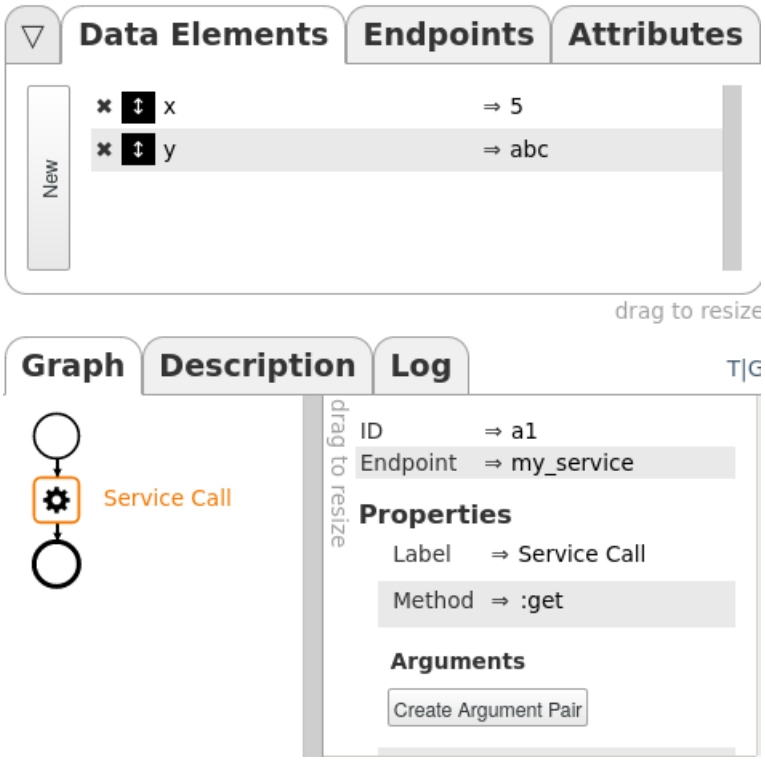


FIGURE 6.9: CPEE process model - GUI

```
1  <testset xmlns="http://cpee.org/ns/properties/2.0">
2    <executionhandler>ruby</executionhandler>
3    <dataelements>
4      <x>5</x>
5      <y>abc</y>
6    </dataelements>
7    <endpoints>
8      <my_service>http://localhost:1234/my_service</my_service>
9    </endpoints>
10   <attributes>
11     <info>process info</info>
12     <modeltype>CPEE</modeltype>
13     <theme>preset</theme>
14   </attributes>
15   <description>
16     <description xmlns="http://cpee.org/ns/description/1.0">
17       <call id="a1" endpoint="my_service">
18         <parameters>
19           <label>Service Call</label>
20           <method>:get</method>
21           <arguments/>
22           ...
```

```
23        </call>
24      </description>
25    </description>
26    <transformation>
27      <description type="copy"/>
28      <dataelements type="none"/>
29      <endpoints type="none"/>
30    </transformation>
31  </testset>
```

Listing 4: CPEE process model - XML

6.1.4 Process Repository

All executable CPEE process models for different users and in different variants are stored in the process repository. If processes are instantiated by the worker, the desired variant is loaded directly from the repository in its current form and sent to the CPEE. Since it is a shared repository (see description of single/shared repository in chapter 4 "Solution Design", section 4.1.3.3), processes are stored in different variants (by different users). The structure of the repository is shown in figure 6.10.

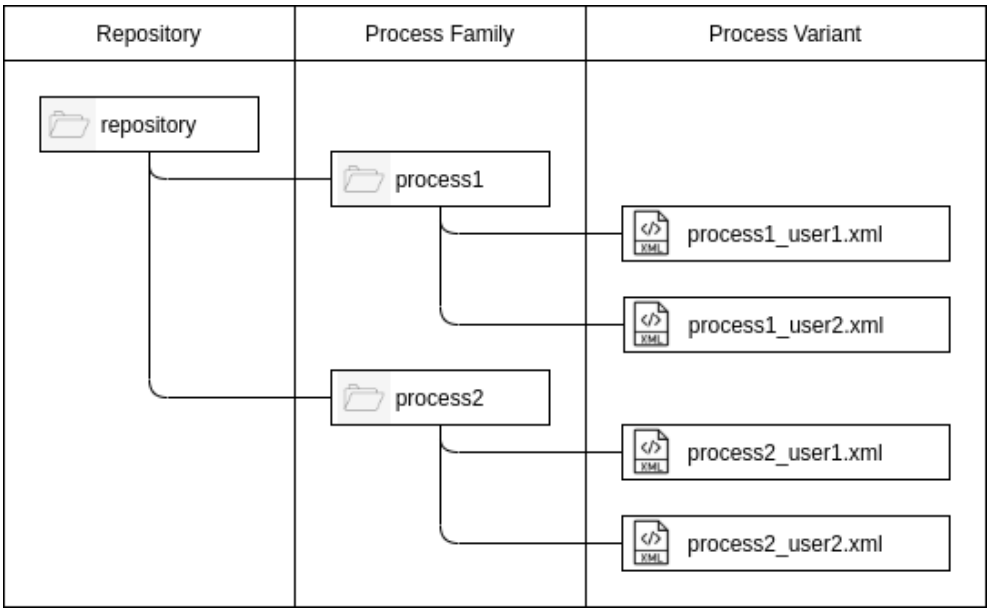


Figure 6.10: Process repository

Since the process models are simple XML documents, they are located in a directory on the file system. The main folder is called "Repository". Below this folder, sub-folders for each individual process family (in this example process families "process1" and "process2") exist. In each of these process family folders, all variants (for different users) are stored in the form of XML documents. The differentiation between these variants take place via the file name in the form {process family}_{user}.xml.

6.1.5 Configuration File

The configuration file is crucial to enable the pattern "Configurable Process Start". It contains the information about which process variant should be instantiated with which parameters. It literally represents the configuration for the process variants. The config file is of type XML, and can be seen in listing 5.

```xml
<?xml version="1.0" encoding="utf-8"?>
<config>
    <user>user1</user>
    <cpee>
        <server>https://cpee.org/flow/engine/</server>
        <instantiation>https://cpee.org/flow/start/</instantiation>
        <repository>/home/pc-user/repository/</repository>
    </cpee>
    <processes>
        <process id="insertObject">
            <dataelements>
                <x>5</x>
                <y>abc</y>
            </dataelements>
            <endpoints>
                <service_1>http://cpee.org/service_1</service_1>
                <service_2>http://cpee.org/service_2</service_2>
            </endpoints>
        </process>
    </processes>
</config>
```

Listing 5: Config file - XML

The configuration file consists of the root element "config", and 3 central categories "user", "cpee" and "processes":

- **User:** The user represents the prefix that is used in order to search for a desired process variant. As can be seen in the section 6.1.4 "Process Repository", the process models are stored under the file name processfamiliy_user.xml. The user is therefore necessary in order to define the correct process variant.
- **CPEE:** The paths to the necessary CPEE services are saved here. Since the CPEE (engine and instantiation, see section 6.1.3 "Process Engine - CPEE") can be deployed on different locations (both different ports and different infrastructures), it is important to specify the correct uniform resource locator (URL). The same applies to the process repository, which can also be located in different places (depending on the client's preferences), which is why the correct location must be specified in the configuration file.
- **Processes:** Data elements and endpoints of all processes of an user are stored under the part "processes". There can be a desired number of processes, which are all registered under the tag <processes>. The attribute "id" for a process indicates the process family and must be unique. Among them there are always the tags <data elements> and <endpoints>, where parameters are specified which should be sent to the CPEE at the start of a process. Data elements and endpoints always use the format <key> value </key>.

In summary, the following information can be extracted of the configuration file in listing 5: If a process family named "insertObject" should be started, the process model "insertObject_user1.xml" will be fetched from the repository located at "/home/pc-user/repository/insertObject/" and will be sent to the CPEE with the data elements *"x" = 5* and *"y" = "abc"* and the endpoints "service_1": "http://cpee.org/service_1" and "service_2": "http://cpee.org/service_2".

6.2 Inkscape-Manager Demonstration

In the previous section 6.1 "Architecture", all components with their technical aspects, as well as process sequences, service calls, and more, have already been explained. The aim of this section is to put these components together in a clear sequence in order to demonstrate a typical process

flow of this sample implementation. Figure 6.11 shows the components and their interrelationships in order to demonstrate the sequence flow. Detailed information on the individual steps can be found in the description of the respective components in section 6.1 "Architecture".

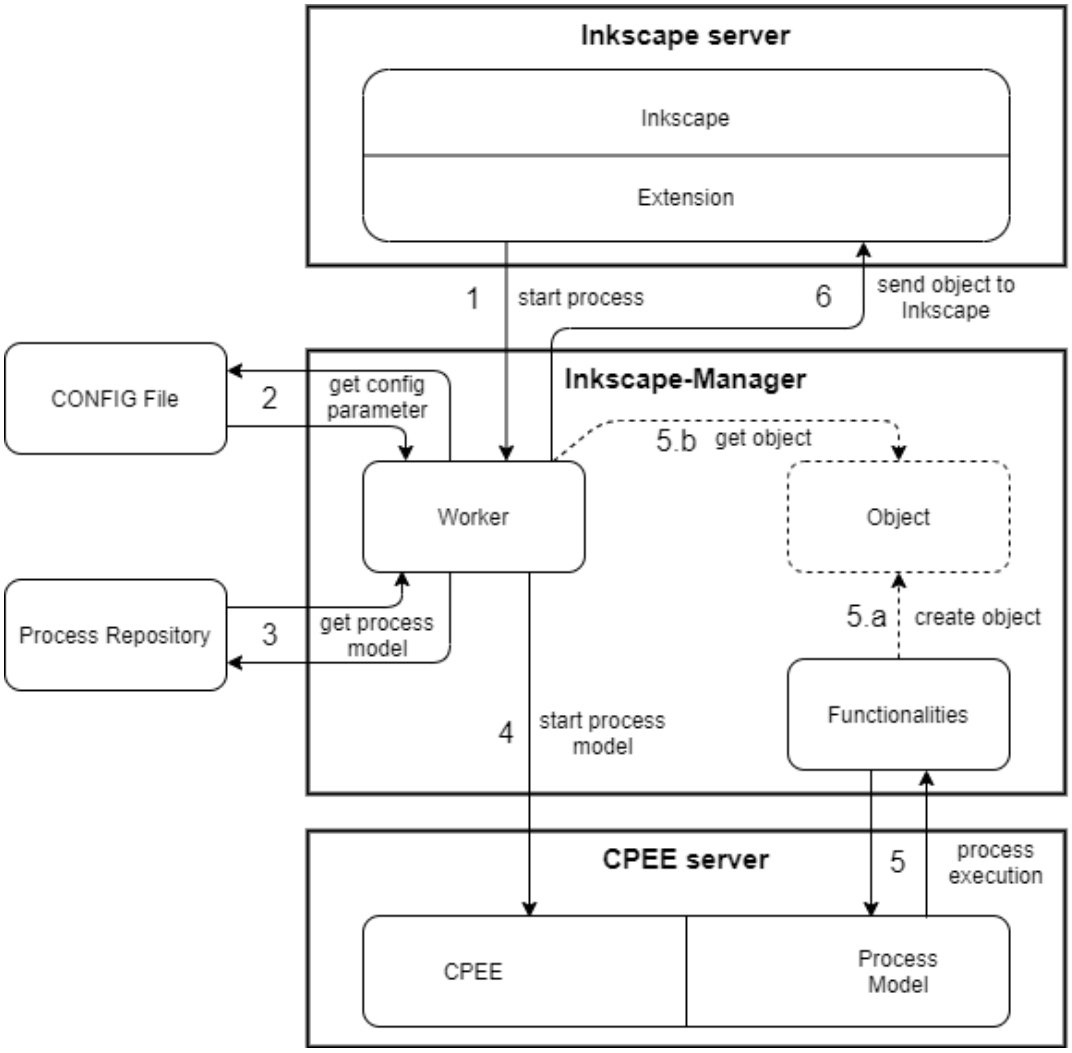


FIGURE 6.11: Typical process flow of Inkscape-Manager

1. *Start process:* The first step is to start a process. For this purpose, an "impulse" is necessary in order to inform the worker about the start of a process. In the case of the Inkscape-Manager, this impulse comes from an user who operates Inkscape. Via Inkscape the user can start an extension, which in turn informs the worker.

2. *Get config parameter:* To initiate the Configurable Process Start, all parameters are read from the config file. The information in it provides details about which process variant should be started with which parameters (data elements and endpoints).

3. *Get process model:* After the information about the variant to be executed has been read from the config file, the worker can fetch the process variant from the repository in the form of its CPEE process model.

4. *Start process model:* The appropriate process model is then sent to the CPEE with the appropriate data elements and endpoints and the process is started.

5. *Process execution:* Now the actual execution of the process begins. The process model is enacted and communicates with the functionalities of the Inkscape-Manager. This also includes the communication with the GUI and thus the communication with the user.

(a) *Create object:* If the process enacts properly and an object should be sent to Inkscape, a functionality creates the object on the file system.

(b) *Get object:* Since the start of the process, the worker constantly checks whether an object has been created and fetches it as soon as it exists.

6. *Send object to Inkscape:* As soon as the object exists and the worker has fetched it, it is returned to Inkscape.

6.3 Comparison to SX

The motivation of the whole thesis is to develop a novel solution approach in order to solve the variability problems of CP's software SX (see chapter 2 "Scenario"). Since the implementation of CP's solution is not part of this thesis, this sample implementation "Inkscape-Manager" is intended to demonstrate the solution approach and draw very clear parallels to the use case of CP. Therefore, this section shows to what extent the conceptual solution approach PBMC with Configurable Process Start could be demonstrated by this sample implementation.

Therefore, two crucial points are addressed in this section:

1. Architecture: To what extent are the architectures of our implementations comparable?
2. Process Input and Output: To what extent are the sample processes used comparable?

6.3.1 Architecture

In this section, the target architecture of CP after application of PBMC is compared with that of the sample implementation "Inkscape-Manager". Figure 6.12 shows the two architectures.

The two cases result in almost exactly the same architecture. The only small difference is in the interface to the external component (plant or Inkscape). This difference results from the fact that SX with its functionalities takes over the entire control of the plant and therefore has direct access to it. Therefore, the process is also started via these functionalities.

In contrast, the functionalities of the Inkscape-Manager do not have direct access to Inkscape, because the control of Inkscape is only possible via the interface of the Inkscape extension. Therefore, the process is started via the Inkscape extension.

However, since main concepts (Worker, CPEE, Configuration File, Process Repository) are exactly the same for both use case, these small differences do not matter in order to demonstrate the concept of the service-oriented architecture with process technology and process variants through Configurable Process Start.

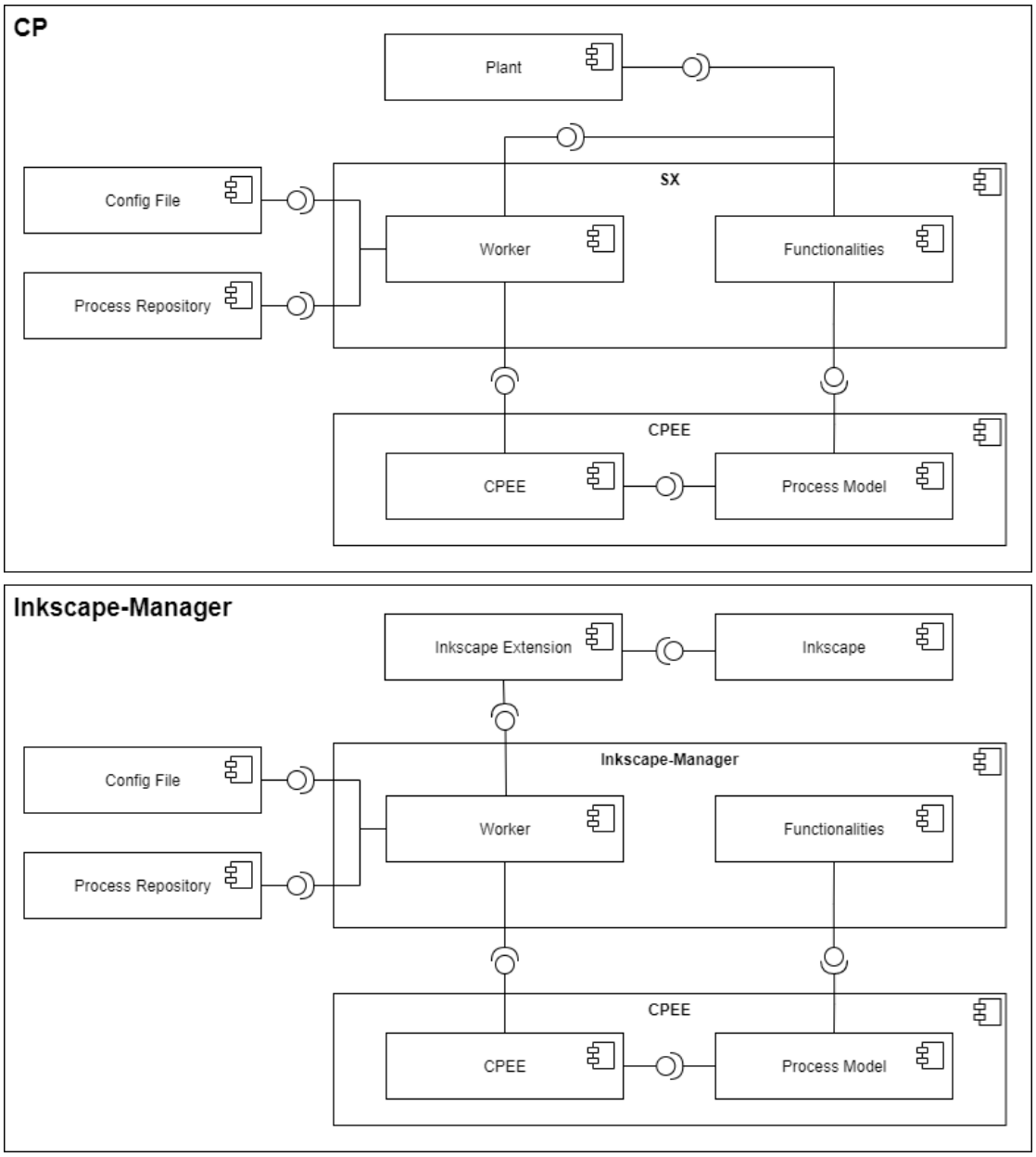


FIGURE 6.12: Comparison: target architecture - CP and Inkscape-Manager

Plant - Inkscape; The actual problem of chapter 2 "Scenario" does not include the plant (intentionally, because the focus should not be on the plant, but on the process variability handled in SX). However, in this sample implementation, it is included for demonstration reasons. In the Inkscape-Manager the plant is demonstrated through Inkscape. It is the component that ultimately receives the transferred object after the process has been enacted. In CP's scenario, the plant receives the grain after the process has been enacted.

SX - Inkscape-Manager; The entire logic of the software is located in SX and the Inkscape-Manager, respectively. Although these two components are developed in different programming languages, they are architecturally similar. Both contain the worker (important component for initialising the Configurable Process Start) and other functionalities (GUI, services, etc.).

CPEE - CPEE; In both scenarios, the CPEE was used as process engine. Therefore, the process engines are exactly the same and there is absolutely no difference between them.

Functionalities - Functionalities; In both architectures it is possible to link services (called functionalities). As already mentioned at the beginning of these section, the functionalities can vary in what they do and which interfaces they have (e.g. functionalities of SX have direct access to the plant, functionalities of the Inkscape-Manager do not have direct access to Inkscape). However, the extent of how these functionalities work has no effect on the concepts taught with this implementation.

6.3.2 Process Input and Output

After showing the similarities in architecture, in this section a comparison of the sample processes is done. As presented in chapter 2 "Scenario", the sample process of CP is an approval process that determines whether incoming grain can be forwarded to production. Figure 6.13 provides an overview on process input and output of the sample processes.

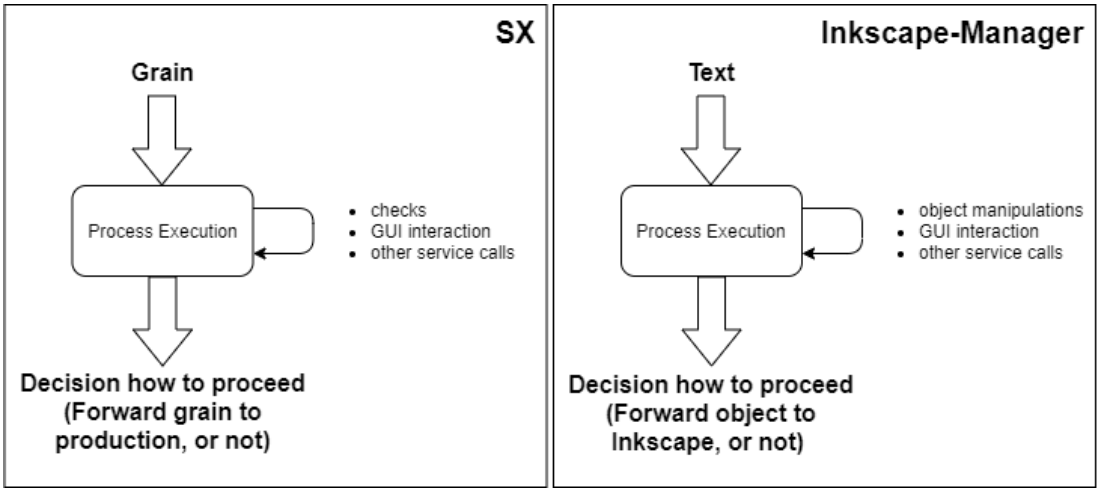


FIGURE 6.13: Comparison - process input and output

In CP's sample process, grain comes into the process as input (either directly at the start of the process or in one of the first tasks). The grain is subjected to various checks within the process through service calls, queries and user interactions. Of course, any other service available can also be used in the process execution. The output of the process is a decision whether the grain can be forwarded to production or not. The initiation of the forwarding could also take place at the end of the process (e.g. updating parameter, starting new processes, etc.).

In comparison, exactly the same process takes place in the sample process of the Inkscape-Manager. The only difference is that SVG-objects are used instead of grain. Text enters the process as input (as in CP's sample process, either directly at the start of the process or in one of the first tasks). Based on this text, an SVG object is created and manipulated within the process through service calls and user interactions. Of course, as in CP's sample process, any other service available can also be used in the process execution. The output of the process is a decision whether the object created can be forwarded to Inkscape or not. For demonstration reasons, this step (if the object should be forwarded) is directly implemented in the sample process of the Inkscape-Manager.

As can be seen, the two sample processes are absolutely similar and have therefore been used for this implementation demonstration.

6.4 Process Variants and Configurable Process Start at Inkscape-Manager

In this section, the process variants and its configurable process start are demonstrated, using the sample implementation. Two different configuration files are presented and the resulting different process flows are pointed out. Process variants defined in different configuration files must also exist in the process repository. Figure 6.14 therefore shows process variants for the users "aztec" and "qr", which both belong to the process family "insertObject" and are existent in the repository.

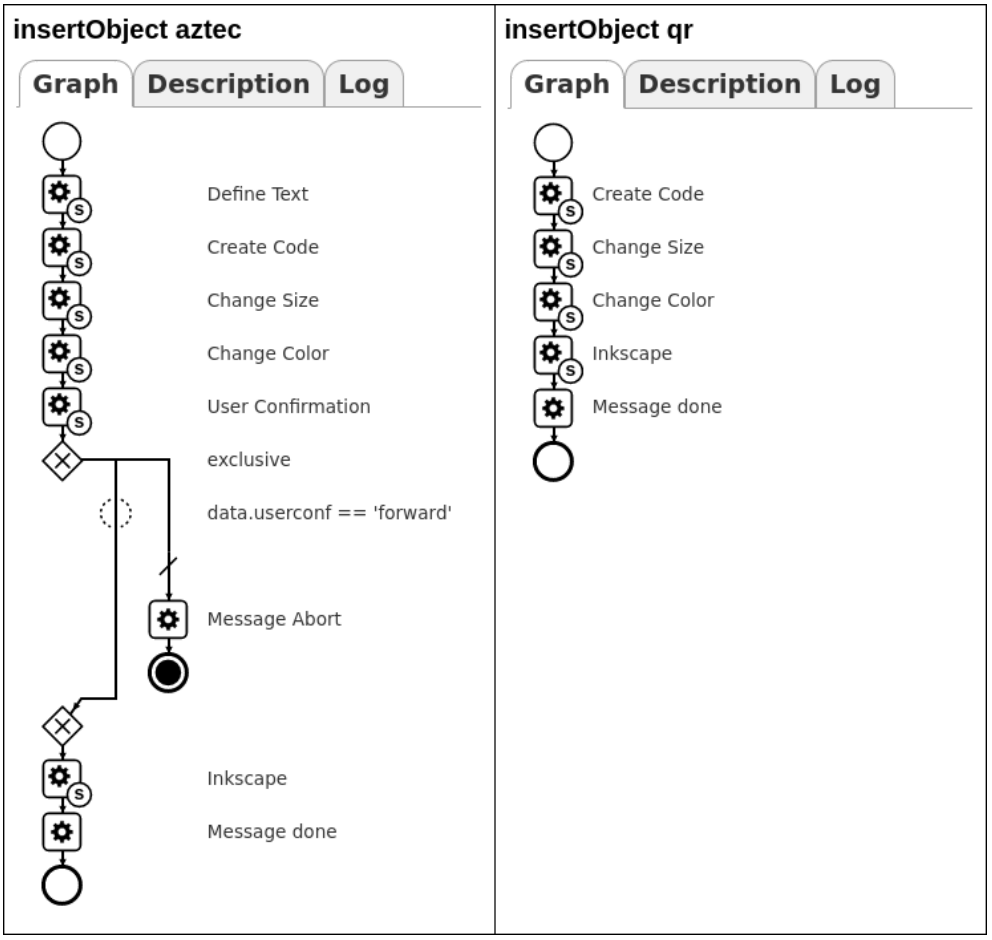


FIGURE 6.14: Comparison of process variants for the users "aztec" and "qr"

Both process variants start by enacting the Inkscape extension "Insert Object". Depending on the current state of the configuration file, either the process variant of user "aztec" or the one of user "qr" is executed. Listing 6 shows the configuration file of "aztec". Therefore, when the Inkscape Extension is started, the process variant of "aztec" is started automatically (no further action is required).

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <config>
3     <user>aztec</user>
4     <cpee>
5         <server>https://cpee.org/flow/engine/</server>
6         <instantiation>https://cpee.org/flow/start/</instantiation>
7         <repository>repository/</repository>
8     </cpee>
9     <processes>
10        <process id="insertObject">
```

```
11          <dataelements>
12              <text>default</text>
13              <size>12</size>
14              <color>#cc3b1b</color>
15          </dataelements>
16          <endpoints>
17              <user_text>http://cpee.org/inkscape-manager/user_text</user_text>
18              <createaztec>http://cpee.org/inkscape-manager/createaztec</
    createaztec>
19              <changesize>http://cpee.org/inkscape-manager/changesize</
    changesize>
20              <changecolor>http://cpee.org/inkscape-manager/changecolor</
    changecolor>
21              <user_confirmation>http://cpee.org/inkscape-manager/
    user_confirmation</user_confirmation>
22              <message>http://cpee.org/inkscape-manager/user_message</message>
23              <inkscape>http://cpee.org/inkscape-manager/inkscape</inkscape>
24          </endpoints>
25        </process>
26    </processes>
27 </config>
```

Listing 6: Config file - variant 1

Since the first service call "Define Text" in the process variant of "aztec" is a text input, the process stops and the user must return an input to continue. After the other service calls have been completed, the service call "User Confirmation" is again an user interaction that requires the user's confirmation. There, the user has the option to end the process prematurely. If the process is not terminated, the object is sent to Inkscape calling the penultimate service "Inkscape". Afterwards, a message is sent to the user, informing that the process is finished. Figure 6.15 shows the entire history of the process variant of user "aztec" and the resulting SVG object, that is inserted in Inkscape.

Figure 6.15: Process variant 1: history and results

If, the configuration file is changed to user "qr", the next time starting the Inkscape extension "Insert Object", process variant "qr" will be executed. The configuration file of user "qr" can be seen in Listing 7. Compared to the configuration file of "aztec" (Listing 6) the tags "user", "dataelements" and "endpoints" are different. If the CPEE would be running on a different infrastructure or the process repository would be stored in a different location, it would be necessary to also change the parameters under the tag "cpee".

```
<?xml version="1.0" encoding="utf-8"?>
<config>
    <user>qr</user>
    <cpee>
        <server>https://cpee.org/flow/engine/</server>
        <instantiation>https://cpee.org/flow/start/</instantiation>
        <repository>repository/</repository>
    </cpee>
    <processes>
        <process id="insertObject">
            <dataelements>
                <text>www.univie.ac.at</text>
                <size>6</size>
                <color>#0063a6</color>
            </dataelements>
            <endpoints>
                <user_text>http://cpee.org/inkscape-manager/user_text</user_text>
                <createqr>http://cpee.org/inkscape-manager/createqr</createqr>
                <changesize>http://cpee.org/inkscape-manager/changesize</
      changesize>
                <changecolor>http://cpee.org/inkscape-manager/changecolor</
      changecolor>
                <inkscape>http://cpee.org/inkscape-manager/inkscape</inkscape>
                <message>http://cpee.org/inkscape-manager/user_message</message>
            </endpoints>
        </process>
    </processes>
</config>
```

Listing 7: Config file - variant 2

If the configuration file of "qr" is activated, the process variant "qr" will be executed on starting the Inkscape Extension "Insert Object". Process variant "qr" does not contain any user interactions, but only uses default values to achieve the result. As soon as the extension is started, the entire process is enacted and the object is inserted to Inkscape without any user interactions. Figure 6.16 shows the whole history of the process variant "qr" and the resulting SVG object. Comparing the process histories of "aztec" and "qr", the detailed differences between the two process variants can be seen.

FIGURE 6.16: Process variant 2: history and results

This sample implementation contains exactly two different process variants of the users "aztec" and "qr". However, there are no limits to the number of different variants that can be created.

Note: The entire implementation is available at the Github repository *tpointer14/inkscape-manager*[1].

[1]https://github.com/tpointer14/inkscape-manager

Chapter 7

Discussion

Highly customized software requires either that (1) different versions of the software are maintained, or that (2) extensive configuration / scripting mechanisms are included in the software. Which possibilities exist to transform / extend existing legacy software to solve the maintenance problems when dealing with software variants?

In order to answer this research question, an extensive literature research was done. Current literature agrees, that software cloning and modifying is cheap in the beginning, but dramatically increases maintenance costs in the long term [16] [39] and is error-prone [40] [41], especially with an increasing number of variants. Therefore, some authors present concepts and tools reducing maintenance difficulties and error-proneness by supporting the software cloning process [39] [40] [41] [42]. However, the systems nevertheless are static and require source code changes in order to adapt to user requirements. As mentioned in the motivation of this thesis (see section 1.1), companies want and need to react flexibly and dynamically to environmental changes in order to retain competitiveness and thus must adapt their systems in shortest possible time. Therefore, Sayagh et al. [17] define, that modern software applications can be adapted to different situations by configuration, without modifying source code. This allows software adaptations on-the-fly, as changes, like e.g. disabling features or algorithms [17], can be made at any time and do not need programmer support. In order to allow configuration, every configuration option must be integrated to the software. Typical storage media for configuration options are [17]

- configuration files (of type "txt", "ini", "xml" or "json"),
- domain-specific languages (DSL) [43],
- relational databases, and
- distributed key-value stores like Apache ZooKeeper[1].

However, since process orientation takes an ever more important role for companies and information technology systems are affected by it, this thesis contributes a process-based configuration mechanism. Variable source code, will be represented by individual process models (for individual software users), which will be executed by a process engine, whereby model-driven configuration becomes possible.

[1] https://zookeeper.apache.org/

Existing concepts regarding model-driven configuration, assume a clear separation between process / business logic, functionalities, and user interaction. Which additional considerations and modeling elements have to be taken into account, when transforming / extending existing legacy software, in order to define a simple / yet comprehensive interface between the legacy software and model-based extensions?

In the course of this thesis a framework *Process-Based Mediation Configuration* (PBMC) is presented, which extends a monolithic software, existing in several variants, with process technology in order to solve the upcoming maintenance problems by model-driven configuration. PBMC consists of four stages:

1. Variation Identification
2. Variation Extraction
3. Augmenting with Process Technology
4. Process Variant Management

1. Variation Identification: Components of the software that contain variability are identified and delimited.

2. Variation Extraction: Components containing variability are extracted from the monolithic software and all interfaces are adapted.

3. Augmenting with Process Technology: A process engine is implemented and the extracted variable components are transformed into executable process models (for each user). After this stage, each user has his individual process variant models.

4. Process Variant Management: Emerging process variants (each process family has an individual variant for each user) are handled.

After applying PBMC, a single service-oriented software architecture whose entire variability is managed by executable process models results. Thus, the same software can be delivered to all users, and individuality is adapted by the model-driven configuration of processes. This simplifies maintenance enormously for the software vendor, since (1) just one single software is maintained instead of many different versions, and (2) processes can be changed via its process model, without adapting and redelivering the entire software. In addition, the architecture, after applying PBMC, brings also advantages for the user of the software, as processes can be adapted dynamically to changing circumstances.

In order to allow a process-based model-driven configuration, a clear separation between process / business logic, and other components like functionalities and user interaction must be given - existing concepts regarding model-driven configuration also assume this separation. Therefore, the first three stages of PBMC ensure this separation by outsourcing any variability to process models and separating it from other functionalities and user interactions. However, since every user can create his individual process for a specific process family, process variants occur. So, the enactment of the first three stages of PBMC represents a transformation process, transforming software variants to process variants. The handling of these emerging process variants is part of stage 4 "Process Variant Management".

Almost all existing approaches for process variant management represent different variants in a single reference model and derive individual variants through configuration mechanisms like extension or / and restriction. Therefore, redundancy can be avoided and changes that affect multiple variants are easier to maintain (changing one model instead of several). However, in order to achieve advantages using this kind of configuration, variants need to (1) have a certain degree of similarity and (2) be regularly changed "together". Since, in the scenario of this thesis, process variants are not changed together, an existing approach brings no maintenance reduction. However, to nevertheless conceptually link variants of a process family, the pattern *Configurable*

Process Start was developed. Van der Aalst et al. [2] defined workflow patterns, which can be used for configuration purposes. However, these configurable workflow patterns always refer to configuration within a process. It is not taken into account, that configuration can also be present at process start.

Therefore, the pattern allows a configurable process entry point for process variants that are stored in different process models. Thus, in order to enact a process, only the process family is called and the actually instantiated variant will be selected dependent on arbitrary parameters (e.g. the respective user). This *Configurable Process Start* was demonstrated within a prototypical implementation. For this purpose, the software Inkscape was extended by an extension inserting an object. Depending on the parameters set in a configuration file, a different variant is enacted despite the same extension "Insert Object" is called. This leads to (1) different process sequences, (2) different user interactions and (3) also totally different objects that finally appear in Inkscape.

So, this thesis contributes a framework *PBMC* allowing to transform a monolithic software to a service-oriented software and extending it with process technology in order to enable process-based model-driven configuration. Moreover, a pattern "Configurable Process Start" is contributed, extending existing configurable workflow pattern with a configurable process entry point.

What are relevant metrics and characteristics, to determine the complexity and feasibility of legacy software transformations / extensions, in comparison to existing approaches?

In order to evaluate PBMC and its Configurable Process Start, a detailed comparison to existing concepts was enacted in the course of this thesis. Therefore, the properties (1) Reference Model, (2) Type of Configuration, (3) Software Design Pattern, (4) Type, (5) Realizability, (6) Application Scenario, (7) Model Representation, (8) Modification of Single Variants and (9) Initial Scenario were analysed and compared using the existing approaches PROVOP [29], C-EPC [2] and REMUS [3].

In comparison to existing approaches regarding model-driven configuration, PBMC is more comprehensive, since it also includes the transformation of software variants to process variants, while the initial scenario of the other approaches already contain process variants. Due to the accompanying architectural change, PBMC includes the software design pattern "Mediator" while the other approaches do not include a software design pattern. Moreover, while existing approaches use a reference model of different extent and configure it by restriction or extension, PBMC has no single reference model, since it represents variants in different models, and thus configures through a configuration file at process start.

According to the analysis regarding existing approaches, they operate only on a conceptual level and are not executable. Only REMUS [3] is executable, but however (in contrast to PBMC) has no real-world application scenario.

Since, each process variant exists in a individual process model, PBMC needs multiple models for representation, however the modification of a single variant is very easy in comparison to other approaches. Moreover, it is simple to implement PBMC (especially its Configurable Process Start), as it does not include a complex configuration mechanism but configures simply and efficiently via the process start.

Therefore this thesis contributes this detailed and independent evaluation, based on relevant metrics and characteristics.

Chapter 8

Conclusio and Future Work

In the course of this thesis, a framework named *Process-Based Mediation Configuration* (PBMC) was developed, to transform / extend monolithic software in order to enable process-based model-driven configuration. During the framework development, existing configurable workflow patterns [2], enabling configuration within a process, were analysed. Additionally, a new pattern *Configurable Process Start* was presented as a part of PBMC, allowing a configurable process entry point. PBMC was

1. successfully implemented in a real-world application scenario (at a company partner) using a sample process,
2. successfully compared with existing approaches of process-based configuration in an independent evaluation, and
3. successfully demonstrated in a prototypical implementation.

Future work may include and in-depth evaluation of advanced implementation stages and the application of PBMC at the company partner. This will include detailed monitoring of many implemented variants, which will therefore allow for measurements of exact improvements, based on suitable parameters (e.g. lead times of software / process changes, transparency for employees, etc.), in order to evaluate strengths and weaknesses after long-term application.

Bibliography

[1] M. Dumas, M. La Rosa, J. Mendling, and H.A. Reijers, *Fundamentals of Business Process Management*. Springer Berlin Heidelberg, 2018.

[2] A. Dreiling, M. Rosemann, W. M. P. van der Aalst, W. Sadiq, and S. Khan, "Model-driven process configuration of enterprise systems," in *Wirtschaftsinformatik 2005* (O. K. Ferstl, E. J. Sinz, S. Eckert, and T. Isselhorst, eds.), (Heidelberg), pp. 687–706, Physica-Verlag HD, 2005.

[3] R. Vigne, "Remus - a restful marketplace for unified services," Master's thesis, University of Vienna, 2011.

[4] P. Kuchana, *Software Architecture Design Patterns in Java*. Auerbach Publications, 2004.

[5] O. Vasilecas, D. Kalibatiene, and D. Lavbič, "Rule- and context-based dynamic business process modelling and simulation," *Journal of Systems and Software*, vol. 122, pp. 1–15, 2016.

[6] T. H. DAVENPORT, "The new industrial engineering information technology and business process redesign," *Sloan Management Review*, pp. 11–27, 1990.

[7] M. Kohlbacher, "The effects of process orientation: A literature review," *Business Process Management Journal*, vol. 16, pp. 135–152, 02 2010.

[8] K. Mccormack, "Business process orientation: Do you have it? placing an emphasis on processes will help organizations move forward," *Quality Progress*, vol. 34, pp. 51–58, 01 2001.

[9] M. Kohlbacher and S. Gruenwald, "Process orientation: conceptualization and measurement," *Business Process Management Journal*, vol. 17, no. 2, pp. 267–283, 2011.

[10] J. Class D., Mundle, "Geschäftsprozessgestaltung im rahmen des cip-prozesses bei bosch," in *Beschleunigung von Geschäftsprozessen*, pp. 211–224, Riekhof, H.-C., Stuttgart (Schäffer-Poeschel), 1997.

[11] S. Hertz, Johansson, J.K., and F. de Jager, "Customer-oriented cost cutting: process management at volvo," in *Supply Chain Management*, vol. 6, pp. 128–142, 2001.

[12] P. Küng and C. Hagen, "The fruits of business process management: an experience report from a swiss bank," in *Business Process Management Journal*, vol. 13, pp. 477–487, 2007.

[13] Y. Gong and M. Janssen, "From policy implementation to business process management: Principles for creating flexibility and agility," *Government Information Quarterly*, vol. 29, pp. S61–S71, 2012. Government Information Networks.

[14] P. Dadam, M. Reichert, and S. Rinderle, "From function-centered to process-centered information systems - challenges and solution approaches," in *Proceedings 17th German Oracle User Conference*, pp. 18–30, DOAG, Deutsche Oracle-Anwendergruppe, Nov. 2004. null ; Conference date: 10-11-2004 Through 11-11-2004.

[15] W. Sun, X. Zhang, C. J. Guo, P. Sun, and H. Su, "Software as a service: Configuration and customization perspectives," in *2008 IEEE Congress on Services Part II (services-2 2008)*, pp. 18–25, 2008.

[16] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *2013 17th European Conference on Software Maintenance and Reengineering*, pp. 25–34, 2013.

[17] M. SAYAGH, N. Kerzazi, B. Adams, and F. Petrillo, "Software configuration engineering in practice interviews, survey, and systematic literature review," *IEEE Transactions on Software Engineering*, vol. 46, no. 6, pp. 646–673, 2020.

[18] P. Koenig, J. Mangler, and S. Rinderle-Ma, "Compliance monitoring on process event streams from multiple sources," in *2019 International Conference on Process Mining (ICPM)*, pp. 113–120, 2019.

[19] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007.

[20] M. Weske, *Business process management architectures*. Springer, 2007.

[21] *The Workflow Management Coalition*, 1996.

[22] OMG, *Business Process Model and Notation (BPMN), Version 2.0*. January 2011.

[23] H. Acker, C. Atkinson, P. Dadam, S. Rinderle-Ma, and M. Reichert, "Aspekte der komponentenorientierten entwicklung adaptiver prozessorientierter unternehmenssoftware.," in *Architekturen, Komponenten, Anwendungen, Proceedings zur 1. Verbundtagung Architekturen, Komponenten, Anwendungen (AKA 2004)*, pp. 7–24, 01 2004.

[24] H. A. Reijers, "Implementing bpm systems: the role of process orientation," *Business Process Management Journal*, vol. 12, pp. 389–409, 2006.

[25] K. McCormack and B. Johnson, "Business process orientation," *Supply Chain Management, and the ECorporation,'IIE Solutions*, 1933.

[26] A. H. M. T. Hofstede and M. Weske, "Business process management: A survey," in *Proceedings of the 1st International Conference on Business Process Management, volume 2678 of LNCS*, pp. 1–12, Springer-Verlag, 2003.

[27] A. Hallerbach, T. Bauer, and M. Reichert, "Configuration and management of process variants," in *Handbook on Business Process Management 1*, pp. 237–255, Springer, 2010.

[28] J. Mendling, H. Reijers, and W. van der Aalst, "Seven process modeling guidelines (7pmg)," *Information and Software Technology*, vol. 52, no. 2, pp. 127–136, 2010.

[29] A. Hallerbach, T. Bauer, and M. Reichert, "Managing process variants in the process life cycle.," vol. 2, pp. 154–161, 06 2008.

[30] A. Hallerbach, T. Bauer, and M. Reichert, "Correct configuration of process variants in provop," Technical Report UIB-2009-03, University of Ulm, Ulm, February 2009.

[31] A. Hallerbach, T. Bauer, and M. Reichert, "Issues in modeling process variants with provop," in *Business Process Management Workshops* (D. Ardagna, M. Mecella, and J. Yang, eds.), (Berlin, Heidelberg), pp. 56–67, Springer Berlin Heidelberg, 2009.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Elements of reusable object-oriented software addison-wesley," *Reading, MA*, vol. 2, pp. 369–378, 1995.

[33] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszweski, and A.P. Barros, "Workflow patterns," in *Distributed and Parallel Databases*, vol. 14, pp. 5–51, 2003.

[34] D. Riehle, H. Zllighoven, and U. Switzerl, "Understanding and using patterns in software development," *Theory and Practice of Object Systems*, vol. 2, 03 2000.

[35] J. Mangler, G. Stuermer, and E. Schikuta, "Cloud process execution engine - evaluation of the core concepts," in *Arxiv preprint arXiv:1003.3330*, 2010.

[36] G. Stürmer, J. Mangler, and E. Schikuta, "Building a modular service oriented workflow engine," in *2009 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 1–4, 2009.

[37] S. Mauricio and E. Aliverti, *jBPM5 Developer Guide*. Red Hat, Inc., 2012.

[38] L.T. Ly, F.M. Maggi, M. Montali, S. Rinderle-Ma, and W.M.P. van der Aalst, "Compliance monitoring in business processes: Functionalities, application, and tool-support," in *Information Systems*, vol. 54, pp. 209–239, 2015.

[39] M. Lillack, S. Stanciulescu, W. Hedman, T. Berger, and A. Wasowski, "Intention-based integration of software variants," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 831–842, 2019.

[40] 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*, pp. 391–400, 2014.

[41] T. Pfofe, T. Thüm, S. Schulze, W. Fenske, and I. Schaefer, "Synchronizing software variants with variantsync," in *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, (New York, NY, USA), p. 329–332, Association for Computing Machinery, 2016.

[42] S. Duszynski, J. Knodel, and M. Becker, "Analyzing the source code of multiple software variants for reuse potential," in *2011 18th Working Conference on Reverse Engineering*, pp. 303–307, 2011.

[43] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "Variability modeling in the real: A perspective from the operating systems domain," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, (New York, NY, USA), p. 73–82, Association for Computing Machinery, 2010.