Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

Process and Product Analysis in Multidisciplinary, Heterogeneous Engineering Environments

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering / Internet Computing

eingereicht von

Michael Petritsch
Matrikelnummer 0126861

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao. Univ.-Prof. Dr. Stefan Biffl
Mitwirkung: Dipl.-Ing. Dietmar Winkler
Dr. Richard Mordinyi

Wien, 03.03.2016

____________________ (Unterschrift Verfasser)

____________________ (Unterschrift Betreuung)

Process and Product Analysis in Multidisciplinary, Heterogeneous Engineering Environments

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering / Internet Computing

by

Michael Petritsch
Registration Number 0126861

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao. Univ.-Prof. Dr. Stefan Biffl
Assistance: Dipl.-Ing. Dietmar Winkler
Dr. Richard Mordinyi

Vienna, 03.03.2016

(Signature of Author)	(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Michael Petritsch
Janis-Joplin-Promenade 6/5/30, 1220, Vienna

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

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

I want to thank my advisor Dr. Stefan Biffl and his assistants Dr. Dietmar Winkler and Dr. Richard Mordinyi for their guidance through this master thesis. Especially Dietmar Winkler for his patience and motivation. My colleagues in the CDL-Flex, the co-students, developers and scientific staff also had a large impact on this thesis, a lot of it built upon previous work we had done during the past 6 years, and of course the contacts at the industry partner and the involved staff at logicals. Thank you all!

I would also like to thank my family and friends for their support, and most importantly my most precious Nadja who was always there for me when I needed it.

Abstract

In heterogeneous engineering environments, such as power plant engineering, different tools and systems are used independently by engineers from various disciplines, like mechanical, electrical or software engineering.

Collaboration between engineers, tools and system domains can be facilitated by designing a system of systems that implements all required use-cases. Current approaches have the disadvantage that they require stakeholders to perform critical human tasks outside of the system and are therefore time consuming, prone to errors and offer room for improvement.

Another limitation of current solutions is that processes store events that do not allow a connection to the associated engineering data. The lack of such a modeled connection means that it is not possible to create queries spanning both process and product data to measure metrics and key performance indicators. As a result, the stakeholders are not able to analyze where problems in their engineering processes occur and which process factors contributed to them.

Therefore, such a system also has to provide monitoring capabilities that deliver ad hoc insights into ongoing and past processes and allow to derive meaningful data based on defined metrics for project progress and collaboration analysis. This is particularly important for project and business process managers who may not only be interested in factual process data, but also in the collaborative behavior of the involved stakeholders across different domains. With such a monitoring system, they obtain more information for decision making, prediction of future trends about development activities, as well as indicators of product quality.

This thesis addresses these problems theoretically and practically and offers a solution that allows improved insights into multidisciplinary engineering environments. The approach of this thesis will be the modeling and execution of collaboration related processes to collect event data from, which can be used to derive characteristics of system behavior. This is achieved by not only storing historical progress of common concept instances or signals (which represent a model of the created product) in the engineering database, but also by linking process event data with this historical progress by adding references.

The implementation has been evaluated by testing it both against fictional test scenarios as well as real-world test data provided by an industry partner in a case study. The results of the evaluation showed that the benefits of the solution are improved collaboration processes, identification of standard system behavior and detection of anomalies, as well as reducing the overall time effort of the involved stakeholders.

Kurzfassung

In heterogenen Engineering-Umgebungen, wie zum Beispiel der Kraftwerkstechnik, werden inkompatible Werkzeuge und Systeme unabhängig voneinander durch verschiedene Ingenieure aus unterschiedlichen Disziplinen eingesetzt.

Die Zusammenarbeit zwischen diesen Werkzeugen und Systemdomänen kann durch die Gestaltung eines Systems-of-Systemen erleichtert werden, das alle erforderlichen Anwendungsfälle implementiert. Aktuelle Ansätze haben den Nachteil, dass Akteure noch nicht alle kritischen Prozessschritte innerhalb des Systems durchführen können und sind daher zeitaufwändig, fehleranfällig und bieten Raum für Verbesserungen.

Eine weitere Einschränkung ist, dass Prozesse Ereignisse speichern, die keine Verbindung zu den zugehörigen Engineering-Daten zulassen. Das Fehlen einer solchen Verbindung bedeutet, dass es nicht möglich ist, Abfragen sowohl über Prozess- als auch Produktdaten zu erstellen, mit denen übergreifende Metriken und Kennzahlen erfasst werden können. Daraus resultierend sind die Beteiligten nicht in der Lage zu analysieren, wo Probleme in ihren Engineering-Prozessen auftreten und welche Prozessfaktoren dazu beitrugen.

Daher hat ein solches System auch Monitoring-Funktionen anzubieten, die Ad-hoc-Einblicke in laufende und vergangene Prozesse gewähren und aussagekräftige Daten anhand definierter Kennzahlen für den Projektfortschritt und die Kollaborationsanalyse abzuleiten. Dies ist vor allem für Projektmanager wichtig, die nicht nur Interesse an reinen Prozessdaten haben, sondern auch an der Kooperation zwischen den beteiligten Akteuren und den verschiedenen Domänen. Mit einem solchen Monitoring-System erhalten sie weitere Informationen für die Entscheidungsfindung, Vorhersage zukünftiger Trends von Entwicklungsaktivitäten, sowie Indikatoren für die Produktqualität.

Diese Diplomarbeit bietet eine Lösung an, die tiefere Einblicke in fachübergreifende Engineering-Umgebungen ermöglicht. Der Ansatz ist die Modellierung und Ausführung kollaborativer Prozesse in denen Event-Daten gesammelt werden, die dazu genutzt werden Eigenschaften des Systemverhaltens abzuleiten. Dies wird einerseits durch die Speicherung des historischer Fortschritts der "common concept"-Instanzen oder Signale (die ein Modell des entwickelten Produkts darstellen) in der Engineering-Datenbank erreicht und andererseits durch die Verknüpfung der Event-Daten mit diesem historischen Fortschritt, indem Referenzen hinzugefügt werden.

Die Umsetzung wird durch Tests evaluiert, die sowohl fiktiven Szenarien beinhalten als auch eine Fallstudie mit realen Testdaten, die von einem Industriepartner bereitgestellt wurden. Darüber hinaus zeigt die Evaluierung, dass die Lösung die kollaborativen Prozesse verbessert, die Identifizierung von Standardverhalten und Erkennung von Anomalien erleichtert, sowie eine Verringerung des Zeitaufwandes der beteiligten Akteure erreicht.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem statement . . . 2
1.2.1 Stakeholders . . . 3
1.2.2 Scenarios . . . 4
1.3 Aim of the work . . . 6
1.4 Thesis Structure . . . 6

2 Related Work **9**
2.1 Engineering Service Bus . . . 9
2.2 Project Observation and Analysis Framework . . . 10
2.2.1 POAF Architecture . . . 10
2.2.2 POAF Application . . . 12
2.3 Business Process Modeling . . . 13
2.4 Process Mining . . . 16
2.4.1 Event log . . . 18
2.4.2 Process model . . . 19
2.4.3 Process mining types . . . 20
2.4.4 ProM . . . 21
2.5 Product Modelling . . . 22

3 Research Issues **23**
3.1 RI-1 Process definition and implementation . . . 23
3.2 RI-2 Bridging between process and product data . . . 23
3.2.1 RI-2.1 Structure of the Process and Product Data Model . . . 24
3.2.2 RI-2.2 Placement of Bridging events . . . 24
3.3 RI-3 Process efficiency and effectiveness . . . 25
3.3.1 RI-3.1 Initial effort and correctness . . . 25
3.3.2 RI-3.2 Formulating Queries . . . 26

4 Research Approach **27**
4.1 Design Science Research Approach . . . 27
4.2 Design Science Research Methodology . . . 28

5 Use Case & Environment **31**
5.1 Use Case - Change Management in multidisciplinary and heterogeneous engineering environments . . . 31
5.1.1 Previous use case versions . . . 32
5.1.2 Notification Handling . . . 35
5.1.3 Metrics and Key Performance Indicators . . . 37
5.2 Environment . . . 38
5.2.1 Open Engineering Service Bus . . . 38
5.2.2 OpenEngSB services used in this work . . . 38

6 Solution Approach **41**
6.1 Architectural Overview . . . 41
6.1.1 Process Focused Components. . . . 41
6.1.2 Product Data Storage Components. . . . 42
6.1.3 Auxiliary Services. . . . 43
6.2 Activiti . . . 43
6.3 Process BPMN definitions . . . 45
6.3.1 Check-In Process . . . 45
6.3.2 Update Status Process . . . 51
6.4 Process implementations . . . 53
6.4.1 XML Process Definitions . . . 53
6.4.2 Notification Services . . . 56
6.5 Process testing . . . 60
6.5.1 Test Model . . . 60
6.5.2 Test Parameters File . . . 61
6.5.3 Test Class and Methods . . . 62
6.6 Process and Product Analysis . . . 64
6.6.1 Process Analysis . . . 64
6.6.2 Product Analysis . . . 65
6.6.3 H2 and SQL functions . . . 66

7 Evaluation **67**
7.1 Process Implementation Evaluation . . . 67
7.1.1 Evaluation of Check-In Process . . . 67
7.1.2 Evaluation of Status Update Process . . . 72
7.2 Case Study . . . 74
7.2.1 Case Study without Status Updates. . . . 74
7.2.2 Case study with Status Updates. . . . 81
7.3 Summary of Key Performance Indicators . . . 89
7.3.1 Product Key Performance Indicators . . . 89
7.3.2 Process Key Performance Indicators . . . 89

8 Discussion **91**
8.1 Discussion of Research Issues . . . 91

8.1.1 RI-1 Process definition and implementation 91
8.1.2 RI-2 Bridging between process and product data 92
8.1.3 RI-3 Process efficiency and effectiveness 93
8.2 Limitations 94
8.2.1 Artifact Limitations. 94
8.2.2 Evaluation Limitations. 96

9 Conclusion and Future Work **99**
9.1 Conclusion 99
9.1.1 Improvements for Stakeholders 100
9.2 Future work 100

Bibliography **103**

A Used Technologies **109**

B Test Environment **111**

List of Figures **112**

List of Tables **114**

List of Acronyms **117**

CHAPTER 1

Introduction

In this first chapter we provide an overview of the topic of this thesis. In the first section we describe the motivation and background behind this thesis, followed by a problem statement describing the challenges in more detail and the aim of the work where we present the expected results of this work. Finally we conclude this chapter with a brief overview of the thesis structure.

1.1 Motivation

Projects in heterogeneous engineering environments, such as power plant engineering, typically involve different tools and systems that are used independently by engineers from various disciplines to complete their collective work. These engineers can be from disciplines such as mechanical, electrical, process or software engineering which each have their own tools providing a different view on and modifying different parts of the engineering product. Since in reality they are all working on the same product (e.g. on a power plant) their work is interdependent and changes made by one engineer may often have effects on the work of other engineers. As a result they have to synchronize their work frequently during the course of a project, so that every engineer is as up-to-date with the work of his colleagues as he needs to be. This synchronization process cannot be done easily because these tools, while usually having basic import/export capabilities for their own discipline's data formats, typically lack interoperability with tools from different disciplines[Fay et al., 2013]. Therefore it requires considerable manual efforts for synchronization on parts of the involved engineers which is slow and increases the probability of errors.

Additionally, each engineering project consists of different phases or states (e.g. from "initial" to "commissioning complete") which are defined by the project managers. During the course of the project the components of the product (e.g. generators, turbines, etc.) traverse through these phases from the first (at the beginning of planning) to the last (when they are ready to be used). As the work on the various components (or even parts of components) progresses in different speeds, some components of the product may be in different phases than

others. Whenever a component enters a new phase the engineers have to manually apply the states to their components so that the real world and model states are aligned. These states have an effect on the above-mentioned synchronization process: depending on the state of a component, other stakeholders have to be notified about changes on this component; not only to synchronize their work, but also to decide whether or not to accept these changes. Stakeholders include (again) engineers, but also the customer and the manufacturer. For example if the component (e.g. a generator) has already been constructed and installed at the power plant site, then a change would not be made easily and must be approved by another stakeholder.

All these synchronization, notification and approval actions typically are communicated via e-mail, because there is no software or system that supports these actions, resulting in hundreds to thousands of e-mails per project. Project managers have the difficult task to keep track and determine the project progress based on these emails, component part lists containing the product data and verbal communication with the engineers in meetings. They have to derive various process and product metrics based on this information and report the progress and state of the project to their superiors. This is such an overwhelming task that they cannot keep track of all metrics that would actually be beneficial to them and only stick to the most essential metrics. Even these metrics cannot be derived easily and require a considerable amount of manual efforts.

1.2 Problem statement

In response to this scenario, the concept of the Engineering Service Bus (EngSB) was introduced in [Biffl and Schatten, 2009], an adaption of the commonly known Enterprise Service Bus (ESB)[Chappell, 2004] specifically designed for the needs of engineering environments. This concept was subsequently implemented as Open Engineering Service Bus (OpenEngSB[1]), an open source, OSGi-based[2] middleware platform which provides the basic infrastructure to build applications, tailored to specific heterogeneous engineering environments.

In the OpenEngSB, typical use cases for tool collaboration involve the orchestration of multiple services, glued together with business logic that sometimes requires user interaction. In ongoing research at the Christian Doppler Laboratory for ”Software Engineering Integration for Flexible Automation Systems”[3] (CDL-Flex) various problems of tool data transformation [Pircher, 2013], tool data integration [Waltersdorfer et al., 2010] and the synchronization [Moser et al., 2011], [Mordinyi et al., 2015] have been tackled, and first prototypical implementations have been made, as well as engineering workflows proposed which model the synchronization process [Winkler et al., 2011], [Winkler et al., 2014], but some industry requirements have not yet been addressed and a fully workflow-driven and systematic implementation of the scenario has not been successfully implemented yet. On top of this, process and product data should be collected and made available to conduct various queries that can be used to measure process and product metrics which could be useful to stakeholders, especially to project managers, by providing deeper insights into the actual engineering process. The underlying scenarios should be fully modeled using modern business process modeling languages like the Busi-

[1]OpenEngSB: http://www.openengsb.org

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

[3]CDL-Flex: http://cdl.ifs.tuwien.ac.at/

ness Process Model and Notation (BPMN[4]) or the WS-Business Process Execution Language (BPEL[5]) as has already been suggested in the original EngSB paper [Biffl and Schatten, 2009].

Following up in the next subsections we will give an overview of the stakeholders and the scenarios that they have to cope with, based on the input of an industry partner in the area of power plant engineering:

1.2.1 Stakeholders

Each project typically involves four types of stakeholders: *engineers* from different disciplines who create the product, *project managers* who supervise the project and have to report to their superior managers, *customers* who also have to be informed about some product changes and who can also request changes which affect the work of the engineers and finally *manufacturers* who manufacture the parts of the plant.

Engineers

The first type of project stakeholders are the engineers. Usually they work in different disciplines like system, process, electrical, mechanical and software engineering and have to use different specialized software tools to create their products like Siemens' OPM (Output Point Module) for process engineers or Eplan for electrical engineers in this use case. While they all work on the same products, everyone has a different view angle as presented by their tools. For example, while a process engineer plans the links between software and hardware elements of the plant, he is not concerned with the actual connections of the electrical pins of the hardware signals, which are the main concern of the electrical engineer. At the same time the electrical engineer is not interested in software signals unlike the process engineer. These circumstances are reflected by the views in the tools of the engineering disciplines: the mechanical engineer's tool does not display concrete electrical pins and the electrical engineer's tool does not display software signals. But these restrictions are not limited to the views alone; typically the whole data model of the tools exclusively contains information which is required by its discipline. Despite these restricted views there is still a considerable amount of common product data [Waltersdorfer et al., 2010] and changes in one discipline can have effects on the work of other disciplines. As a result, whenever an engineer A, using tool X, makes changes to the artifact that affects the view of an engineer B, using tool Y, B has to synchronize with the changes from A to keep their work consistent. Since the used tools are each from different vendors who usually do not consider interoperability with other vendors (except for some basic csv export capabilities) and general standardization efforts have not yet matured in this particular area, this cannot be done automatically and requires a manual synchronization effort by the engineers, which is mainly done via e-mail and by comparing Excel[6] style sheets containing the artifact data manually or with the help of a plethora of self crafted scripts. This manual synchronization procedure is prone to errors and time consuming, especially if not done regularly it increases the risk of errors when high amounts of changes accumulate over time [Sunindyo et al., 2013].

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

[5]BPEL: http://www.oasis-open.org/committees/wsbpel/

[6]Excel: https://products.office.com/en/excel

Customers

The second type of stakeholders are customers. Customers are not involved as actively as engineers in the engineering process but sometimes need to be informed about changes and legitimize them, especially in the later project phases. Occasionally, they also need to propose changes in the form of an Excel style sheet. They do not directly merge these changes into the system though; instead they send the list to an engineer who has to synchronize existing product data for them.

Project Managers

The third type of stakeholders are project managers. Their responsibility is the general project planning, definition of project phases, project monitoring and controlling, as well as coordination among the projects partners. They are not involved directly in the signal change management process as in changing signals or approving changes, but want to have insights into the ongoing project activities and the overall project progress. Of course this is difficult to almost impossible with justifiable effort, when the activities and progress do not occur in a single information system which offers reporting features, and instead has to be done via e-mails and manual offline synchronization.

Manufacturers

The fourth and last type of stakeholders are the manufacturers. They are responsible for the manufacturing of the product planned by the engineers. They do not provide any changes themselves but need to be notified about changes after the manufacturing process has already started and approve or decline the changes they are notified about.

1.2.2 Scenarios

During the course of a project, stakeholders are involved in or have to complete four different scenarios: *product data synchronization* to keep work among engineers synchronized, *status updates* to apply states/phases to the product model, *notification and decision* to involve other project stakeholders into the synchronization process when necessary, and *process and product reporting* which provides reports about process and product metrics based on the data collected during the previous scenarios.

Data Synchronization Scenario

Currently the synchronization is not a simple task because these tools typically lack interoperability [Fay et al., 2013] and results in the following steps, as shown in figure 1: (1) the product data from one tool has to be extracted, (2) sent to all engineers who then have to (3) transform the data to match their view, (4) check the data for changes and (5) import these changes into their own tools. The data extraction is usually done with an export function of the tool to a file in a data exchange format like Comma-Separated values (CSV) [Shafranovich, 2005] where each

line contains a so-called signal, which is the smallest unique part of the product, like the electrical I/O pins of the hardware but may also refer to larger components further up in the hierarchy. This signal file is then sent to other engineers in the project via e-mail who then have to transform the product data before being able to compare it with their own product data and identify the changes. This signal transformation and comparison is very tedious and prone to errors as it can include hundred thousands of signals and has to be either done by hand or (typically) with scripts (e.g. Excel scripts) written by each engineer for his tool or more complex programs (e.g. Access[7] database and user interface) also written by an engineer. These scripts and programs have to be maintained and adapted for each project. Finally, the engineers can re-import the data into their tools via import functions of the tools and continue their actual engineering work.

Status Update Scenario

Similar to the synchronization scenario is the status update scenario, with the difference that the signal's status is more of a process information than a product information. Nevertheless, this information is so important and so closely related to the product or parts of the product that it has to be available most of the time when engineers are dealing with product information. Depending on the state of a signal, additional stakeholders may have to be notified when changes are made to it (see also next scenario). When a signal should transitions into another state, the engineers have to apply the new state to the product data. This has to be done either again in Excel lists or in an Access DB and made available to other engineers.

Notification and Decision Scenario

When a change to a signal is detected in the data synchronization scenario, it may be required to notify not only engineers about these changes but also manufacturers and customers to obtain their permission to apply the changes. Whether or not a notification is required depends on the state of the signal and the fields changed. Notifications and decisions are communicated via e-mail: e.g. the engineer sends the customer an Excel list, the customer checks the list and gives his approval or denies the changes. When multiple stakeholders have to be notified about the exact same changes this may lead to contradicting decisions, when a change is approved by one stakeholder but declined by another. In practice this notification may take several days and further changes may already have been made when the decisions return.

Process and Product Reporting Scenario

The above mentioned scenarios contain process and product information that can be used to get a better overview over the project processes and improve decision processes of project managers. This is difficult to achieve, because process activities are performed in different systems and tools and are communicated by e-mail. If all or most of the processes could be performed in a single system, it would be possible to collect process and product data and use this data to create insightful reports about various process and product metrics.

[7] Access: https://products.office.com/en/access

1.3 Aim of the work

The goal of this thesis is to define and implement prototypes of engineering processes involving multiple stakeholders from different disciplines and also provide means to store process and product data for analysis and to provide insight views on the engineering environments. The processes are based on the input from industry and ongoing research and will be implemented using state-of-the-art technology which has to be integrated in an already existing EngSB environment. The main expected results are as follows:

- Process descriptions for the proposed solution
- The extension and implementation of the described processes
- The definition of process and product metrics
- Collection mechanisms for process and product data measurement
- Evaluation results of a real world use case

1.4 Thesis Structure

The remainder of the thesis is structured as follows:

- Chapter 2 describes the related work of this thesis in which similar problems were handled and solved and on which this work is based upon.
- Chapter 3 describes the research issues, which should be tackled by this thesis.
- Chapter 4 describes the research approach and methodology we have followed during the creation of this work.
- Chapter 5 describes the use case and requirements as provided by the industry partners and previous research and also describes the preexisting environment in which the prototype is implemented.
- Chapter 6 describes the implemented process prototypes that assist engineers in synchronizing their work and and other associated artifacts created during the course of this work which are required for testing and evaluation of the prototype.
- Chapter 7 describes the evaluation of the implemented prototype split into two parts. In the first part, the correctness of the implemented process prototypes is demonstrated by small yet comprehensive test scenarios and in the second part data from an industry partner is used to conduct a case study to demonstrate and test the analysis capabilities of the implementation.
- Chapter 8 contains the discussion of this thesis in which we reflect upon the research issues introduced in chapter 3 and explain how we solved them. Additionally, we list limitations both in the implemented prototype and the evaluation.

- Chapter 9 contains the conclusion of this thesis which summarizes the results and provide an outlook on future work.
- Appendix A provides a list of all technologies and libraries used during the creation of the research prototype.
- Appendix B provides the system specs of the test environment.

CHAPTER 2

Related Work

This chapter discusses related work that serves as the knowledge base and provides the research context of this thesis. First we will introduce the concept of the Engineering Service Bus (EngSB) which supports engineering tool integration, followed by the Project Observation and Analysis Framework (POAF) on which we base our solution approach. Next, we will introduce Business Process Modeling and the basics of the Business Process Model and Notation (BPMN) which will be used extensively in this work. This is followed by Process Mining which supports our process analysis. The chapter is concluded with an overview of previous product modeling research results in the context of the EngSB.

2.1 Engineering Service Bus

The concept of the Engineering Service Bus (EngSB) was introduced in [Biffl and Schatten, 2009], [Biffl et al., 2009] and is an adaption of the commonly known Enterprise Service Bus (ESB) [Chappell, 2004] with the difference that it is specifically designed for the needs of heterogeneous engineering environments where engineers from different disciplines work together with different views and typically non-interoperable tools to create engineering products such as power plants and factories.

In the EngSB engineering tools from various disciplines are integrated by so called *tool domains* which contain the interfaces and a common description (e.g. data models) required for the integration of all tools of a type. *Tool connectors* implement these tool domains and represent the bridge between a concrete tool instance and the EngSB. As a result there can be multiple connectors for a single tool domain. Figure 2.1 gives a high-level overview of this concept, where the colored boxes represent tools from a domain (each color corresponds to a domain type) which are connected to the connectors (the boxes marked with ”c”) which bridge to the EngSB.

As can be seen the intention of the domain-connector concept is not only to support and integrate engineering tools, but all kind of tools, like project management and communication

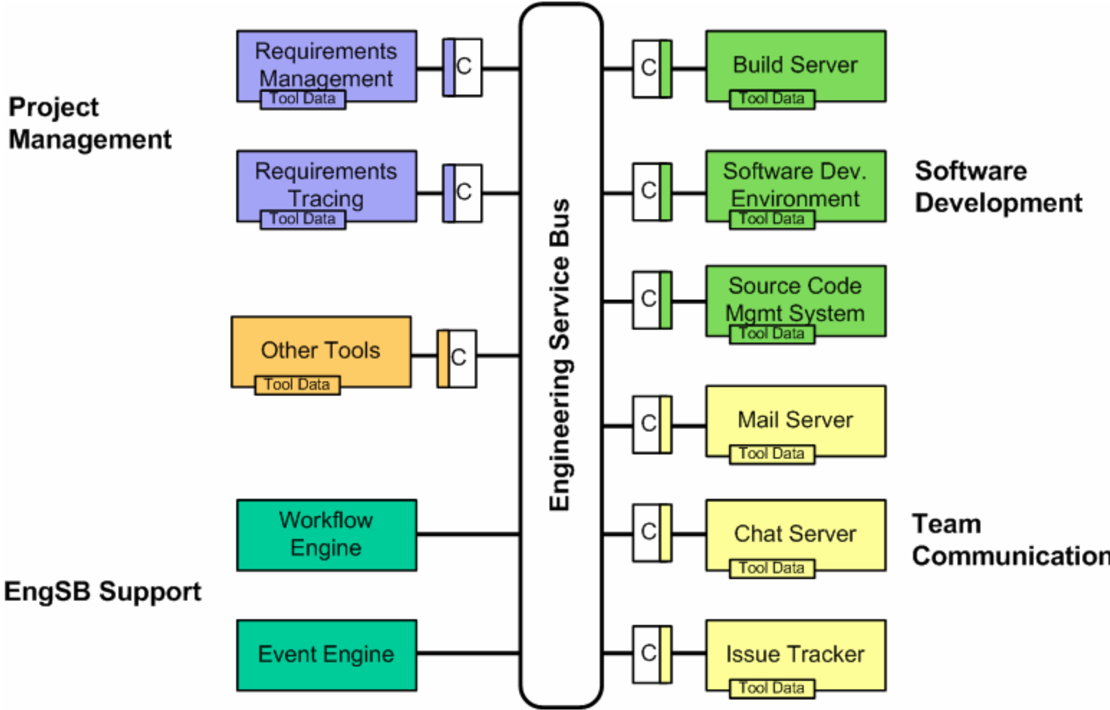

Figure 2.1: EngSB Domain and Connector Concept plus Workflow Engine Service. [Biffl and Schatten, 2009]

tools. Additionally the EngSB can support additional services (represented by the workflow and event engine in the "EngSB Support" category) running directly on the EngSB which do not require domains and connectors to communicate.

The original paper also included plans for a workflow engine to support long-running process, however, up to now no workflow engine has been implemented successfully yet. This workflow engine should be based on the Business Process Model Notation (BPMN) (which we are going to introduce later), as it can sufficiently describe processes in the EngSB context.

2.2 Project Observation and Analysis Framework

In his PhD thesis [Sunindyo, 2012] Sunindyo proposed the Project Observation and Analysis Framework (POAF). It is specifically designed to address the needs of project stakeholders in heterogenous software and systems development environments as described by [Moser et al., 2011], [Sunindyo et al., 2013].

2.2.1 POAF Architecture

In the POAF, seen in figure 2.2 and starting from the left, engineers can input data from different tools via the tool domains and connectors. After an initial preparation which consists of steps

that have to be done before step one (e.g. configuration of data transformation components [Pircher, 2013]) This data is then (1) collected & integrated, (2) followed by an analysis and (3) finally presented visually to project managers and other project participants.

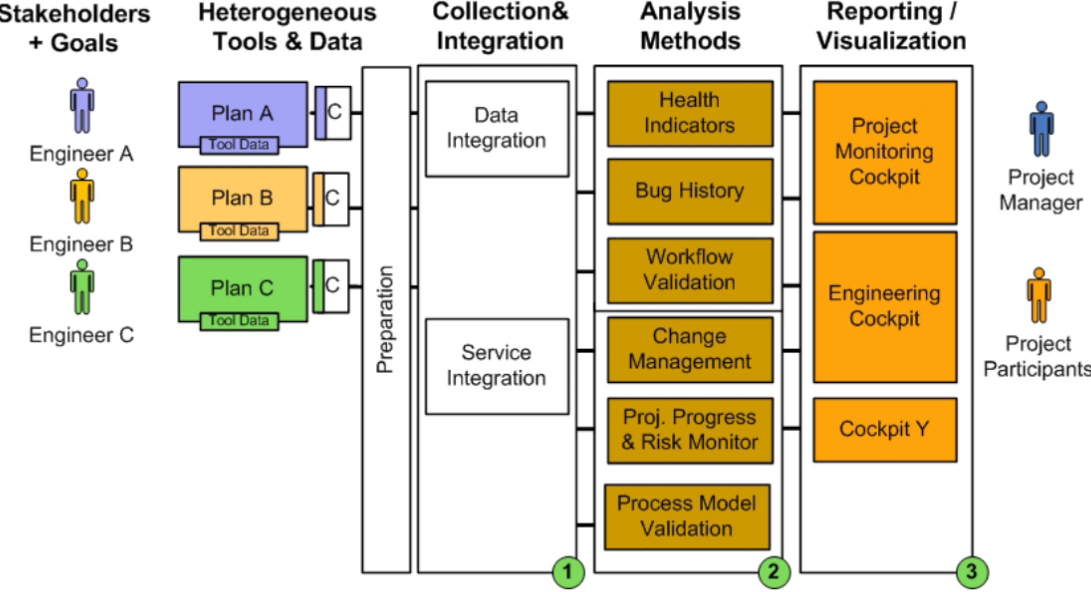

Figure 2.2: Project Observation and Analysis Framework by Sunindyo [Sunindyo, 2012]

Preparation Process

Before the application of any other steps of the POAF, [Sunindyo, 2012] mentions a preparation process that needs to be executed. This process consists of several steps: (1) definition of a process model to identify the process running in the engineering environment, (2) transformation of this process model to a formal notation, e.g. the Business Process Model and Notation (BPMN), (3) an evaluation of the formal process model, (4) a transformation of the formal process model to a rule engine as an event rule, (5) an evaluation of the event rule and (6) the actual generation of the event rule.

Collection & Integration

The first step is the collection and integration of engineering process data and consists of two parts: the data integration and the service integration. [Sunindyo, 2012] suggests that data integration mechanisms should be like the Engineering Knowledge Base (EKB) [Moser, 2009], [Waltersdorfer et al., 2010] or ontology-based approaches [Doan et al., 2004] which both integrate heterogenous data semantically. For the service integration part he suggests the Enterprise Service Bus [Chappell, 2004], [Yin et al., 2009] or the Engineering Service Bus (EngSB) [Biffl and Schatten, 2009], [Biffl et al., 2009] which provide the environments for the data mechanism services to run in and which we have already introduced in the previous section.

Analysis Methods

The second step is the application of various analysis methods onto the collected and integrated engineering process data. [Sunindyo, 2012] mentions six analysis methods: health indicators [Schatten et al., 2006], [Wahyudin et al., 2007], bug history [Sunindyo et al., 2012], workflow validation [Sunindyo et al., 2011a], change management [Winkler et al., 2011], project progress and risk monitoring [Sunindyo et al., 2013] and process model validation [Sunindyo et al., 2011b]. Based on the context of the domain, goals of the stakeholders and the requirements of the stakeholders, different analysis methods should be selected.

Reporting / Visualization

The third and final step of the POAF is the visual representation and reporting of analysis results. [Sunindyo, 2012] mentioned two approaches that deal with this aspect: the project monitoring cockpit [Biffl et al., 2010] and the engineering cockpit [Moser et al., 2011], as well as Decision Support Systems and Executive Information Systems.

2.2.2 POAF Application

[Sunindyo, 2012] applied the POAF to two domains in his work: to an open source software project and to an automation systems engineering environment similar to the domain we are dealing with in this work. Interesting for this work in particular is the second application to the automation systems engineering environment. Figure 2.3 shows the implementation of POAF in this domain with the following steps:

1. an informal process description in the form of workflows described by the customer
2. transformed to a formal model description by the workflow designer
3. which is then transformed and implemented on a rule or process engine
4. executing the process on the engine produces workflow results
5. these worklfow results are converted to an event log
6. the event log is used for process analysis
7. which includes the evaluation of process and product metrics
8. additionally a conformance check can be applied and the results are displayed to a project manager

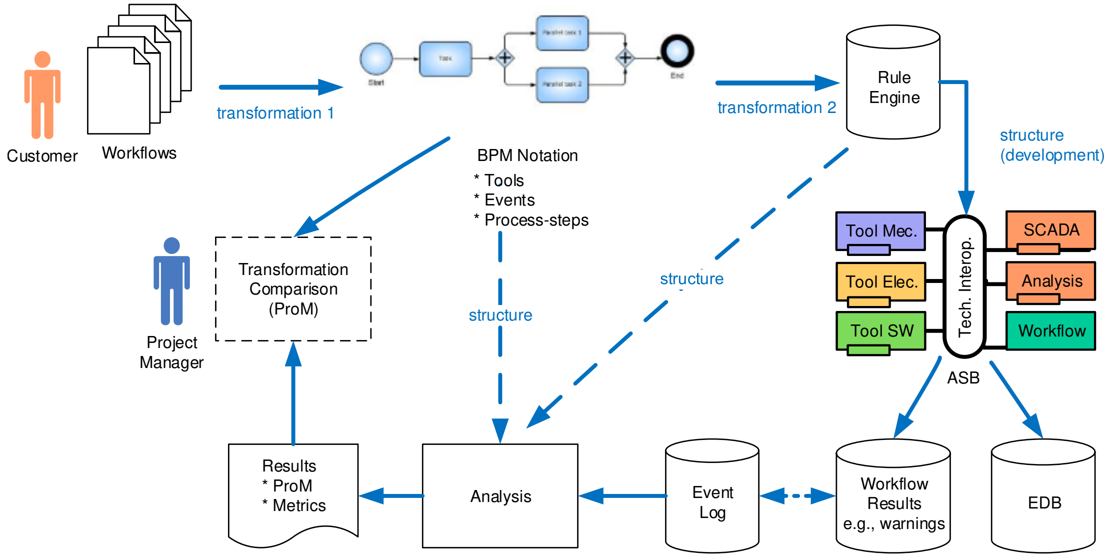

Figure 2.3: Workflow Observation Process [Sunindyo, 2012]

2.3 Business Process Modeling

The Engineering Service Bus [Biffl and Schatten, 2009] and the Project Observation and Analysis Framework [Sunindyo, 2012] both mention business process modelling [Havey, 2005] as viable means to model engineering processes. Most notable candidates are the Business Process Model and Notation (BPMN) [OMG, 2011] and the WS-Business Process Execution Language (BPEL) [OASIS, 2007] with the WS-BPEL extension for People [Kloppmann et al., 2007] to model human interactions.

These business processing notations and languages have seen extensive research in the past [Wohed et al., 2006], [Mulyar, 2005], [van der Aalst et al., 2005] among others, from control-flow, data and resource workflow patterns [Databases et al., 2003] to their support by vendor-specific implementations[1] and their general usage [Recker, 2010].

In this work we are going to introduce only BPMN because previous work has also been based on BPMN and because it also has recently become a research topic in process mining [Kalenkova et al., 2014] [Kalenkova et al., 2015], which we are going to introduce later.

Business Process Modeling with BPMN

The Business Process Model and Notation (BPMN) was developed by the Object Management Group (OMG) to provide a notation that is understandable by all business users. [OMG, 2011] These business users include business analysts who create initial drafts of processes, to developers who implement these processes on process engines, to business people who manage and monitor these processes. The first draft of BPMN was released by the Business Process

[1] Workflow Patterns: http://www.workflowpatterns.com/

Management Initiative (BPMI) in 2004 as BPMN 1.0. In 2005 the BPMI merged with OMG which is now officially responsible for further development. The latest version of BPMN is 2.0.2 [OMG, 2013] which was released in 2013. BPMN has seen increasing popularity during the last years, up to the point where it became a de-facto standard in business process modelling and recently in 2013 was approved as an ISO-standard by the International Organization for Standardization (ISO). The BPMN standard describes a human readable, visual representation of workflow elements and their connections for the modeling of business processes, as well as a machine readable (and to some extend also human readable) representation of the workflow elements in XML. Following we provide a list of the visual representations of the most important BPMN 2.0 elements used in this and in related work, grouped into events, activities, flows, gateways and other elements. For a full list as well as the corresponding XML representations we refer to the BPMN 2.0 specification by OMG[OMG, 2011].

Events are occurrences that can happen during the execution of a process. Table 2.1 shows a list of the events used in this work. Events have a cause (trigger) or an impact (result) and have an effect on the flow. Events can have different internal markers which indicate causes. For example when an event is triggered by an incoming message it contains an envelope symbol. Table 2.1 lists the most important type of events encountered in this work.

Element	Description	Notation
Event	A generic event. It is symbolized by a circle that can contain additional symbols according to the type of the event.	
Start	Indicates the start point of a process. A process typically has one start event, but may also have multiple start events.	
Intermediate	Intermediate events are encountered anywhere between start and end-events and affect the flow of the process without starting or terminating it.	
End	Indicates the end point of a process. A process may have multiple end events.	

Table 2.1: BPMN events [OMG, 2011]

Activities are work that a company performs during a process. Activities can have different levels of granularity, depending on the needed abstraction level in the workflow. Table 2.2 lists the most important type of activities encountered in this work.

Element	Description	Notation
Activity	A generic activity. It is symbolized by a square that can contain additional symbols according to the type of activity.	
Task	A simple tasks. It is an atomic activity in a process. It can be executed by different actors, e.g. the system or a user, depending on the symbol inside.	Task Name
Sub-Process	A subprocess is a non-atomic activity in a process. It is an abstraction for a whole process that is contained in a process.	Sub-Process Name
Sequential	An activity that is executed multiple times sequentially. Can be applied to tasks and sub-processes.	
Parallel	An activity that is executed multiple times in parallel. Can be applied to tasks and sub-processes.	|||

Table 2.2: BPMN activities [OMG, 2011]

Flows are connections between two elements and typically provide the direction of the control and data flow between these elements. Table 2.3 lists the most important type of events encountered in this work.

Gateways are used for splits and joins of the sequence flow in a process. Internal markers indicate the type of behavior control. Table 2.4 shows a list of the events used in this work.

Other Elements do not fit any of the above categories and are listed in table 2.5.

Element	Description	Notation
Sequence Flow	Connects two elements and indicates the order of the elements in a process and a choreography.	
Message Flow	Connects two elements and indicates the flow of messages between two participants. Each pool represents a participant.	
Association	Links information and artifacts with elements. Either directed or undirected.	

Table 2.3: BPMN flows [OMG, 2011]

Element	Description	Notation
Gateway	A generic gateway. It is symbolized by a diamond that can contain additional symbols according to the type of gateway.	
Exclusive	Mutually exclusive selects or joins a single path. (XOR)	X
Inclusive	Selects one or more paths based on conditions. (OR)	O
Parallel	Splits the process up into all following paths. If used as join gateway, waits for all previous paths to finish.	+

Table 2.4: BPMN Gateways [OMG, 2011]

2.4 Process Mining

Due to the steadily increasing size of information systems and successively due to large amount of event logs they produce, a need arose for methods which allow to efficiently harvest these event logs for data that can deliver deeper insights into the systems. Because these computer systems also become more and more closely intertwined with real business processes it is possible to derive more and more useful and accurate information about the business itself.
This is where Process mining came into play: process mining can be used to discover and extract process models from event logs, check the conformance of existing process models with event logs and enhance existing process models with extracted knowledge from event logs, among many other things. Process mining is a relatively young research field, started in the late 1990ies, that sits between computational intelligence and data mining on one side and process modelling

Element	Description	Notation
Pool	A pool represents a participant in a process. E.g. an organization or department. A pool can contain multiple lanes.	Name
Lane	A lane is a subpartition within a pool used to organize and categorize parts of the workflow like activities and events.	Name Name Name
Data Objects	A data object can be used to represent the data flow between workflow elements. It can represent a single object or a collection of objects.	

Table 2.5: BPMN other elements [OMG, 2011]

and analysis on the other side [van der Aalst, 2011].

Figure 2.4 gives an overview of the full process mining framework to put these techniques in the correct context. The real world contains people and organizations using business processes. These processes are supported by information systems. *Provenance* is the systematic collection of process information from these information systems into event logs, which are used for process mining techniques. These event logs can be separated into *pre mortem* data (running processes) and *post mortem* data (completed processes). Navigation, Auditing and Cartography are the main categories of process minging which contain concrete process mining techniques. For a detailed description of all parts we refer to [van der Aalst, 2011].

In 2012 an IEEE Task force on process mining was created under the lead of Wil van der Aalst that gives another excellent and more detailed (yet still compact) overview of process mining in the form of a Process Mining Manifesto [van der Aalst et al., 2011] than we can present here. Nevertheless, we are going to briefly introduce the most basic terms and techniques of process mining in the following subsections, which are:

1. Event logs
2. Process model
3. Process discovery

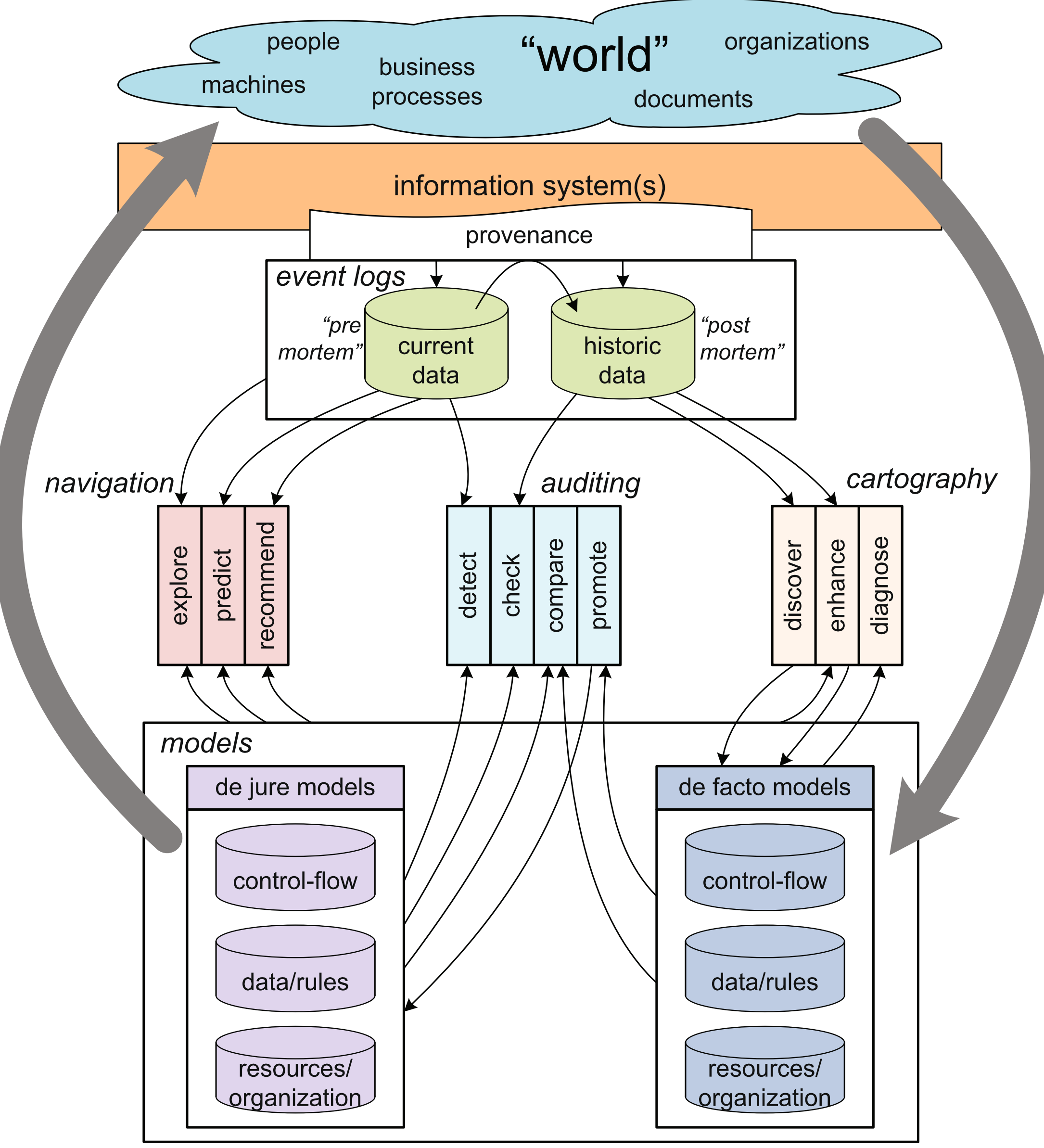

Figure 2.4: The process mining framework. [van der Aalst, 2011]

4. Process conformance

5. Process enhancement

2.4.1 Event log

Event logs are, together with the process models described in the next section, the foundation of process mining. They provide the input data for all types of process mining techniques. For

Case id	Event id	Timestamp	Activity	Resource
1	324556	12-06-2015:13.01	Checkin csv	Mechanical Engineer
	324559	12-06-2015:13.04	Query EDB	System
	324561	12-06-2015:13.06	Signal comparison	System
	324562	12-06-2015:13.07	Accept changes	Mechanical Engineer
	324563	12-06-2015:13.07	Notify collaborator	System
	324565	12-06-2015:13.10	Accept changes	Customer
	324567	12-06-2015:13.11	Persist changes	System
2	330101	13-07-2015:10.00	Checkin csv	Electrical Engineer
	330103	13-07-2015:10.02	Query EDB	System
	330105	13-06-2015:10.06	Signal comparison	System
	330106	13-06-2015:10.07	Reject changes	Electrical Engineer
	330108	13-06-2015:10.08	Persist changes	System

Table 2.6: An event log fragment

example for process discovery, an event log delivers the input for a process discovery algorithm which then creates a process model based on the events captured. Table 2.6 shows a simple sample of an event log containing two process executions. Each line corresponds to a single event. Each event is linked to a process instance via a case id, which is usually not known a-priori in process-unaware systems (this is an information which is derived via process discovery), but added here for the reader to help keeping the process instances apart. Each event furthermore contains a unique event id, a timestamp of its occurrence, a description of the activity performed and a resource or actor that performed the activity. Timestamp, Activity and Resource are called attributes and are optional.

Event logs can be constructed from data available in current information systems in manifold ways like flat text files, databases and message logs. Standardized XML-based formats like MXML (Mining eXtensible Markup Language), SA-MXML (Semantically Annotated MXML) and especially the IEEE standard XES (eXtensible Event Stream) [Günther and Verbeek, 2014], which replaces the former two and is the current de facto standard, have been defined to give event logs a standardized syntax which is understood by many process mining tools.

2.4.2 Process model

Process models are the second corner stone of process mining. A process model depicts an abstraction of a set of activities in real life or an information system. Notable notations are (in order of simplicity) transition systems, petri nets and BPMN (Business Process Model and Notation). There are *de facto models* describing processes as they were detected by process mining techniques and *de jure models* describing the processes as they should be.

Petri nets Petri nets are the standard process model used in process mining since most process mining algorithms use Petri nets as input or output. [Kalenkova et al., 2015] They allow the modeling of concurrency and provide an exact mathematical foundation, allowing for execution

and analysis. [Murata, 1989] A Petri net is a bipartite graph consisting of transitions (rectangles), places (circles) and directed arcs between transitions and places. Figure 2.5 shows a petri net of a simplified engineering process we have created based on some requirements given in chapter 5 and the event log given in 2.6.

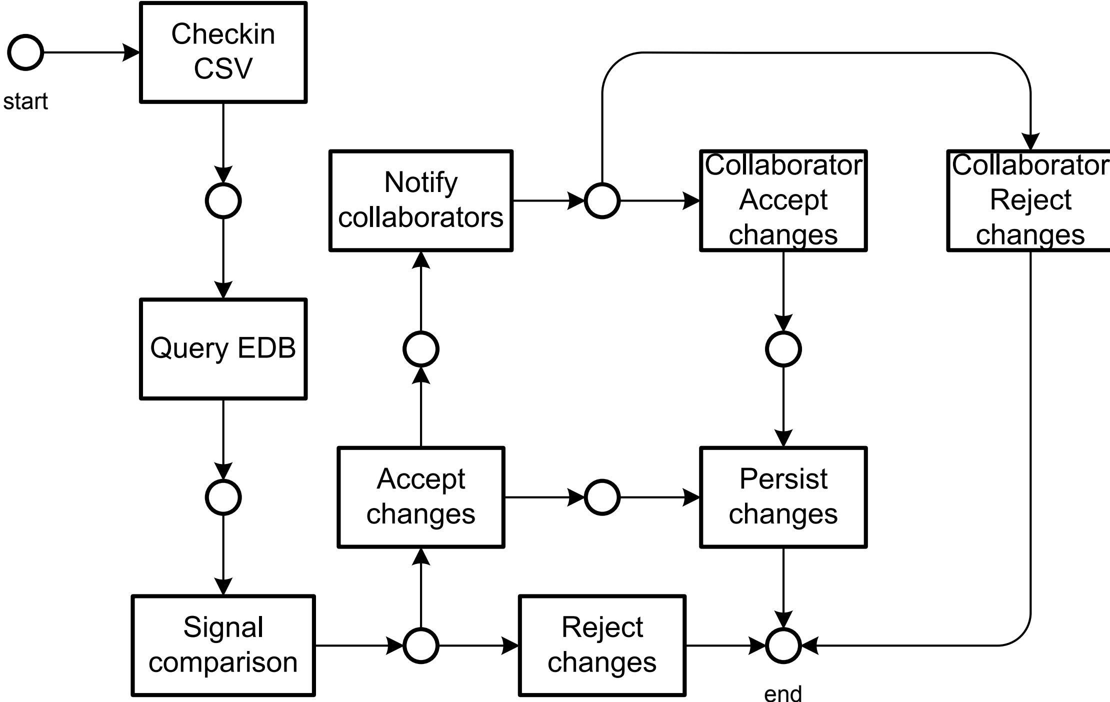

Figure 2.5: De jure petri net based on the event log in table 2.6.

2.4.3 Process mining types

Next we are going to introduce three basic process mining types as described in [van der Aalst, 2011] [van der Aalst et al., 2011]: process discovery, process conformance and process enhancement.

Process discovery is used to discover (de facto) process models from event logs without any a-priori information. It is the most basic and popular process mining type. There are various types of process discovery algorithms, like the alpha-algorithm [van Der Aalst et al., 2004], heuristic [Weijters and van der Aalst, 2003] [Weijters and Ribeiro, 2011], genetic [De Medeiros et al., 2007] or region-based algorithms [van Dongen et al., 2007] [Bergenthum et al., 2007].

Figure 2.6a shows the concept of process discovery. The input for the discovery algorithm is an event log, the output is a process model as e.g. a petri net or a BPMN-diagram.

Process conformance is used to find differences between a process model and an event log. [van der Aalst et al., 2012]

Figure 2.6 shows the concept of process conformance. The input for the conformance algorithm is an event log and process model, the output are some diagnostics.

Process enhancement is used to enhance given process models with an event log.

Figure 2.6c shows the concept of process enhancement. The input for the enhancement algorithm is an event log and process model, the output is a new process model as e.g. a petri net or a BPMN-diagram.

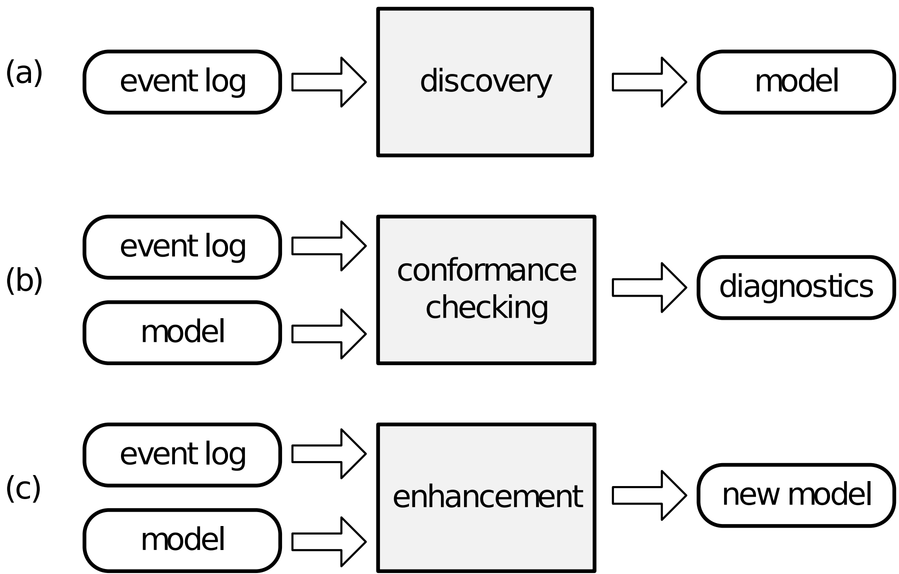

Figure 2.6: Three basic types of process mining. [van der Aalst et al., 2011].

2.4.4 ProM

ProM [van Dongen et al., 2005] is the swiss army knife of process mining tools available, supporting a plethora of different process mining algorithms. It is Java-based and offers a visual user interface where the user can select event logs, models and other input parameters and apply algorithms to perform various process mining tasks. The results can be visually presented and saved to various file formats (depending on what the algorithm produced).

ProM has already been used in other related work. For example in [Sunindyo et al., 2011c] the event-log based workflow validation approach by [Cook and Wolf, 1999] was applied to the Signal Change Management Workflow. A Petri net was used as process model for the validation. The conclusion was that Petri nets can help project managers grasp the big picture of the interactions between different engineering fields during the signal change process. However, an evaluation and comparison with other approaches was postponed to future work. In [Kalenkova et al., 2015] it was recently discussed that BPMN workflows could also be used for any process mining related tasks. While most process mining algorithms do not support BPMN directly, there are algorithms which can convert BPMN to Petri nets and vice versa. As the work-

flow representation in BPMN is more compact and better readable for business users it might be better to use BPMN for process modeling instead of Petri nets.

At the time of writing the current version of ProM, ProM 6, already supports a variety of BPMN algorithms.

2.5 Product Modelling

[Mordinyi et al., 2015] summarizes the results from previous research in the context of the EngSB concerning the integration of heterogeneous engineering tools and various data models required. Four approaches of engineering data integration were compared: Virtual Common Data Model (VCDM) [Waltersdorfer et al., 2010], Enterprise Service Bus (ESB)-like [Chappell, 2004], ESB-Domains only and Multiple Domains and one Engineering Object, with the latter coming out on top, because it is (1) robust against changes, (2) can hold multiple common models, (3) supports versioning, (4) services are re-usable, (5) incorporates data validation and (6) provides data sovereignty.

The Multiple Domains Approach with one Engineering Object includes tool data models (TDM) which are predefined for and originating from each tool in the engineering environment. It also includes tool domain data models (TDDM) which abstract from the TDMs and represent a common but tool independent model for the integration solution. Common Data Models ("Engineering Objects") are used to interconnect TDDMs and allow a common, discipline-independent view of data instances. Mappings allow the import and export between TDMs and TDDMs and transformations support the data conversion between TDDMs and engineering objects.

CHAPTER 3

Research Issues

In this chapter we will define the research issues and questions to pursue in this thesis which we will answer later in the Discussion chapter 8 with respect to the evaluation results in the Evaluation chapter 7 and the Related Work chapter 2.

Based on the Project Observation and Analysis Framework [Sunindyo, 2012] applied to the domain of Automation System Engineering Domain we derive the following Process and Product Analysis Framework as seen in figure 3.1 which is similar to the Process and Workflow Management Cycle in ASE projects [Winkler et al., 2014]. Indicated from one to four are the main challenges encountered during this work: (1) process definition and modelling, (2) process implementation, (3) data collection and (4) process and product analysis. While this thesis includes all four steps, our research questions mainly focus on steps two, three and four, the process implementation, the data collection and the process and product analysis.

3.1 RI-1 Process definition and implementation

"How do we model and implement the processes to fulfill the new requirements?"

While previous work has dealt with the modeling of similar engineering processes, none have implemented the processes yet. Additionally, new requirements have arisen which require changes to already existing process definitions and the definition of new processes based on suggested methods already presented in previous work. While we will also provide new process definitions the focus in this issue will be on the implementation part which has not been tackled yet.

3.2 RI-2 Bridging between process and product data

"How do we link process and product data and how should these links be structured?"

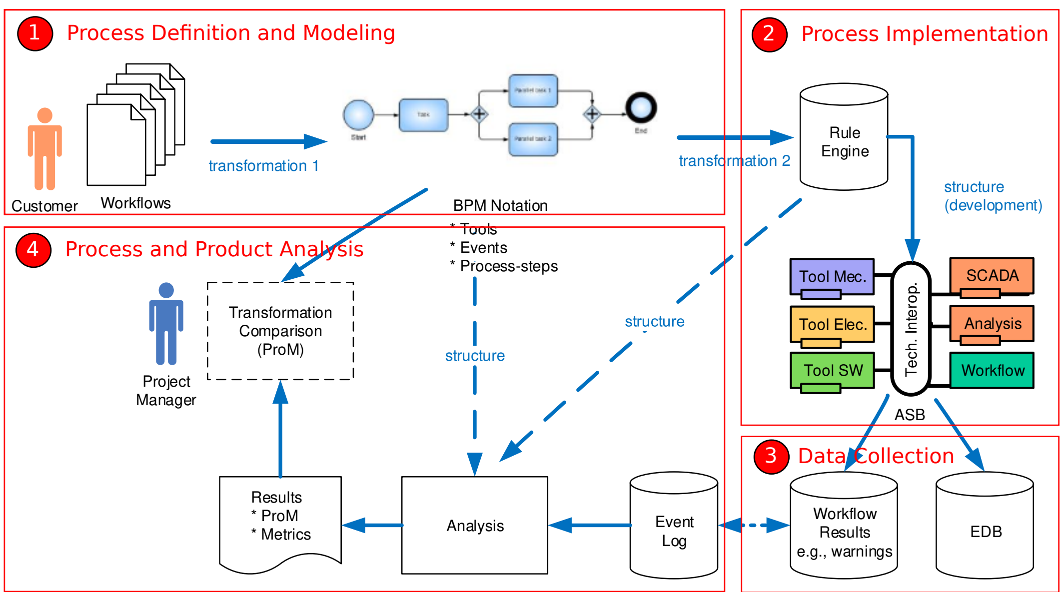

Figure 3.1: Process and Product Analysis Framework based on [Sunindyo, 2012]

Since process and product data may differ in terms of their models and storage locations, a bridging is required. This requires either a reshaping of existing models to merge and incorporate both into a single model, or the placement of bridging events in the implemented process in case the process and product models remain separated. Furthermore the structure of the process data model has to be designed to be capable of fully representing product data and being immune to future model changes of product data.

3.2.1 RI-2.1 Structure of the Process and Product Data Model

"How should the process and product data model be structured in order to gather necessary data?"

This research issue deals with the structure of the process and product data models. A process data model has to be defined that contains events collected during process execution. The model should be capable to fully represent process data and at the same time changes to product data must not have any significant effects on the process model and vice versa. It needs to be determined, also with respect to existing solutions, whether process and product data should be merged or stored separately

3.2.2 RI-2.2 Placement of Bridging events

"Where should bridging events be placed and how many should be placed?"

In case the process and product models are separated we need to place bridging events in the implemented process. These are used to link between process and product data and are required

to formulate queries spanning over both process and product data. We need to determine how many bridging events are required and derive good locations for the events in the process model. Additionally we also have to determine the content of these events that link to the product data.

3.3 RI-3 Process efficiency and effectiveness

"How efficient and effective is the implemented process at solving the requirements?"

In this research issue we will analyze the implemented process in terms of its efficiency and effectiveness to solve current problems as described in the Use Case chapter 5 and future problems which may require process extensions and changes. We base our understanding of "efficiency" and "effectiveness" on the definition given in the Business Dictionary [1]:

- efficiency is *The comparison of what is actually produced or performed with what can be achieved with the same consumption of resources (money, time, labor, etc.).*
- effectiveness is *The degree to which objectives are achieved and the extent to which targeted problems are solved.*
- to differentiate further between the terms the dictionary states that *In contrast to efficiency, effectiveness is determined without reference to costs and, whereas efficiency means "doing the thing right," effectiveness means "doing the right thing."*.

Therefore the process efficiency and effectiveness of the implemented process are investigated, especially the effort and correctness of executed scenarios (compared to the current solutions), whether the new solution allows stakeholders to formulate queries which a) couldn't be formulated before and b) could only be formulated with an excessive amount of effort and finally how the process can be extended or changed.

3.3.1 RI-3.1 Initial effort and correctness

"How much effort does the execution of the implemented process cause and how does it compare to the traditional approach?"

In order to evaluate the solution we need to asses if the implemented processes are capable of replacing the traditional approach, how efficient it is compared to the traditional approach. This includes the definition and execution of test scenarios and a case study. The test scenarios will be defined to demonstrate the correctness of the implemented processes and will be designed to trigger all important features of the processes while being as brief as possible. Therefore it will be based on artificial or modified, real test data. The case study will use real data from an industry partner in the area of power plant engineering. To conduct and define the tests, a test setup has to be created which can be used in all test scenarios. Therefore it has to be capable of executing multiple processes sequentially in a test run and provide all types of test and user input parameters, like test data files and transformers.

[1] Business Dictionary: http://www.businessdictionary.com/

3.3.2 RI-3.2 Formulating Queries

"Can new queries be formulated, allowing deeper insights with less effort?"

In order to get deeper insights into process and product data, queries on the database(s) can be defined for or be defined by stakeholders. We are going to examine whether the solution allows to formulate queries which a) could not be formulated before and b) could only be formulated with an excessive amount of effort and complexity. This indicates whether or not the solution improves the process effectiveness and efficiency. We will formulate a variety of queries which measure key performance indicators (KPI), apply them to the solution and evaluate the results. The queries will be executed directly on the database and not via higher level interfaces.

CHAPTER 4

Research Approach

In this chapter we will describe the research approach used in this work, as well as the methodology that lead the whole research process. We will introduce the theoretical foundations, as far as the scope of this work allows, and provide links to other chapters that describe the concrete application in more detail.

4.1 Design Science Research Approach

The research approach applied in this work follows the design science research approach as proposed in [Hevner et al., 2004]. This information systems research paradigm is complementary to the behavioral research approach. While the behavioral research approach seeks to create or refine new theories based on the observation of phenomena occurring in an environment, design science seeks to create artifacts which help to solve problems in an environment, based on solid scientific grounding. These artifacts are defined as constructs, models, methods and instantiations:

Constructs provide the vocabulary and symbols used to define problems and solutions. Examples for constructs are Arabic numbers or zero, or nodes and edges in graphs.

Models are abstractions and representations of systems. Constructs are required to build these models. Examples for models are an entity-relationship model or a business process model.

Methods are algorithms and practices that can be used to solve problems or build models.

Instantiations are fully or partially implemented systems or prototypes.

Figure 4.1 shows the design science research cycles defined by [Hevner, 2007]. Positioned in the center is Design Science Research with its goal to build artifacts. The requirements for the artifacts to build are derived from an environment or application domain which provides

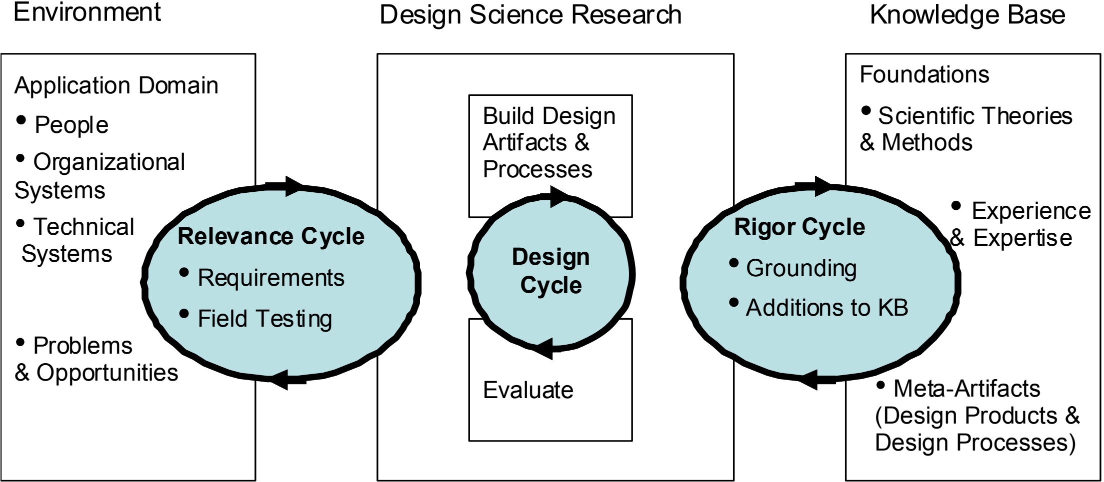

Figure 4.1: The design science research cycles [Hevner, 2007]

problems and opportunities in organizational or technical systems that need to be solved. The environment is in constant exchange with the research process via the relevance cycle. The requirements may change or additional requirements may appear and the created artifacts may need some field testing in order to fully evaluate their utility. Just as important as the relevance cycle is the rigor cycle which ensures that the creation of the artifacts is grounded in scientific knowledge from a knowledge base. This knowledge base provides scientific theories and methods, experience and expertise from researchers, as well as meta-artifacts which include design products and processes. Ideally, design science research not only leads to new artifacts, but also to new additions to the knowledge base as in e.g. papers released in a journal. Finally, the design cycle between the artifact creation and evaluation ensures that constructed artifacts meet the requirements and are solidly grounded.

4.2 Design Science Research Methodology

While the design science research approach by Hevner et al. provides a solid framework, we also need a concrete methodology to pursue our work within this framework. Pfeffers et al. [Peffers et al., 2007] proposed a process consisting of six activities to conduct design science research which we will follow in this work and can be seen in Figure 4.2. The activities are iterated as often as necessary and do not need to occur in sequence. Following is a brief explanation of the activities and how we implemented them:

Activity 1: Problem identification and motivation The problem has to be identified and the value of the solution justified. Complex problems should be divided into smaller parts. We have given the motivation and initial problem statement in the Introduction 1.

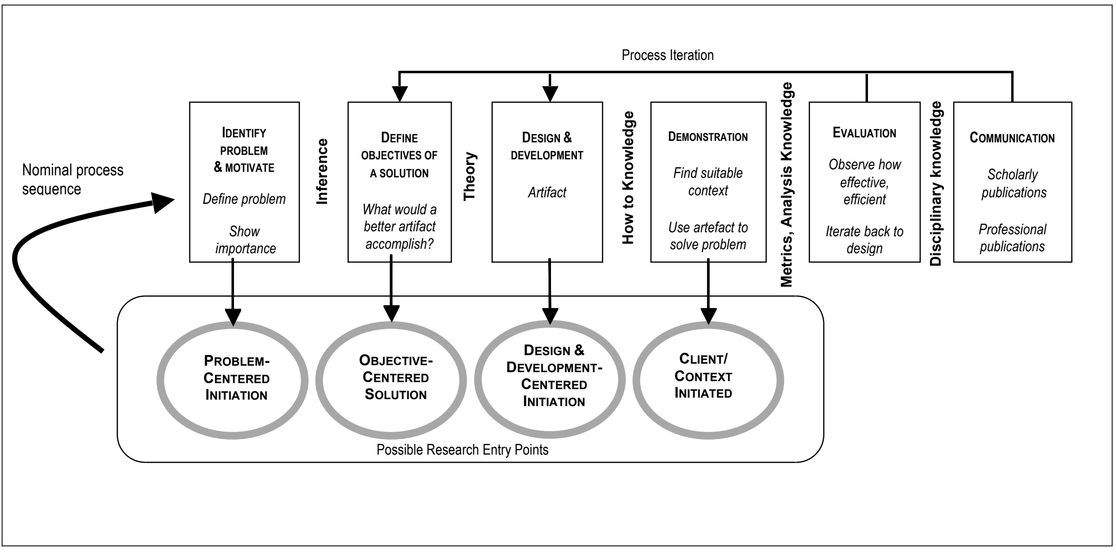

Figure 4.2: The design science research methodology [Peffers et al., 2007]

Activity 2: Define the objectives for a solution Outgoing from the problem definition the objectives for a solution are defined. We address this in the Use Case 5 and Related Work chapters 2, where we ensure a proper implementation of the relevance and rigor cycles mentioned above.

Activity 3: Design and development during this activity the actual artifact is created. The design and development heavily depends on results from literature research, ensuring rigor. We will describe it closer in the Related Work 2 and Solution Approach 6 chapters.

Activity 4: Demonstration The use of the artifact is demonstrated by solving instances of problems. This can be done by experimentation, simulation, case study, proof or other appropriate activity. We have done this in the Evaluation chapter 7.

Activity 5: Evaluation The artifact is evaluated with respect to the objectives defined during activity 2. This is also done in the Evaluation chapter 7.

Activity 6: Communication To close the rigor cycle, communication has to take place in order to return new knowledge to the knowledge base. This can be done by scholarly or professional publications. This whole thesis in itself is an implementation of the communication activity. Additionally, in the conclusion chapter we will also address future work which could enhance the disciplinary knowledge or stimulate the building of future solutions based on this work.

Pfeffers et al. state that research entries are possible at each of the first four activities. For this work we have chosen a design and development centered initiation at activity three. The reason for this was that we already had a working prototype that was able to handle product data but not process data and wanted to introduce proper process models to gather process data.

Therefore we wanted to develop an instantiation of an artifact that supports process models. So starting from activity three, we asked ourselves what kind of problems this artifact could address (activity one), by looking at a concrete environment it could be used in and defined objectives of a solution (activity two).

CHAPTER 5

Use Case & Environment

As required by the chosen research approach, we will introduce environment characteristics and requirements that originate from this environment for the created artifact in this chapter. The structure of this chapter is as follows: In the first part we will discuss the use case in detail and present preceding papers about the engineering process and new requirements which have been brought up by the industry partner in the recent past but which have not been tackled yet. The second part of this chapter deals with the technical system where the solution will be integrated and existing services which have to be taken into consideration.

5.1 Use Case - Change Management in multidisciplinary and heterogeneous engineering environments

The concrete use cases covered in this work stem from the collaboration with an Austrian industry partner in the hydro power plant engineering area with focus on turbines, generators and pumps. They conduct power plant projects all around the world from the planning phase all the way to commissioning. The products are created with various tools from different vendors which were not designed with interoperability in mind, except for some basic export functionality like csv-export. The synchronization between this tools has to be done by sending these exported csv-files via e-mail to all necessary participants who in turn have to synchronize and match them with their data manually or with a plethora of individually tinkered scripts.

Our goal is to provide an artifact that enables product synchronization in an automatized way and that at the same time allows for deeper insights into the process and product development for engineers, project managers and customers. Figure 5.1 gives a general overview over the use cases (broken down from the scenarios of the introduction 1) of the system and the identified use cases, which are:

Synchronization of product data. An engineer wants to synchronize the product data he created with the tools of his discipline with the existing product data (if available) and store the

result as a new version in the system.

Notification of collaborators. Sometimes collaborators should be informed about the pending changes by the engineer. These collaborators can be other engineers, customers or manufacturers who want to be notified via email and log into the system to review the notifications.

Decision about changes. Notified collaborators want to accept or decline changes made to the product. If changes are declined they should be excluded from the final commit.

Updating signal states. Signals transition through various phases during their development process. (see also NotificationMapper) This is indicated by states that the engineer wants to apply on existing signals in the system.

Monitoring of running process instances. Project managers want to have an overview of running synchronization process instances with additional info like who has started to process, who was notified, whose response is still pending etc.

Viewing of process statistics. Project managers want to view various statistics of finished (or aborted) synchronization process instances.

Viewing of product statistics. Project managers want to view various statistics about the product and product change over time.

5.1.1 Previous use case versions

Next we are presenting previous versions of the use case as a reference for the comparison and evaluation with the solution approach in this work.

First conception and implementation

The first version was introduced in [Winkler et al., 2011] as "Signal Change Management Workflow" and is shown in figure 5.3. The process iterates over a list of signals and each signal is compared with the list of existing signals from the Engineering Database (EDB). The outcome of each iteration is that the signal is either similar, new, changed or deleted. If the signal is similar, no action has to be taken. In the other three cases the engineer who has started the process then decides whether or not to persist the result of the signal comparison. Finally, notifications are sent to collaborators (however a response is not expected or supported) and the process is repeated with the next element in the signal list.
Special importance has to be paid to the events thrown in the process. These can be used to measure time metrics like the duration of the whole process (E1 E_checkin_started & E10 E_checkin_completed) or of a single signal comparison iteration (E2 E_signal_comparison_started & E9 E_signal_comparison_completed). Other interesting metrics that can be measured are how many signals during a process were added (E6 E_signal_new), updated (E5 E_signal_changed),

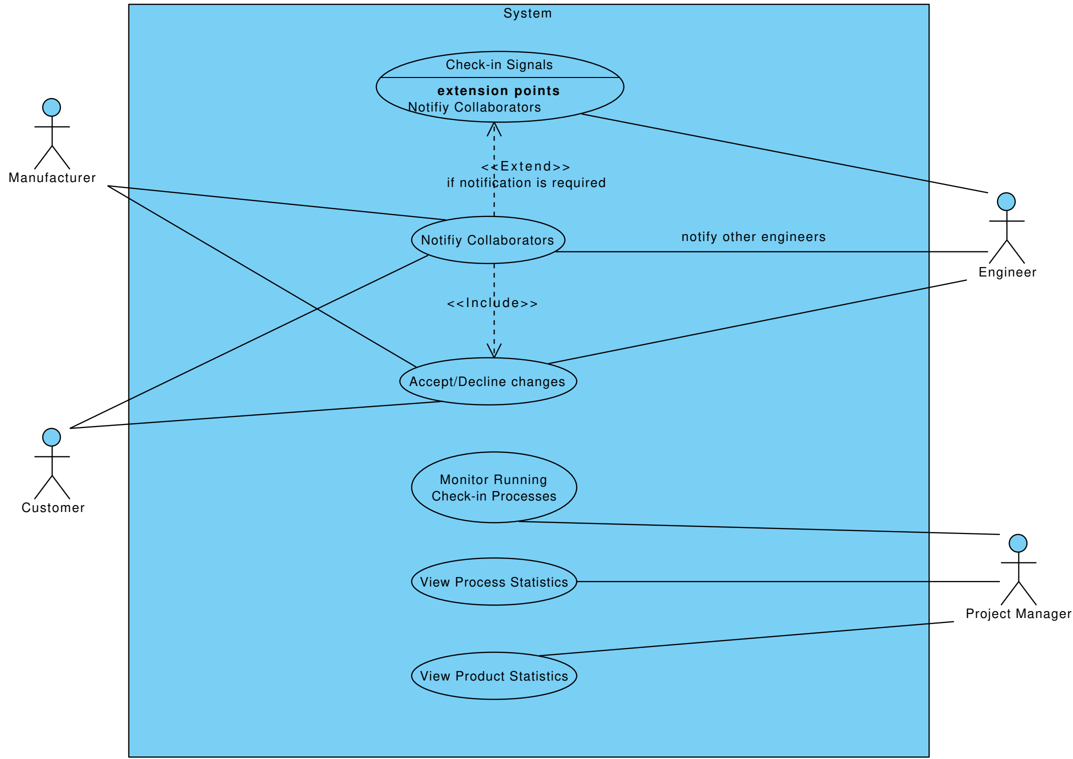

Figure 5.1: Identified Use Cases based on the requirements of the industry partner.

deleted (E7 E_signal_deleted), left unchanged (E3 E_signal_similar) or left unchanged by decision (E4 E_signal_not_changed) and how many notifications had to be sent (E8 E_notification). This workflow was implemented as a prototype without using a BPM engine and the events were analyzed using ProM.

Modeled as BPMN

In order to implement the workflow using a BPM engine it was later refined and modeled using BPMN [Winkler et al., 2014]. The resulting BPMN-diagram can be seen in figure 5.3 which displays the main process, figure 5.4 which displays the signal comparison subtask and figure 5.5 which displays the human interaction subtask.

Main Process The main process starts with the upload of a csv file which was produced by the export function of the engineering tool. Next the EDB is queried to obtain the existing product data. Then the automated signal comparison is triggered multiple times until all signals have been compared with the existing data. After this has finished, the human interaction process is started where the user can decide which changes should be persisted in the EDB. Finally a report is created and the process is finished.

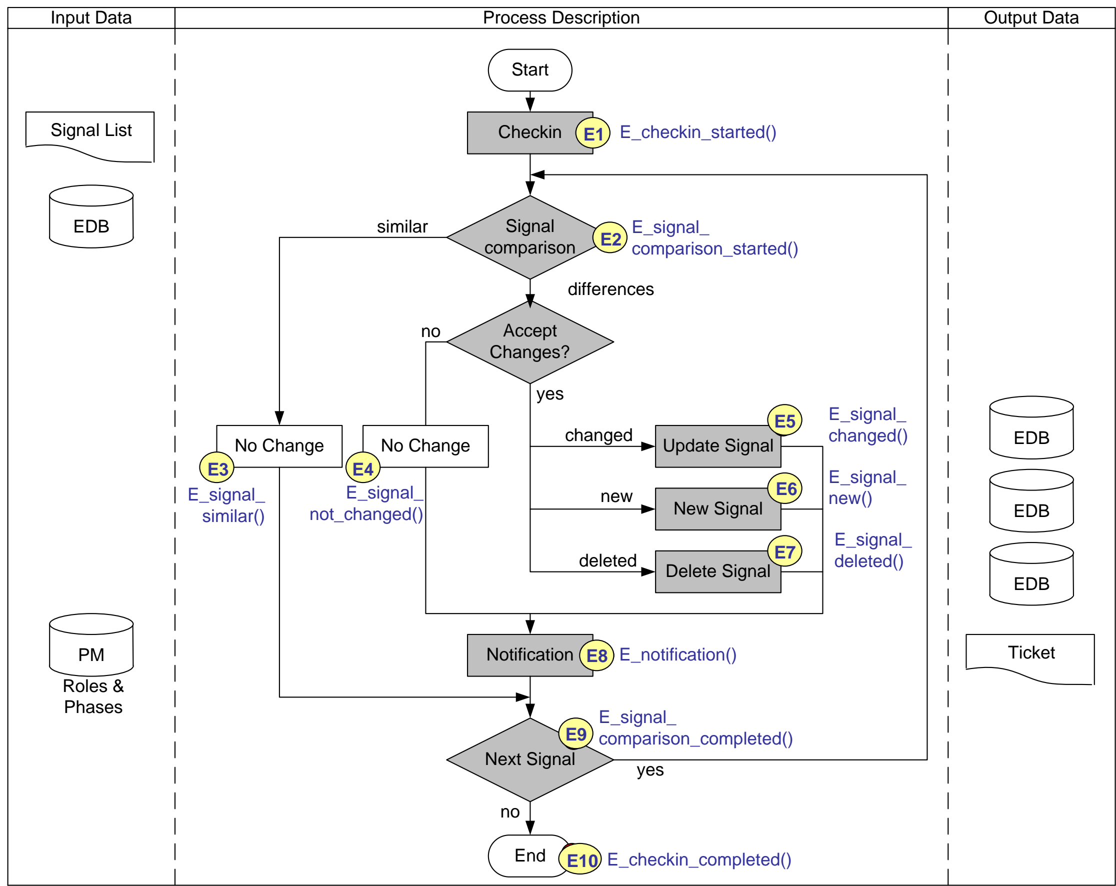

Figure 5.2: Signal Change Management Workflow [Winkler et al., 2011]

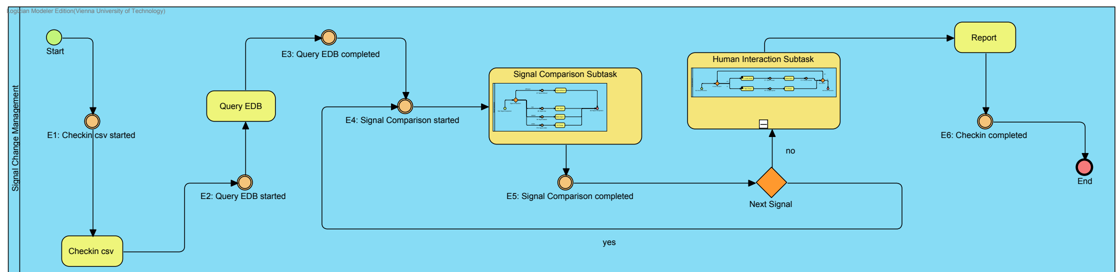

Figure 5.3: Signal Change Management Process [Winkler et al., 2014]

Signal Comparison Subtask This subprocess is triggered for every signal which was uploaded by the engineer. For every signal there are four possible mutual exclusive outcomes: Signal different, New Signal, Delete Signal or No Change.

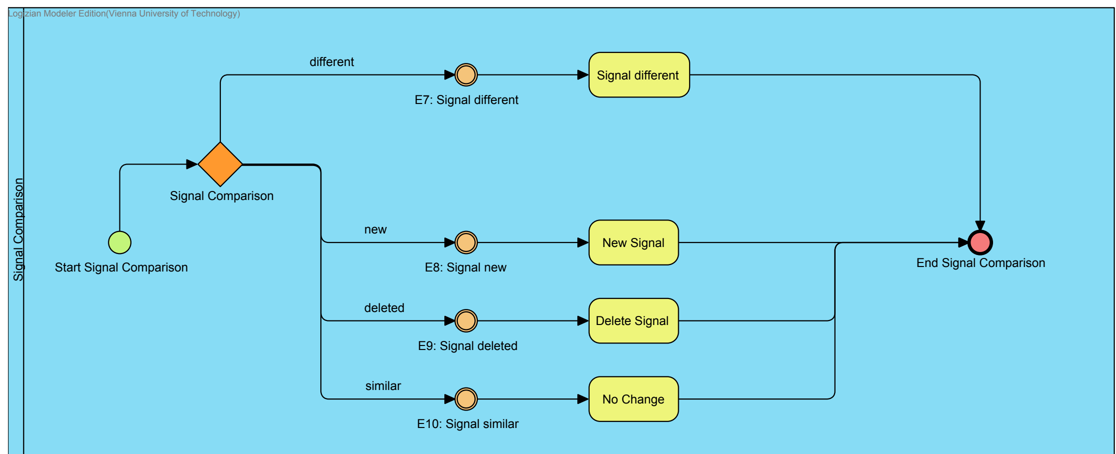

Figure 5.4: Signal Comparison Subtask [Winkler et al., 2014]

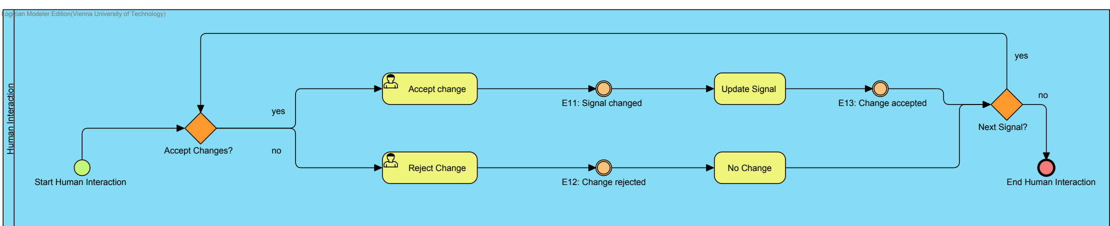

Figure 5.5: Human Interaction Subtask [Winkler et al., 2014]

Human Interaction Subtask This subprocess is triggered after the signal comparison has completed. For each signal the user has to decide whether the results from the signal comparison should be persisted or not. When he accepts a change the signal is updated, otherwise there is no change.

Again multiple events are placed which allow the measurement of process metrics. In the main process there are process duration (E1: Checkin csv started & E6: Checkin completed), query duration (E2: Query EDB started & E3: Query EDB completed), signal comparison duration for each signal (E4: Signal Comparison started & E5: Signal comparison completed). In the signal comparison subtask the number of identified new signals (E8: Signal new), changed signals (E7: Signal different), deleted signals (E9: Signal deleted) and unchanged signals (E10: Signal similar). And in the human interaction subtask the number of accepted (E11: Signal changed) and rejected (E12: Change rejected) changes and the duration of persisting a change (E11: Signal changed & E13: Change accepted).

5.1.2 Notification Handling

A new feature that has not been implemented yet but which was required by the industry partner is the notification handling. When an engineer makes changes to the product, it may be required

State	Description
St1	Signal from sub-supplier
St2	Signal from customer
St3	Approved KKS number and description from customers
St4	OPM hardware/software I/O addresses assigned
St5	Electrical Engineering (Eplan) started
St6	Manufacturing of hardware units
St7	Manufacturing finished and approved
St8	After commissioning

Table 5.1: Signal States

Notification	Recipient
N1	customer
N2	process engineer (OPM)
N3	electrical engineer (Eplan)
N4	manufacturer (wiring)

Table 5.2: Ticket types

to notify other engineers, customers or manufacturers about these changes and obtain their permission. The notification should include only the signals relevant to the recipient. E.g. if ten signals are checked in and only two require notification, then the recipient should only be notified about these two. He should then be able to log into the system and view the lists of changes which include lists for new, changed and deleted signals. Finally he should be able to reject or accept the changes for each signal and mark his notification as completed.

Whether a notification is required or not depends on rules which we are going to explain next. The input for the rules are the state of the signal and the changes that are pending. The output is the list of the recipients that should be notified. Table 5.1 shows a list of all eight states that are used at the industry partner. They are ordered chronologically, starting when a signal is received which may be either received from a sub-supplier or a customer (state 1 and 2), after that the next step (state 3) is reached when the Kraftwerk Kennzeichensystem (english: Power Plant Designation System) number (KKS-number) and description are approved by the customer. Later (state 4) OPM hardware and software I/O addresses are assigned. Next (state 5) the electrical engineer is started, followed by the manufacturing of hardware units (state 6). Finally, after the manufacturing is finished and approved (state 7), the product is commissioned (state 8).

Table 5.2 shows a list of all five different notification recipients reported by the industry partner. They correspond to some of the actors seen in 5.1.

Table 5.3 shows the mapping between changes and notifications. Horizontally we have the change criteria that require notification, and vertically we have the states. Changes which require notification are changes to the KKS-number (kks0, kks1, kks2, kks3), signal description (func-

State/Change	KKS-number	Signal description	I/O channel
St1	-	-	-
St2	-	-	-
St3	N1	N1	-
St4	N2	N2	N2
St5	N3	N3	N3
St6	N3	N3	N3,N4
St7	N3	N3	N3
St8	N3	N3	N3

Table 5.3: The Notification Table

tionText) and I/O channels (channelNumber and IONumber respectively). (Note that this list is not exhaustive; there may be additional changes which require notification which have not been fully specified yet.) When signals are in state 1, all changes are allowed without notification, in state 3 the customer needs to be notified when the kks-number or signal description is modified but not when I/O channels are modified. In state 4 the process engineer has to be notified when any of the notification criteria are met, in state 5, 7 and 8 the electrical engineer. State 6 is a special case where two actors must be notified simultaneously: the electrical engineer and the manufacturer when the I/O channels are changed. For changes to the KKS-number and signal description only the electrical engineer has to be notified.

As we have already mentioned the presented notification handling is not exhaustive. It may change or be extended in the future (e.g. by addition of new states or recipients), therefore it is important to make this component configurable at run-time to add more states, criteria and recipients as needed.

5.1.3 Metrics and Key Performance Indicators

In [Winkler et al., 2011] first KPIs were defined: number of checkins, number of signals during checkin, number of unchanged signals, number of accepted signals, number of rejected signals. In [Moser et al., 2011] this list was further extended. Together with our industry partner we have created a list of additional KPIs (and real-world approximations) which are interesting to project managers:

- Effort for inter-tool synchronization (30 mins)
- Frequency of inter-tool synchronization (1-2 per month)
- Effort for changing the signal status (n/a)
- Effort to collect data (=query) (several days)
- Number of signals
- Number of signals per phase

- Number of components
- Number of components per phase
- Number of changes
- Number of changes per phase
- Number of changes per person/discipline
- Number of changes per internal or external cause

5.2 Environment

In this section we will describe the existing environment where the use case will be implemented.

5.2.1 Open Engineering Service Bus

[Biffl and Schatten, 2009] introduced the concept of the Automation Service Bus (ASB), an adaptation of the commonly known Enterprise Service Bus (ESB) [Chappell, 2004], specifically designed for the needs of multi-disciplinary engineering environments by supporting the integration of various tools. This concept was subsequently implemented as Open Engineering Service Bus (OpenEngSB[1]) [Pieber and Biffl, 2010], an open source, OSGi-based[2] middleware platform.
In the OpenEngSB, typical use-cases for tool collaboration involve the orchestration of multiple services, glued together with business logic that requires user interactions.

5.2.2 OpenEngSB services used in this work

In this section we will introduce other pre-existing services in and around the OpenEngSB that we will use for our implementation.

Engineering Database

The Engineering Database (EDB) [Mayerhuber, 2011] provides an interface to versionize different kinds of Plain old Java objects (POJO). These POJOs are implementations of the interface OpenEngSBModel which provides methods to generically access fields of the object and are called Engineering Objects in the context of the OpenEngSB. The EDB provides a PersistInterface for write operations on the database and a QueryInterface for read operations. It supports versionized storage of Engineering Objects by bundling them together into commits, split into sets of new, deleted and updated Engineering Objects. The QueryInterface Service provides methods to query the current content of the EDB. It supports a simple text-based query and a

[1] Open Engineering Service Bus: http://www.openengsb.org
[2] Open Services Gateway Initiative (now OSGi Alliance): http://www.osgi.org

more powerful QueryRequest object, that supports more parameters via a fluent API. The result is always a subset of the current content of the EDB, returning the most recent version of currently non-deleted signals.

Engineering Database Index

The Engineering Database Index (EDBI) [Rausch, 2014] provides a service which provides means to execute more efficient and native SQL queries on versioned product data than the EDB. When the feature is activated, the EDB passes all commit operation it performs to the EDBI, which persists the data in its own format in the SQL database. The core tables for querying are the HEAD-table and the HISTORY-table. The HEAD-table contains all current signals in the EDB (just as a default query would return from the QueryInterface) and the HISTORY-table contains all signal versions. Due to the flat nature of how signals are stored in the EDBI, it is much easier and efficient to perform complex queries for analysis. The downside is that the EDBI is not usable for all kinds of models the EDB can handle. Hierarchical models can not be handled in its current version.

Transformer Services

The transformer services [Pircher, 2013] allow the creation, modification and execution of tool-to-model (Input) and model-to-tool (Output) transformers. It provides a user interface to configure transformers which are essentially smooks [3] transformation descriptions and services executing these transformers.

Analyzer Service

The Analyzer Service is an auxiliary service created to compare different lists of Engineering Objects for changes. Typically the input is a new set of Engineering Objects from a tool and a set of existing Engineering Objects from the EDB. The service determines which tool objects are new, deleted, changed or unchanged relative to the existing data. Additionally it can be used to perform other checks on the data at hand, e.g. filtering out duplicate Engineering Objects.

[3] Smooks: http://smooks.org

CHAPTER 6

Solution Approach

In this chapter we will describe the artifacts we have implemented to solve the problems and satisfy the requirements introduced in the use-case, research issues and related work chapters. In section 6.1 we will start with an architectural overview where we describe the new components in context with the existing components. Section 6.2 presents the reasons for selecting the Activiti process engine for the solution. Next in section 6.3 we will describe the process definitions in detail, followed by the process implementation in section 6.4. In section 6.5 we will introduce the testing environment we created to test the implemented processes in various scenarios. The chapter is concluded with section 6.6 where we will describe the facilities we will use for process and product analysis.

6.1 Architectural Overview

We have already discussed a few architectural features of the environment and its components in chapter 5. In this section we will describe the architecture of our solution and how it utilizes these components. Figure 6.1 gives an overview of the architecture. We have separated the components into three groups: Components that are focused on process execution and data collection, components that are focused on product data storage and auxiliary services which can be involved in either process or product focused tasks or both, but which do not persist any process and product data.

6.1.1 Process Focused Components.

Process focused components are components which are required for process execution and definition and include the process implementations themselves and the Activiti process engine.

Process Implementations. These are the central contribution of this work and consist of a Check-In process and a Status Update process. These processes are deployed on the process

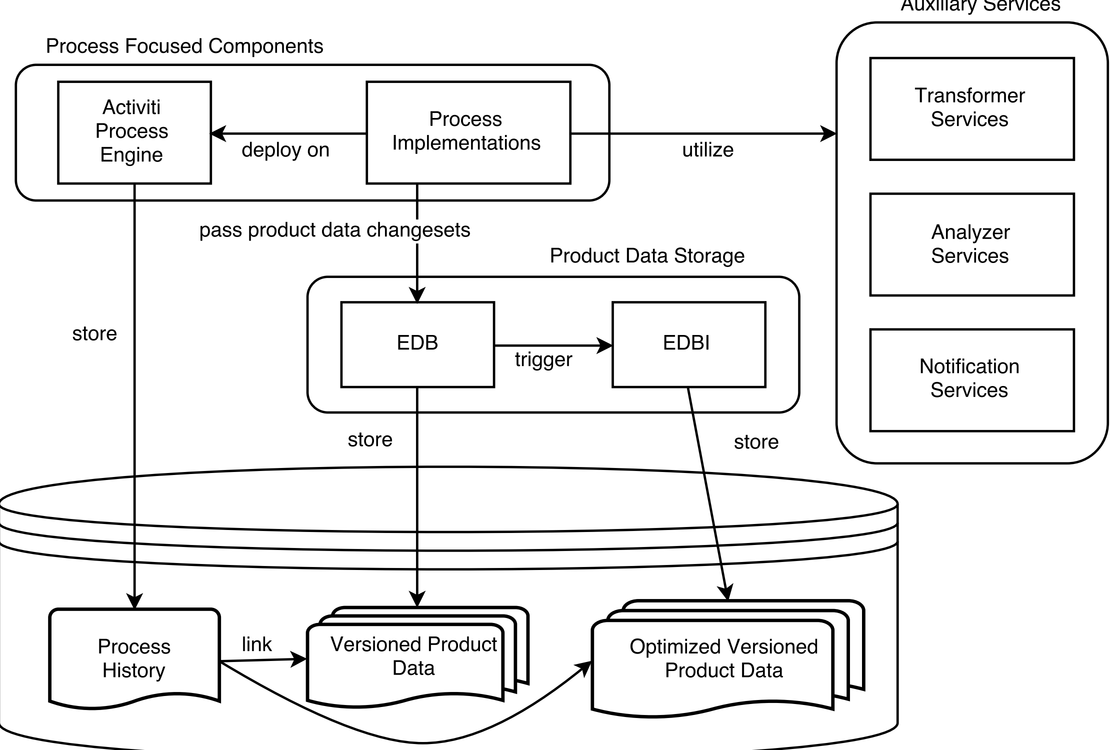

Figure 6.1: The Component Architecture grouped into Process focused, Product focused and auxiliary components.

engine and orchestrate communication with all other components and services. We will present a detailed overview in sections 6.3 and 6.4 of this chapter.

Activiti Process Engine. The process engine controls the execution of deployed processes. It executes automated tasks and waits for and collects input for user tasks. Each process element and variable is logged automatically into a process history where it can be used for analysis. We will present a detailed explanation of why we selected Activiti for this task in section 6.2 of this chapter.

6.1.2 Product Data Storage Components.

These components are responsible for persisting versioned product data. The main component used by the process is the Engineering Data Base (EDB) which stores the product data grouped into commits into the general database. Each commit also triggers the Engineering Data Base Index (EDBI) which stores the product data in an optimized way for more efficient querying. For more information on these components we refer to chapter 5 where we have already given a more detailed overview of these components.

6.1.3 Auxiliary Services.

These services are used by the process implementations during process execution. While they focus on process or product specific tasks or both, they do not themselves persist any process and product data. We have already described the transformation and analyzer services in chapter 5. The Notification Services are described in section 6.4.2 of this chapter.

6.2 Activiti

Activiti is the process engine which we will use for our process implementation.

We have had some experience with process engines before starting this thesis, but Activiti has shown to be the most viable candidate for this solution. The main reasons for using activiti are the following: Java compatibility, OSGi compatibility, open-source and free of charge, suitable license, full BPMN 2.0 support, extendable and customizable, automated logging, REST API, ease of testing, good documentation and an active community.

Java Compatibility Since the OpenEngSB is java based we also need a java compatible process engine.

OSGi Compatibility The OpenEngSB is running in an OSGi environment and therefore the process engine has to be fully OSGi compatible. This means it has to be properly defined as a bundle and define all its imported and exported packages in the manifest file. This is actually a K.O.-criterium for many of the available java-based process engines.

Open-source and Free of Charge The CDL focuses mainly on open-source software. Open-source allows us to create our own forks with our own modifications and hacks when we need them. It also means we can fix bugs ourselves. Being free of charge is also an important criterium for software in an academic environment with limited funding. This rules out process engines like camunda (which is itself a fork of activiti with closed source additions) and SAP Business Process Management.

Suitable License It is important for the used process engine to have a flexible license that does provide enough freedom to not have to bother with publicizing code when we do not want to. While the OpenEngSB itself is open-source, there might be some implementations based on it for customers which should not be available to the public. Activiti uses the Apache Software License 2.0 [1] which is perfect for our needs.

Full BPMN 2.0 Support obviously we want to have full BPMN support since we want to implement a BPMN process. Most process engines support BPMN 2.0 so this was not the most important criterium for selecting activiti. Note that while the most recent version of BPMN is 2.0.2, no workflow engine actually supports it officially at the time being.

[1] Apache Software License 2.0: http://www.apache.org/licenses/LICENSE-2.0.html

Level	Notes
none	On this level history archiving is deactivated. However, it provides best performance.
activity	This level only stores process and activity instances as well as top level process instance variables (at the end of a process instance).
audit	This is the default level. It is like the activity level, except that process instance variables are updated immediately and additionally form properties are recorded.
full	This is the highest possible level. It is like the audit level, except that it also records updates to process variables.

Table 6.1: Activiti History Levels [Rademakers, 2012]

Extendability and Customizability Ideally the used process engine is extendable and customizable. For example while Activiti offers a default implementation for user tasks using the Vaadin web framework[2] to render forms, we might want to use a different framework like e.g. wicket. Activiti's architecture allows us to do that with an acceptable amount of effort and implement our own user interface when required. Another reason (also related to user tasks) for a customizable process engine are the input parameters of user tasks. Activiti only supports basic form input type parameters like string, long, enum, date and boolean. However we may also want to have more complex object types as input parameters (e.g. when a user selects objects from a list) and Activiti provides the means to define and add our own types.

Process History A process history that can be queried is an important feature that we want to utilize. Activiti offers such a configurable feature that logs information such as activity start/end times, and process variables and their values that were used in a process instance. Table 6.1 shows the available log levels and their effect. This integrated process history has the huge advantage of reducing the amount of boilerplate code required for logging to zero, thus keeping the code of our task implementations clean. We do not have to manually put our log calls into the code or use difficult to implement and debug extensions like aspect oriented programming or add additional events to the workflow for logging purposes. This historical process data can be either queried natively with SQL or via Activiti"s HistoryService. We can use this data to transform it to XES or other formats for e.g. process mining tasks.

REST API Activiti allows access to all of its services via Represential State Transfer (REST). This is important because we want to develop additional, external tools that should be able to use the process. For example the Engineering Object Editor is an Excel plugin that communicates with the OpenEngSB and also allows for modification of product data.

Ease of Testing Since we are using test-driven-development to ensure a minimum amount of quality even for the prototype, it is important to be able to write tests for every component on

[2] Vaadin Web Framework: https://vaadin.com/

every level, be that unit, integration or acceptance tests. Activiti specifically provides features for unit testing and due to its OSGi compatibility it is also easy to write integration tests using pax-exam, which is the default test framework for OSGi/karaf based implementations like the OpenEngSB. Activiti also facilitates the injection of user input via its FormService, allowing us to demonstrate our solution without actually having to implement a user interface.

One small caveat however is that Activiti does not have proper process simulation support. There is the activiti-crystallball package planned for that but it is in a very early development stage and lacks a complete documentation. As a result we can not unit test our implementation easily, since it relies heavily on other OSGi based OpenEngSB services (which will be described later) which we have to mock manually. However, for our evaluation purposes integration testing is far more important because it fully demonstrates the working solution.

Good Documentation A good documentation is important to avoid having to waste time trying to figure out things by just looking at the code. Activiti's API is well documented and on top of that additional detailed materials exist like books which describe the implementation of Activiti in smalls scenarios as well as provide working code samples which we have consulted frequently during the course of this work.

Active Community An active community means that it is easier to get support, be it bugfixes, new features or just competent answers to open questions. Activiti has very active user and developer forums and developers engaging with the community. The code base receives updates almost daily.

6.3 Process BPMN definitions

6.3.1 Check-In Process

In the use case chapter 5 we have already introduced a previous BPMN definition 5.3, 5.4, 5.5. Based on the additional requirements, implementation experience and selected process engine we present the redefined BPMN process which we have implemented which can also be seen in figures 6.2 and 6.3. In general, user tasks have no other representation than their BPMN definition. The parameters are added as extension elements of type "activiti:formProperty". The service tasks are all implementations of the JavaDelegates interface provided by Activiti.

When the process is triggered by a user there is a check first if another process instance is currently running. This is required because inter-process synchronization is difficult to implement and out of scope of this thesis. Note that it is only important to check if a process instance is running for the same project. Each project may have one process instance running at the time.

1. User Task: Upload Product Data. The first user task in the process is used to upload the raw tool data that should be added to the EDB and also to set initial parameters for the process that will be used throughout its execution and which need to be known beforehand. Table 6.2 lists all the parameters the user can set in this task.

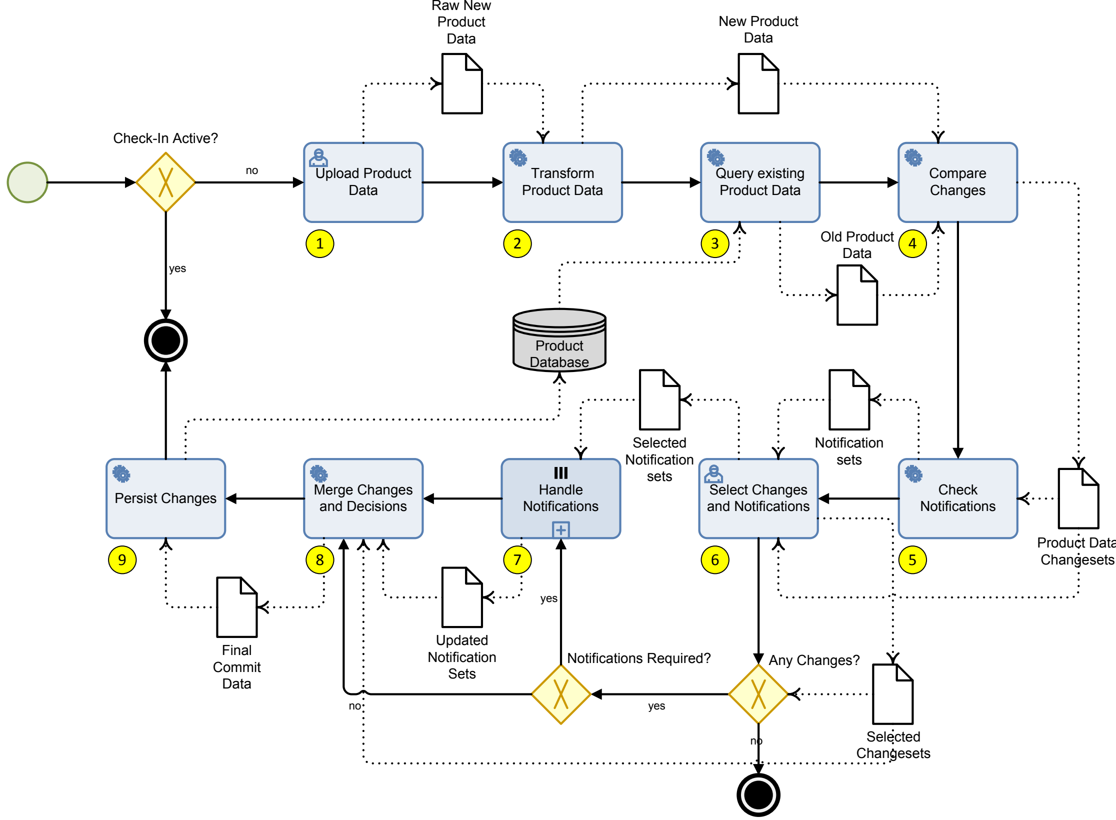

Figure 6.2: BPMN diagram of the implemented process showing the control and data flow. The data flow only shows the flow of signal data and not for all params.

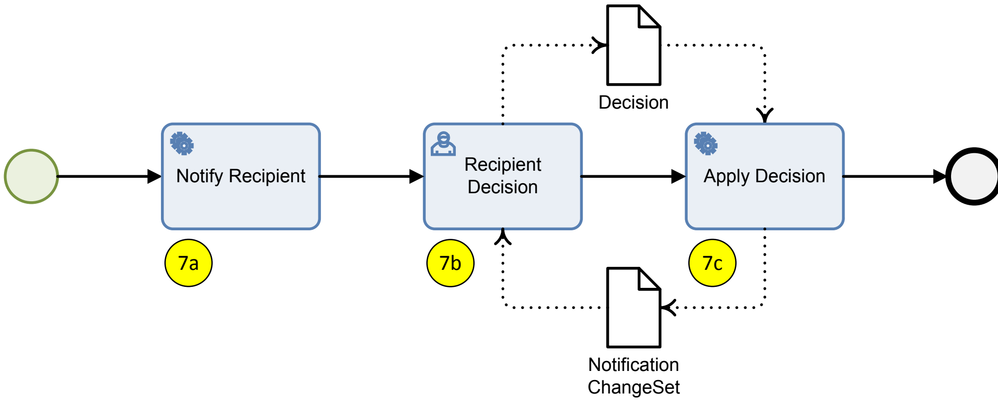

Figure 6.3: The subprocess handling the notifications.

2. Service Task: Transform Product Data. This service task uses the TransformerService to transform the raw product data with the transformer specified in the Upload Product Data task. The result of this transformation is a list of objects of type OpenEngSBModel which is stored as

Name	Description	Required
rawData	The location of the raw tool data to upload.	yes
transformer	The transformer to be used for the transformation of the raw tool data to the internal data model.	yes
tool	The tool from which the raw data originated from.	no
idFields	The list of fields that is used to uniquely identify signals from the raw data and match them with existing data in the EDB.	yes
queryParams	A list of comma separated key-value pairs which can be used to limit the data returned in the ”Query Existing Product Data” task. As per default, all data in the current project will be queried.	no

Table 6.2: The input parameters for the Upload Product Data Task

a new parameter in the VariableScope. Table 6.3 summarizes the input and output parameters of this task.

Name	Description	Type
rawData	The location of the raw tool data to upload.	input
transformer	The transformer to be used for the transformation of the raw tool data to the internal data model.	input
models	The raw tool data transformed to the internal representation.	output

Table 6.3: The parameters for the Transform Product Data Task

3. Service Task: Query Existing Product Data. This service task is used to query the EDB for existing product data. Additionally it also queries and stores the last revision number of the current EDB content. This is required for consistency checks. Table 6.13 summarizes the input and output parameters of this task.

Name	Description	Type
project	The project for which the product data is intended.	input
queryParams	The list of comma separated key-value pairs which can be used to limit the data returned in the ”Query Existing Product Data” task. As per default, all data in the current project will be queried.	input
oldData	The current product data stored in the EDB for this project.	output
revision	The current revision number of the EDB at the time of the query execution.	output

Table 6.4: The parameters for the Query Existing Product Data Task

4. Service Task: Compare Changes. This service task uses the AnalyserService to compare the newly uploaded product data with the old data queried in the previous task. The AnalyzerService tries to match each existing signal with the new signals by their idFields. There are four possible outcomes for each signal: it is classified as "added' if it could not be matched by its idFields with any of the existing signals, it is classified as "updated" if a match of idFields is found but at least one of the other field values differ, it is classified as "unchanged" if a match of idFields is found and none of the other fields value differ, and existing signals are classified as "deleted" if they cannot be matched with the newly uploaded product data. Table 6.5 summarizes the input and output parameters of this task.

Name	Description	Type
models	The raw tool data transformed to the internal representation.	input
oldData	The current product data stored in the EDB for this project.	input
idFields	The list of fields that is used to uniquely identify signals from the raw data and match them with existing data in the EDB.	input
addedData	The models that could not be matched with existing ones in oldData.	output
updatedData	The models that could be matched with existing ones in oldData but were changed.	output
unchangedData	The models that could be matched with existing ones in oldData and were not changed.	output
deletedData	The remaining signals in oldData that could not be matched with signals in models.	output

Table 6.5: The parameters for the Compare Changes Task

5. Service Task: Check Notifications. In this task the new, changed and deleted signals are further analyzed using the NotificationService and it is determined which signals require a notification of another stakeholder. For more information about the notification rules we refer to the requirements chapter where we have described them in detail. Table 6.6 summarizes the input and output parameters of this task.

Name	Description	Type
addedData	The list of signals that were classified as newly added.	input
updatedData	The list of signals that were classified as updated.	input
deletedData	The list of signals that were classified as deleted.	input
notifications	The RecipientsChangeSetsMap mapping Recipients to their respective NotificationChangeSets from which they should approve or decline changes.	output

Table 6.6: The parameters for the Check Notifications Task

6. User Task: Select Changes and Notifications. This user task presents the user who has started the process the results of the ”Compare changes” and ”Check Notifications” tasks and he can decide which changes he would actually like to add and who should be notified. This is useful because in some cases he might have already talked with the other stakeholders about the changes and they could have given their O.K. beforehand. Table 6.7 summarizes the input parameters of this task. Per default all changes determined by the services will be selected. Therefore it is not mandatory to set the parameters. It may seem counter-intuitive and bad design to use negated variables names (e.g. ”unSelectAdded” instead of ”selectAdded”) but we are assuming (based on past experiences with the customer) that in most cases all preselected changes will be selected by the user. From a testing point of view it is also more convenient, because whenever a large amount of data is used, we would have to explicitly select every single signal, which in some cases can amount to hundreds or thousands and in extreme cases even tens of thousands or even hundreds of thousands.

Name	Description	Required
unSelectAdded	The list of indexes for the elements in ”addedData” which should not be added. As per default no added signals will be unselected.	no
unSelectUpdated	The list of indexes for the elements in ”updatedData” which should not be updated. As per default no updated signals will be unselected.	no
unSelectDeleted	The list of indexes for the elements in ”deletedData” which should not be deleted. As per default no deleted signals will be unselected.	no
unSelectRecipients	The list of indexes of recipients which should not be notified. As per default no recipients will be unselected.	no

Table 6.7: The input parameters for the Select Changes and Notifications Task

7. Subprocess: Handle Notifications. If the initiator of the process has selected some notifications to send, the notifications are created and the decisions of the other stakeholders are collected. For each recipient in the list of notifications a subprocess is started. The processes are executed in parallel and when all of them have finished, the main process continues. Table 6.8 summarizes the input and output parameters of this task.

Name	Description	Type
notifications	The RecipientsChangeSetsMap mapping Recipients to their respective NotificationChangeSets from which they should approve or decline changes.	input
recipient	The local variable containing the recipient of the subprocess.	local output

Table 6.8: The parameters for the Handle Notifications Subprocess

7a. Service Task: Notify Recipient. This task can be used to notify the recipients in various ways. E.g. by sending an email containing a link to a web application where the recipients can perform their decisions. This task is currently not implemented because it is not required for our evaluation. In our evaluation users are notified by the process engine directly.

7b. User Task: Recipient Decision. This is the user task where the recipient can decide whether they accept or decline the signals contained in the notification. Table 6.9 summarizes the input and output parameters of this task. Per default all changes in the notification will be selected. Therefore it is not mandatory to set the parameters.

Name	Description	required
declineAdded	The list of indexes for the elements in the notification which should not be added.	no
declineUpdated	The list of indexes for the elements in the notification which should not be updated.	no
declineDeleted	The list of indexes for the elements in the notification which should not be deleted.	no

Table 6.9: The input parameters for the Decide Task

7c. Service Task: Apply Decision. This service task is required to apply the decisions done by the notified stakeholder to the process data model. This is done by adding the decision information to the NotificationChangeSets which were stored in the global process variable ”notifications”. Table 6.10 summarizes the input and output parameters of this task.

Name	Description	Type
recipient	The local variable containing the recipient of the subprocess.	local input
declineAdded	The list of indexes for the elements in the notification which should not be added.	local input
declineUpdated	The list of indexes for the elements in the notification which should not be updated.	local input
declineDeleted	The list of indexes for the elements in the notification which should not be deleted.	local input
notifications	The RecipientsChangeSetsMap mapping Recipients to their respective NotificationChangeSets from which they should approve or decline changes.	output

Table 6.10: The parameters for the Apply Decision Task

8. Service Task: Merge Changes and Decisions. After all stakeholders of the process have performed their selections and decisions the results are merged by removing all unselected and

declined signals from the lists that are going to be persisted. Table 6.11 summarizes the input and output parameters of this task.

Name	Description	Type
addedData	The list of signals that were classified as newly added.	input/output
updatedData	The list of signals that were classified as updated.	input/output
deletedData	The list of signals that were classified as deleted.	input/output
notifications	The RecipientsChangeSetsMap mapping Recipients to their respective NotificationChangeSets from which they should approve or decline changes.	output
amountOfAdded	The amount of signals that were classified as newly added.	output
amountOfUpdated	The amount of signals that were classified as updated.	output
amountOfDeleted	The amount of signals that were classified as deleted.	output
amountOfUnchanged	The amount of signals that were classified as unchanged. (relative to the uploaded signals, not relative to the amount of signals in the product database)	output

Table 6.11: The parameters for the Merge Task

9. Service Task: Persist Product Data. The final task persists the product data in the product database by creating an EDBCommit containing all added, updated and deleted signals.

Name	Description	Type
project	The name of the project that the commit is created for.	input
revision	The UUID of the current top level commit in the project. This is required to verify that the product data in the database does not change during the Check-In process.	input
addedData	The list of signals that were classified as newly added.	input
updatedData	The list of signals that were classified as updated.	input
deletedData	The list of signals that were classified as deleted.	input
commitId	The UUID produced by the successful commit.	output

Table 6.12: The parameters for the Persist Task

6.3.2 Update Status Process

While status updates could technically also be handled with the Check-In process, they are different from normal signal updates in various ways: they are rather process information than product information, they never require notifications, they are always executed on existing product data and they never cause merge conflicts. As a result running the full Check-In process with all its features for status updates would cause a significant overhead in both system run-

time and user decision time. Additionally, since the status in the signal is stored just like actual product data fields, changes to the status cannot be easily distinguished from product changes on a database level, where they both are treated as 'update' operations. Thus, separating them during analysis (e.g. when analyzing all product changes without status changes) would require very complicated and most likely also badly performing queries where different versions of signals would have to be compared for status changes. To tackle this we could simply introduce a new operation just for status updates to the database level, however this would be a very bad design decision on multiple levels. Therefore we have defined a separate process for updating signal states, as seen in figure 6.4, that reduces user decision and process run times while it simultaneously helps to distinguish between pure product updates and status updates for analysis purposes.

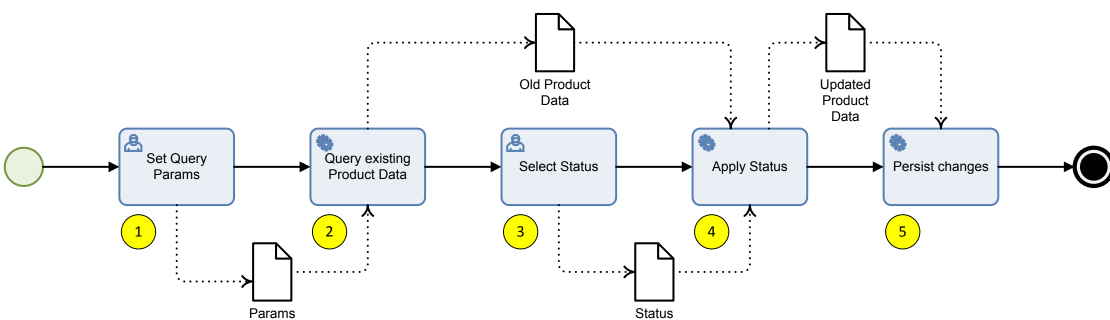

Figure 6.4: The process for status updates.

1. User Task: Set Query Parameters. The user sets the parameters for the query which allows him to limit the amount of signals to work with. The current project is always a default parameter. Table 6.13 summarizes the input and output parameters of this task.

Name	Description	required
queryParams	A list of comma separated key-value pairs which can be used to limit the data returned in the "Query Existing Product Data" task. As per default, all data in the current project will be queried.	no

Table 6.13: The input parameters for the Set Query Parameters Task

2. Service Task: Query Existing Product Data. This task is identical to task no. 3 of the Check-In process and therefore we refer to the detailed explanation there. 6.3.1

3. User Task: Select Status. The user selects the status which should be applied to the selected signals. This task will also be used to present the user the result of the query to allow him to double-check his selection. Table 6.14 summarizes the input parameters of this task.

Name	Description	required
status	The status that should be set for all signals selected by the query.	yes

Table 6.14: The input parameters for the Select Status Task

4. Service Task: Apply Status Task. The status is applied to the selected signals in preparation for the pending commit. Table 6.15 summarizes the input and output parameters of this task.

Name	Description	Type
oldData	The list of signals to set the status for.	input
status	The status to set for the selected signals.	input
updatedData	The list of signals with set status.	output
amountOfAdded	The amount of signals that were newly added. (always 0 for this process)	output
amountOfUpdated	The amount of signals that were updated.	output
amountOfDeleted	The amount of signals that were deleted. (always 0 for this process)	output
amountOfUnchanged	The amount of signals that were classified as unchanged. (always 0 for this process)	output

Table 6.15: The parameters for the Apply Status Task

5. Service Task: Persist Product Data. This task is identical to task no. 9 of the Check-In process and therefore we refer to the detailed explanation there. 6.3.1

6.4 Process implementations

After having discussed the process definitions in detail we will also have a brief look at the actual process implementation and other important components that were implemented. Of course we cannot give full code listings of all parts of the entire solution here, but instead will have a look at selected parts.

6.4.1 XML Process Definitions

The Activiti process engine uses the BPMN XML process definition as primary input. Each process definition contains only a single top level process (but may include sub-processes).

Check-In Process. Listing 6.1 shows the process definition of the check-in process that we have implemented. As can be seen it is still relatively compact (less than 100 lines of code). Usually such a file could also contain additional location information for visual BPMN editors in a separate XML element, but this is irrelevant for the process engine. Since we have already

described the service and user tasks in detail we will limit our explanation only to interesting parts. All elements are in chronological order. Processes start with element ”startEvent” and end with ”endEvent”. SequenceFlows mark the transitions between tasks, gateways and events. If they have a gateway as sourceRef, sequenceFlows can have a conditionExpression containing the condition when the transition is triggered. It is possible to access variables from the VariableScope here (e.g. line 41). User tasks may have form properties as extensionElements which can be seen in lines 14 - 17. These allow users to pass input to the process. The parameters are stored into the VariableScope of the process instance where they can be accessed by the value specified in the ”id” attribute. FormProperties have a specific datatype (”type”) and can required or optional (”required”). Service tasks call a delegate java bean (specified in the attribute ”activiti:delegateExpression”) where we can execute actual java code. These beans are defined in blueprint and may inject other OSGi services and beans. All service tasks are implemented this way.

Listing 6.1: The ”Check-In” process implemented with Activiti.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <definitions id="definitions"
4               xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
5               xmlns:activiti="http://activiti.org/bpmn"
6               targetNamespace="Examples" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7               xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL http://www.omg.org
                    /spec/BPMN/2.0/20100501/BPMN20.xsd">
8    <!-- begin of root process -->
9    <process id="checkinProcess" name="Activiti Checkin Process">
10     <startEvent id="start" activiti:initiator="initiator"/>
11     <sequenceFlow sourceRef="start" targetRef="uploadTask"/>
12     <userTask id="uploadTask" activiti:assignee="${initiator}">
13       <extensionElements>
14         <activiti:formProperty id="rawData" type="string" required="true"/>
15         <activiti:formProperty id="transformer" type="string" required="true"/>
16         <activiti:formProperty id="tool" type="string" required="true"/>
17         <activiti:formProperty id="idFields" type="string" required="true"/>
18       </extensionElements>
19     </userTask>
20     <sequenceFlow sourceRef="uploadTask" targetRef="transformTask"/>
21     <serviceTask id="transformTask" activiti:delegateExpression="${transformTask}"/>
22     <sequenceFlow sourceRef="transformTask" targetRef="queryTask"/>
23     <serviceTask id="queryTask" activiti:delegateExpression="${queryTask}"/>
24     <sequenceFlow sourceRef="queryTask" targetRef="compareTask"/>
25     <serviceTask id="compareTask" activiti:delegateExpression="${compareTask}"/>
26     <sequenceFlow sourceRef="compareTask" targetRef="checkNotificationsTask"/>
27     <serviceTask id="checkNotificationsTask" activiti:delegateExpression="${
           checkNotificationsTask}"/>
28     <sequenceFlow sourceRef="checkNotificationsTask" targetRef="selectTask"/>
29     <userTask id="selectTask" activiti:assignee="${initiator}">
30       <extensionElements>
31         <activiti:formProperty id="unSelectNew" type="string" required="false"/>
32         <activiti:formProperty id="unSelectUpdated" type="string" required="false"/>
33         <activiti:formProperty id="unSelectDeleted" type="string" required="false"/>
34         <activiti:formProperty id="unSelectNotifications" type="string" required="false"/>
35       </extensionElements>
36     </userTask>
37     <sequenceFlow sourceRef="selectTask" targetRef="madeChanges"/>
38     <!-- first gateway checking for selected changes -->
39     <exclusiveGateway id="madeChanges"/>
40     <sequenceFlow sourceRef="madeChanges" targetRef="needNotifications">
41       <conditionExpression>
42         ${newData.size() > 0 || deletedData.size() > 0 || updatedData.size() > 0}
43       </conditionExpression>
44     </sequenceFlow>
45     <sequenceFlow sourceRef="madeChanges" targetRef="end">
46       <conditionExpression>
```

```
47          ${newData.size() == 0 && deletedData.size() == 0 && updatedData.size() == 0}
48        </conditionExpression>
49      </sequenceFlow>
50      <!-- second gateway checking for pending notifications -->
51      <exclusiveGateway id="needNotifications"/>
52      <sequenceFlow sourceRef="needNotifications" targetRef="notifySubprocess">
53        <conditionExpression>
54          ${notifications.size() > 0}
55        </conditionExpression>
56      </sequenceFlow>
57      <sequenceFlow sourceRef="needNotifications" targetRef="persistTask">
58        <conditionExpression>
59          ${notifications.size() == 0}
60        </conditionExpression>
61      </sequenceFlow>
62      <!-- notification subprocess starts here -->
63      <subProcess>
64      <!-- see subprocess listing -->
65      </subProcess>
66      <!-- resume root process -->
67      <sequenceFlow sourceRef="notifySubprocess" targetRef="persistTask"/>
68      <serviceTask id="persistTask" activiti:delegateExpression="${persistTask}"/>
69      <sequenceFlow sourceRef="persistTask" targetRef="end"/>
70      <endEvent id="end"/>
71    </process>
72  </definitions>
```

Notification Subprocess. The notification subprocess is defined in lines 63 and 64 and in detail in listing 6.2. It looks just like an ordinary process except for the element ”multiInstanceLoopCharacteristics”. This element is required to execute the process multiple times without having to explicitly model the execution. For each element in the specified ”activiti:collection” a process is launched. In this case the map ”notifications” from the variable scope which maps each recipient to his respective changeset. It is also possible to call functions on these variables, in this case the we want to iterate over each recipient and therefore have to call ”keySet()” on the map. The current element of the iteration is available in the local variable scope of the subprocess in the field specified in the attribute ”activiti:elementVariable”. Finally the attribute ”isSequential” determines whether sub processes are launched sequential or parallel.

Listing 6.2: The ”Handle Notifications” subprocess implemented with Activiti.

```
1  <subProcess id="notifySubprocess">
2    <multiInstanceLoopCharacteristics isSequential="false"
3                                      activiti:collection="${notifications.keySet()}"
4                                      activiti:elementVariable="recipient">
5    </multiInstanceLoopCharacteristics>
6    <startEvent id="notifyStart"/>
7    <sequenceFlow sourceRef="notifyStart" targetRef="notifyTask"/>
8    <serviceTask id="notifyTask" activiti:delegateExpression="${notifyTask}" />
9    <sequenceFlow sourceRef="notifyTask" targetRef="decideTask"/>
10   <userTask id="decideTask" activiti:assignee="${initiator}">
11     <extensionElements>
12       <activiti:formProperty id="declineNew" type="string" required="false"/>
13       <activiti:formProperty id="declineUpdated" type="string" required="false"/>
14       <activiti:formProperty id="declineDeleted" type="string" required="false"/>
15     </extensionElements>
16   </userTask>
17   <sequenceFlow sourceRef="decideTask" targetRef="applyDecisionTask"/>
18   <serviceTask id="applyDecisionTask" activiti:delegateExpression="${applyDecisionTask}"/>
19   <sequenceFlow sourceRef="applyDecisionTask" targetRef="notifyEnd"/>
20   <endEvent id="notifyEnd"/>
21 </subProcess>
```

Update Status Process. The last process definition is the "Update Status Process" which can be seen in listing 6.3. This is a simple and straightforward process and quite unspectacular in comparison to the other processes and contains no special constructs that require explanation.

Listing 6.3: The update status process implemented with Activiti.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <definitions id="definitions"
4               xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
5               xmlns:activiti="http://activiti.org/bpmn"
6               targetNamespace="Examples" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7               xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL http://www.omg.org
                    /spec/BPMN/2.0/20100501/BPMN20.xsd">
8
9    <process id="statusProcess" name="Activiti Status Process">
10     <startEvent id="start" activiti:initiator="initiator"/>
11     <sequenceFlow sourceRef="start" targetRef="setQueryParamsTask"/>
12     <userTask id="setQueryParamsTask" activiti:initiator="${initiator}">
13       <extensionElements>
14         <activiti:formProperty id="queryParams" type="string" required="false"/>
15       </extensionElements>
16     </userTask>
17     <sequenceFlow sourceRef="setQueryParamsTask" targetRef="queryTask"/>
18     <serviceTask id="queryTask" activiti:delegateExpression="${queryTask}"/>
19     <sequenceFlow sourceRef="queryTask" targetRef="selectStatusTask"/>
20     <userTask id="selectStatusTask" activiti:initiator="${initiator}">
21       <extensionElements>
22         <activiti:formProperty id="status" type="string" required="true"/>
23       </extensionElements>
24     </userTask>
25     <sequenceFlow sourceRef="selectStatusTask" targetRef="applyStatusTask"/>
26     <serviceTask id="applyStatusTask" activiti:delegateExpression="${applyStatusTask}"/>
27     <sequenceFlow sourceRef="applyStatusTask" targetRef="persistTask"/>
28     <serviceTask id="persistTask" activiti:delegateExpression="${persistTask}"/>
29     <sequenceFlow sourceRef="persistTask" targetRef="end"/>
30     <endEvent id="end"/>
31   </process>
32 </definitions>
```

6.4.2 Notification Services

The NotificationMapper is the new component that helps with determining which product changes require a notification and which stakeholder have to receive a notification and subsequently make a decision about these changes. Figure 6.5 depicts the class diagram of the involved classes that we have created to solve this problem. We will further split the more detailed explanation of this diagram into two parts, application and configuration:

Configuration The configuration of the NotificationMapper can be done either via configuration files by using the JSON format or programmatically by using the setters. There are four main variables which are used for the definition of recipients, status, rules and the mapping between all of these. The NotificationMapperFactory is used to produce usable and preconfigured instances of NotificationMappers. Our implementation reads the configuration files and uses the JacksonMapperUtil to convert them to Java objects which are then used to configure the NotificationMapper instance. Listings 6.4, 6.5, 6.6 and 6.7 show the configuration files containing the configuration described in the requirements chapter.

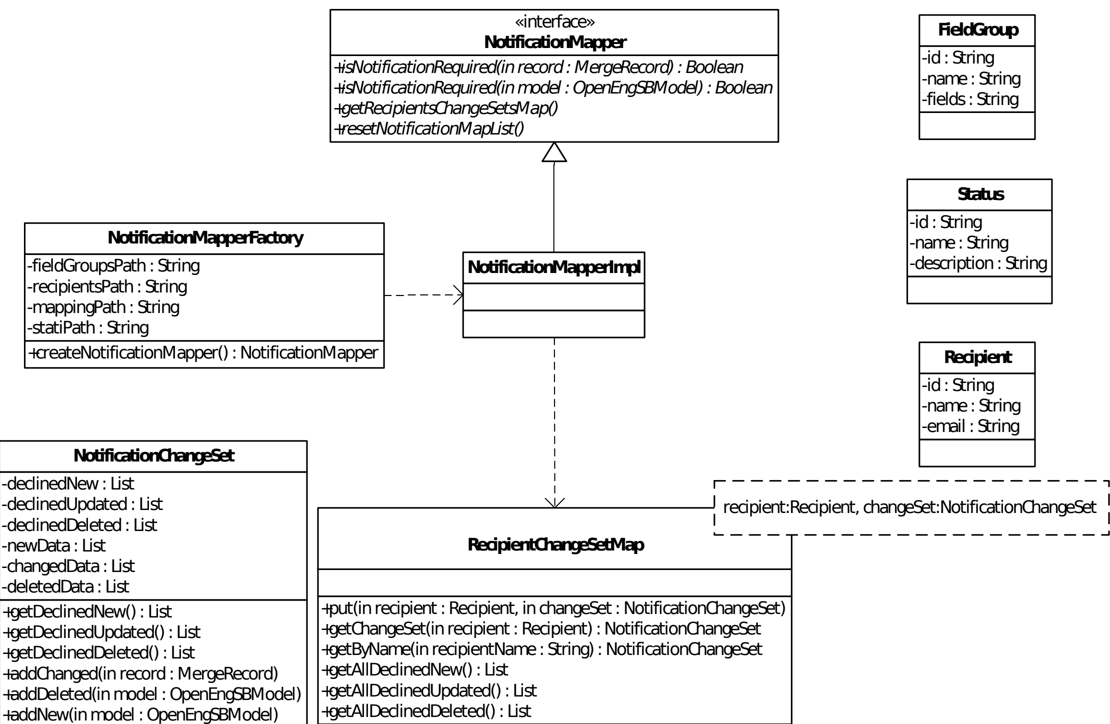

Figure 6.5: The class diagram of the NotificationMapper and associated classes.

Each state shown in listing 6.4 contains an id which is used by the NotificationMapper, a name that is used by tools to identify a state and an optional description. The redundant object names (e.g. ”1” in line 2) which are equal to the ids is required due to technical reasons of the used JSON parser which creates a map which map the objects to their ids.

Listing 6.4: The configuration of states.

```
1  {
2    "1": {
3      "id": "1",
4      "name": "St1",
5      "description": "Signal from sub-supplier"
6    },
7    "2": {
8      "id": "2",
9      "name": "St2",
10     "description": "Signal from customer"
11   },
12   "3": {
13     "id": "3",
14     "name": "St3",
15     "description": "Approved KKS number and description from customer"
16   },
17   "4": {
18     "id": "4",
19     "name": "St4",
20     "description": "OPM hardware/software I/O addresses assigned"
21   },
22   "5": {
23     "id": "5",
24     "name": "St5",
```

```
25      "description": "Electrical Engineering (Eplan) started"
26    },
27    "6": {
28      "id": "6",
29      "name": "St6",
30      "description": "Manufacturing of hardware units"
31    },
32    "7": {
33      "id": "7",
34      "name": "St7",
35      "description": "Manufacturing finished and approved"
36    },
37    "8": {
38      "id": "8",
39      "name": "St8",
40      "description": "After commissioning"
41    }
42  }
```

The definition of recipients 6.5 is quite similar, with an internal id, a name and an email address in case a notification is sent via email.

Listing 6.5: The configuration of recipients.

```
1  {
2    "customer": {
3      "id": "customer",
4      "name": "customer",
5      "email": "michael.petritsch@gmail.com"
6    },
7    "process_engineer": {
8      "id": "process_engineer",
9      "name": "process engineer (OPM)",
10     "email": "michael.petritsch@gmail.com"
11   },
12   "electrical_engineer": {
13     "id": "electrical_engineer",
14     "name": "electrical engineer (Eplan)",
15     "email": "michael.petritsch@gmail.com"
16   },
17   "manufacturer": {
18     "id": "manufacturer",
19     "name": "manufacturer (wiring)",
20     "email": "michael.petritsch@gmail.com"
21   }
22 }
```

The defintion of the field rules 6.6 is also very similar, again with an id and a name, but this time there is the list of fields attached, which contains the fields that have to be checked for changes in this rule.

Listing 6.6: The configuration of field group rules.

```
1  {
2    "kks": {
3      "id": "kks",
4      "name": "KKS-Number",
5      "fields": [ "kks0", "kks1", "kks2", "kks3" ]
6    },
7    "desc": {
8      "id": "desc",
9      "name": "Signal description",
10     "fields": [ "longText" ]
11   },
12   "chan": {
13     "id": "chan",
14     "name": "I/O channel",
```

```
15      "fields": [ "channelName" ]
16    }
17  }
```

Finally, the most complicated but still compact definition of the mapping between, states, recipients and rules. The first object name (e.g. ”6” in line 16) corresponds to the id of a state, the contents of this object (lines 17 - 19) list the rules which should be checked for this state and the arrays for each of the rules contain the recipients that should be notified if the outcome of the check is positive.

Listing 6.7: The configuration of the final mapping.

```
1  {
2    "3": {
3      "kks": [ "customer" ],
4      "desc": [ "customer" ]
5    },
6    "4": {
7      "kks": [ "process_engineer" ],
8      "desc": [ "process_engineer" ],
9      "chan": [ "process_engineer" ]
10   },
11   "5": {
12     "kks": [ "electrical_engineer" ],
13     "desc": [ "electrical_engineer" ],
14     "chan": [ "electrical_engineer" ]
15   },
16   "6": {
17     "kks": [ "electrical_engineer" ],
18     "desc": [ "electrical_engineer" ],
19     "chan": [ "electrical_engineer", "manufacturer" ]
20   },
21   "7": {
22     "kks": [ "electrical_engineer" ],
23     "desc": [ "electrical_engineer" ],
24     "chan": [ "electrical_engineer" ]
25   },
26   "8": {
27     "kks": [ "electrical_engineer" ],
28     "desc": [ "electrical_engineer" ],
29     "chan": [ "electrical_engineer" ]
30   }
31 }
```

Application The main classes for the application of the mapper are classes implementing the NotificationMapper interface, which not only contain all the information required to perform the checks on signals, but also help with collecting all the changed signals which require notification. The ”isNotificationRequired” methods check an object for changes and if a notification is required, the object is added internally to the RecipientChangeSetMap which maps Recipients to their NotificationChangeSets. This map can be obtained via the method ”getRecipientChangeSetMap”. This map can be cleared by calling the method ”resetRecipientChangeSetMap”. The RecipientChangeSetMap offers some adapted standard methods for maps for data storage/retrieval. On top of that it offers three specialized methods ”getAllDeclinedNew”, ”getAllDeclinedUpdated” and ”getAllDeclinedDeleted”. These methods return the lists containing new/updated/deleted signals from the NotificationChangeSet with respect to the user decision. The user decision is also stored in the NotificationChangeSet as lists ”declinedNew”, ”declinedUpdated” and ”declinedDeleted” containing array indices for the other three lists.

6.5 Process testing

Testing the implemented processes is also very important. In this work we have focused on integration testing over unit or acceptance tests. On one hand we could not do full acceptance tests because that would require to implement a user interface which is out of scope of this work. On the other hand unit tests for the whole process are costly to implement (from an effort perspective) as we are using many different services which would all require mocking. For the KPI analysis following in the evaluation chapter, unit tests - unlike acceptance tests - would not offer any benefits either without extreme work overhead. Mocking services for such scenarios in a meaningful way would come close to re-implementing them, if not worse. Therefore the optimal middle ground for this scope are integration tests. In this way we can test various interesting and complex scenarios by abstracting user input to simple parameters while at the same time testing the full process and service implementation.

6.5.1 Test Model

To test all kinds of process execution scenarios we need a way to parameterize our tests. The various parameters for a process in our case are the inputs of users in user tasks. We can do this by using Activiti's FormService to pass the parameters to the process. For the test the parameters are put into a json file. The file is parsed into a simple java class hierarchy as shown in Figure 6.6 which is then used as input data of an integration test.

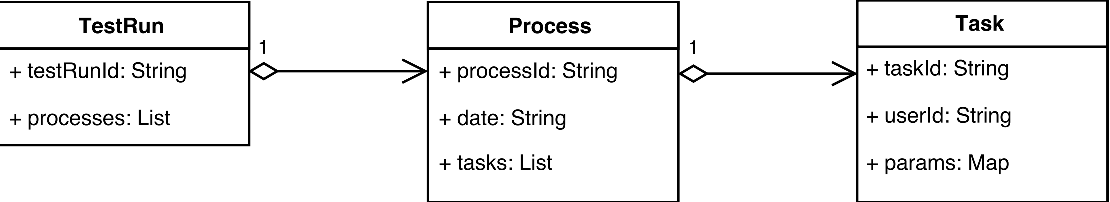

Figure 6.6: The test model for parameterized testing.

The TestRun Class. The TestRun class contains all data required for the complete test run. In this case we provide an ID (though it is not used) and an ordered list of processes that should be executed during this run. Additional parameters may be added to the model as needed, for example it may be useful to provide individual asserts which should be checked before or after the test run execution.

The Process Class. The Process class contains all data required for the execution of a single process. As in the case of the TestRun, we have an ID, but this time the ID is required, because it represents the ID of the process definition (e.g. ”checkinProcess” or ”statusUpdateProcess”). The next parameter is a date difference. In our implementation only a single integer ”x” is supported which represents a month difference. This allows us to change the current system time by ”x” months. The final parameter is an ordered list of user Tasks which are expected during the execution of the process.

The Task Class. The Task class contains all data required for the execution of a single user task. The first parameter ”ID” refers to the userTask id used in the process definition. The second parameter ”params” is a map containing all the parameters provided by the user. The keys provided have to be equal to the formProperty ids in the process definition.

6.5.2 Test Parameters File

Listing 6.8 shows a sample file in JSON format from which a test model can be constructed, containing the minimum amount of test parameters required for our check-in process. It shows a test case with a single process execution which only has two user tasks. Line 2 is the testRunId which can be chosen freely by the tester (this is useful when using multiple parameter files in a single test), Lines 3 - 23 are the array containing all processes. Lines 4 - 22 is the sole process we want to execute with its processId (line 5) which has to match the process id specified in the BPMN file. The array containing all the user tasks which should be created by the process can be seen in lines 6 - 21. Lines 7 - 15 are the first user task in the process, the uploadTask. Again the taskId in line 8 has to match the task id specified in the BPMN. Lines 9 - 14 list the parameters set by the user in this task. In this case they are ”rawData’ which refers to the file containing the data directly exported from the tool, ”transformer” which refers to the transformer that can parse the data and transform it to the internal data model, ”tool” which is the name of the tool from which the raw data originated and ”idFields” which is a list of fields (from the internal model) that are used to uniquely identify a signal. The second and final task created by this process is the ”selectTask”. In this instance no parameters are provided by the user, meaning that the default parameters provided by the process are used.

Listing 6.8: A parameter file containing test parameters.

```
1  {
2    "testRunId": 1,
3    "processes": [
4      {
5        "processId": "checkinProcess",
6        "date": "0",
7        "tasks": [
8          {
9            "taskId": "uploadTask",
10           "userId": "process_engineer",
11           "params": {
12             "rawData": "transformers/samples/DA-OPM.csv",
13             "transformer": "transformers/DA-OPM.zip",
14             "tool": "OPM",
15             "idFields": "region, componentNumber, cpuNumber, rackId",
16             "queryParams": ""
17           }
18         },
19         {
20           "taskId": "selectTask",
21           "userId": "process_engineer",
22           "params": {
23           }
24         }
25       ]
26     }
27   ]
28 }
```

Restrictions Note that all parameters are of type String. This is because the FormService only allows strings as parameters. Also it is often required to pass indices of collections as parameters to simulate a user selecting certain items from a collection. While this is not a problem in the rolled out application (because the user sees what he selects and the indices match the selection), it can have unexpected consequences in our test scenarios when the indices refer to unsorted lists. Therefore it is important to use sorted collections.

6.5.3 Test Class and Methods

Since the OpenEngSB is based on Apache Karaf, the standard way to write integration tests is by using pax exam [3]. This framework allows to launch Karaf runtimes inside a JUnit test class and implement test cases which can access OSGi services running in the OpenEngSB and also from additional bundles which can be added during the test if needed. Listing 6.9 shows the simplified test class that we have used for all our tests. The class extends AbstractExamTest (not shown), which allows to specify which OpenEngSB implementation to use as well as other general test parameters like logging and debugging configuration. At the head of the class we tell JUnit to use the PaxExam testrunner and set the ExamReactorStrategy to ”PerClass” which means that the OpenEngSB will only be started once for this test class. In lines 5 - 12 the OSGi services that we want to use are injected into the test class. In this case we only need three Activiti services which are (6) the RuntimeService to start processes, (9) the TaskService to retrieve pending user tasks and (12) the FormService to pass user input to the user tasks and complete them. Lines 14 - 17 is a concrete test. Since we are testing parameterized (see explanation below) we simply specify the file containing the test parameters and pass it to the method ”runTest” 6.10 which is responsible for executing the test and extracting the parameters.

Listing 6.9: The test class used for integration testing of activiti processes.

```
1  @RunWith(PaxExam.class)
2  @ExamReactorStrategy(PerClass.class)
3  public class CheckinProcessIT extends AbstractExamTest {
4
5      @Inject
6      private RuntimeService runtimeService;
7
8      @Inject
9      private TaskService taskService;
10
11     @Inject
12     private FormService formService;
13
14     @Test
15     public void testMinimalisticCheckin() throws Exception {
16         runTest(new File("src/test/resources/MinimalisticCheckin.json"));
17     }
18 }
```

The ”runTest” method 6.10 first has to disable all timeouts that are set in various components of the OpenEngSB (line 2). This is required because the system date may change between process execution (to realistically simulate long running processes or activities). Listing 6.10 shows the implementation of the ”disableTimeouts” method. It is important that all components provide means to disable the timeouts for this to work. While it is possible to disable the timeouts

[3] Pax Exam: https://ops4j1.jira.com/wiki/display/PAXEXAM4/Pax+Exam

in configuration files, we prefer to do this in the test, because we want to use the same configuration as in the final product that is rolled out to the customer. In our OpenEngSB configuration it is sufficient to disable the session timout of the current user by setting it to ”-1”.

Listing 6.10: The runTest method responsible for test execution.

```
1  private void runTest(File testRunFile) throws Exception {
2      disableTimeouts();
3
4      TestRun testRun = JacksonMapperUtil.readTestRun(testRunFile);
5
6      for (org.cdlflex.activiti.checkin.model.test.Process process : testRun.getProcesses()) {
7          if(process.getDate() != null && !process.getDate().isEmpty()) {
8              int monthChange = Integer.valueOf(process.getDate());
9              changeDate(monthChange);
10         }
11
12         runtimeService.startProcessInstanceByKey(process.getId());
13
14         for (org.cdlflex.activiti.checkin.model.test.Task testTask : process.getTasks()) {
15             List<Task> userTasks = taskService.createTaskQuery().active().list();
16             assertNotNull(userTasks);
17             for (Task userTask : userTasks) {
18                 taskService.claim(userTask.getId(), testTask.getUserId());
19                 List<FormProperty> formList = formService.getTaskFormData(userTask.getId()).
                       getFormProperties();
20                 assertEquals(testTask.getParams().size(), formList.size());
21                 formService.submitTaskFormData(userTask.getId(), testTask.getParams());
22                 break;
23             }
24         }
25     }
26 }
```

Listing 6.11: The method to disable timeouts to prevent timeouts caused by date changes.

```
1  private void disableTimouts() {
2      Subject subject = SecurityUtils.getSubject();
3      Session session = subject.getSession();
4      session.setTimeout(-1);
5  }
```

We continue with our examination of the ”runTest” method and in line 4 we can see the extraction of the test model which we have discussed earlier from the test file provided by the ”testRunFile” parameter. In line 6 the loop over the processes is started. The first step is the examination of the ”date” parameter. If it is set in the configuration file, the ”changeDate” method is called which can be seen in listing 6.12. This method creates a Calendar model from the current Date (lines 2 and 3) and then adds the ”monthChange” number to the calendar. This changes the Calendar date by ”monthChange” months. Finally, it constructs a linux date command in the format ”month/day/year hour:minute:second” (the complete command line can be seen in line 12) and executes it with ”Runtime.getRuntime().exec()”. The command ”sudo” is required, to execute the command as root user, which is required to change the system date in linux. Obviously, this solution only works in linux which is sufficient for the scope of this work. Note that this is a potential security risk and only used for scientific testing purposes and evaluations. It may only be used in a secured testing environment and is not intended for use in a production environment.

Listing 6.12: The method to change the date in linux.

```
1  private void changeDate(int monthChange) throws IOException {
2      Calendar cal = Calendar.getInstance();
3      cal.setTime(new Date());
4      cal.add(Calendar.MONTH, monthChange + 1);
5      String month = String.format("%02d", cal.get(Calendar.MONTH));
6      String day = String.format("%02d", cal.get(Calendar.DAY_OF_MONTH));
7      String year = String.format("%04d", cal.get(Calendar.YEAR));
8      String hour = String.format("%02d", cal.get(Calendar.HOUR_OF_DAY));
9      String minute = String.format("%02d", cal.get(Calendar.MINUTE));
10     String second = String.format("%02d", cal.get(Calendar.SECOND));
11     String date = month + "/" + day + "/" + year + " " + hour + ":" + minute + ":" + second;
12     // sudo date -s MM/dd/yyyy hh:mm:ss
13     Runtime.getRuntime().exec(new String[]{"sudo", "date", "-s", date});
14 }
```

After the date has been set, the process is executed on the process engine in line 12 of listing 6.10 with the ”runtimeService.startProcessInstanceByKey()” and the process id as parameter. The process engine automatically executes all non-user activities and gateways as specified in the process definition and stops the process execution when it hits an end event or user task. When it stops at an end event, the process is finished and the next process (if defined in the test parameter file) is executed, otherwise the test run is completed. If it stops at a user task, it requires some user input. The command ”taskService.createTaskQuery().active().list()” in line 15 returns a list with all open tasks waiting for user input. In line 17 we are iterating over these tasks and provide then with the parameters from the test parameters file, by using the ”formService.submitTaskFormData()” method in line 20. After this method is called, the process execution continues again until it hits a user task or end event.

6.6 Process and Product Analysis

For the process and product analysis we will use the SQL database tables provided and managed by Activiti and the Engineering Database Index (EDBI). Activiti is mainly used to record process related information, while the EDBI contains the versioned product data which is also stored in the EDB, but which is slow and difficult to query.

6.6.1 Process Analysis

For the process analysis we are using Activiti’s automated logging feature as already introduced in 6.1. Whenever a process is executed, Activiti logs various process parameters, depending on its configuration. Here we will only introduce the tables relevant for our evaluation, which are *ACT_HI_PROCINST*, *ACT_HI_ACTINST*, *ACT_HI_VARINST*, *ACT_HI_DETAIL* and *ACT_GE_BYTEARRAY* and can be seen in figure 6.7.

ACT_HI_PROCINST	
PK	**ID_**
	PROC_INST_ID_ BUSINESS_KEY_ **PROC_DEF_ID_** **START_TIME_** END_TIME_ DURATION_ START_USER_ID_ START_ACT_ID_ END_ACT_ID_ SUPER_PROCESS_INSTANCE_ID_ DELETE_REASON_ TENANT_ID_ NAME_

ACT_HI_ACTINST	
PK	**ID**
	PROC_DEF_ID_ **PROC_INST_ID_** **EXECUTION_ID_** **ACT_ID_** TASK_ID_ CALL_PROC_INST_ID_ ACT_NAME_ **ACT_TYPE_** ASSIGNEE_ **START_TIME_** END_TIME_ DURATION_ TENANT_ID_

ACT_HI_VARINST	
PK	**ID_**
	PROC_INST_ID_ EXECUTION_ID_ TASK_ID_ **NAME_** VAR_TYPE_ REV_ BYTEARRAY_ID_ DOUBLE_ LONG_ TEXT_ TEXT2_ CREATE_TIME_ LAST_UPDATED_TIME_

ACT_GE_BYTEARRAY	
PK	**ID_**
	REV_ NAME_ DEPLOYMENT_ BYTES_ GENERATED_

Figure 6.7: The EER model of the Activiti history.

6.6.2 Product Analysis

For the product analysis we are using two specific tables created by the EDBI as can be seen in figure 6.8, which are the tables *HEAD_ID* and *HISTORY_ID*. Note that the IDs are randomly generated and uniquely identify a project and a product model.

HEAD	
PK	UUID
	REV_CREATED REV_MODIFIED REGION COMPONENTNUMBER ...

HISTORY	
PK	REV_ID
	REV_COMMIT REV_TIMESTAMP REV_OPERATION REV_USER REV_CONTEXTID REV_DOMAINID REV_CONNECTORID REV_INSTANCEID UUID REGION COMPONENTNUMBER ...

Figure 6.8: The EER model of the EDBI.

Function	Return Value	Origin
avg	the average value	SQL-89
cast	value casted to a given type	H2
count	total amount records in a column	SQL-89
datediff	the difference between two timestamps in the given unit	H2
distinct	list of values without duplicates	SQL-89
formatdatetime	formated timestamp with the given format	H2
max	maximum value	SQL-89
month	the month of a timestamp	H2
round	rounds a numeric value to the number of given decimal places	SQL-89
stddev_pop	the population standard deviation	H2

Table 6.16: The sql standard functions used.

6.6.3 H2 and SQL functions

To acces process and product data and derive meaningful information, we will use functions provided by the used database H2 which can be seen in table 6.16. In part they are SQL-89 standard functions [Date and Darwen, 1997] which are provided by H2 and include the functions *AVG*, *COUNT*, *DISTINCT*, *MAX* and *ROUND*. Additionally H2 specific functions were also used like *DATEDIFF*, *MONTH*, *FORMATDATETIME*, *STDDEV_POP* and *CAST*. A detailed explanation of all the supported functions can be found on the H2 website[4].

[4]H2 Database: http://www.h2database.com/html/functions.html

CHAPTER 7

Evaluation

As required by the research approach, in this chapter we will evaluate the solution against the use-case description and requirements given in chapter 5. We will start with an evaluation of the process implementations by examining their results in artificial test scenarios which trigger all of the required functions in sections 7.1.1 and 7.1.2. To evaluate the usefulness of the artifacts in a realistic scenario, we will conduct a case study based on real product data by an industry partner and we will present queries which allow us to measure the key performance indicators. We have split this case study into two parts to better demonstrate the differences between processes with and without status updates. Section 7.2 evaluates the process without status updates and section 7.2.2 concludes the chapter with the evaluation of the same process with status updates which also includes queries that query both process and product data in a single query by using the links we have placed.

7.1 Process Implementation Evaluation

In this section we start with an evaluation of the correctness of the implemented processes by defining fictional scenarios and testing the implemented artifacts against these scenarios.

7.1.1 Evaluation of Check-In Process

To evaluate the correctness of synchronization of product data with the implemented process we will first define a test scenario and play it out step-by-step with the implementation.

Test scenario

The test scenario consists of two subsequent check-in process executions for the same project performed by a process engineer with an export from the tool ”OPM”. Figure 7.1 shows a schematical overview over the scenario. On top is the process engineer who starts the pro-

cesses. Before the first process the Engineering Database (EDB) is completely empty. The second process triggers three notifications for three different users.

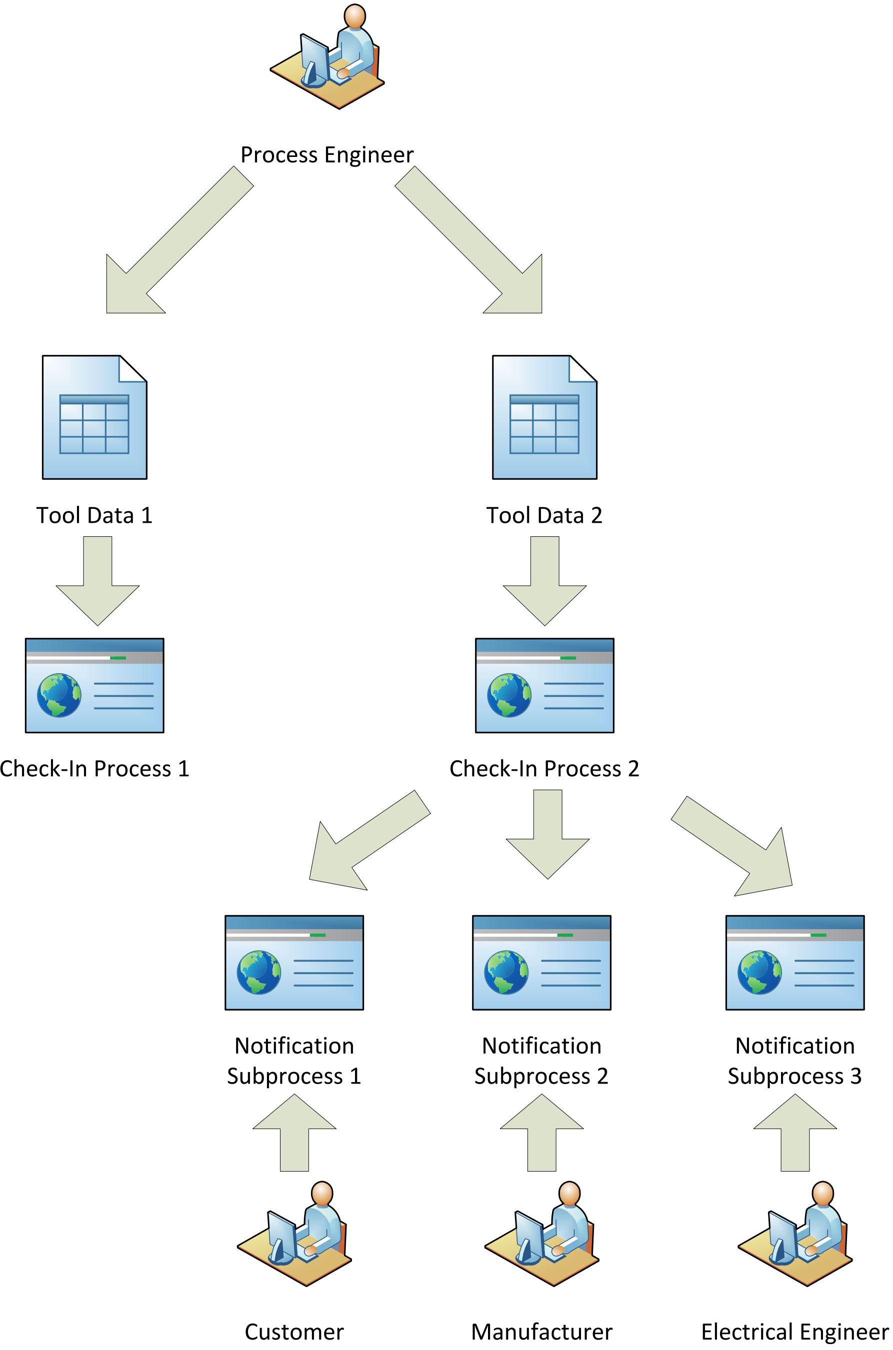

Figure 7.1: Overview of the first evaluation test scenario.

kks0	kks1	kks2	kks3	region	comp	cpu	rack	chan	desc	status
1	ACA10	CE111	XQ01	1	6	20	5	1	description 1	6
1	ACA10	CE112	XQ01	1	7	20	5	1	description 2	6
1	ACA10	CE113	XQ01	1	8	20	5	1	description 3	3
1	ACA10	CE120	XQ01	1	9	20	5	1	description 4	

Table 7.1: Signal data for the first check-in process

kks0	kks1	kks2	kks3	region	comp	cpu	rack	chan	desc	status
1	ACA10	CE111	changed	1	6	20	5	1	description 1	6
1	ACA10	CE112	XQ01	1	7	20	5	changed	description 2	6
1	ACA10	CE113	XQ01	1	8	20	5	1	changed	3
1	ACA10	CE120	XQ01	1	9	20	5	1	description 4	

Table 7.2: Signal data for the second check-in process

Test data for first process

Table 7.1 shows the input data for the first process execution. Note that while this signal contains in part values of real test data, we reduced the amount of fields to a minimum required to demonstrate the functionality and most values are arbitrary and do not reflect functioning components in a real plant. We selected all fields of the hardware path ("region", "comp", "cpu", "rack") which will be used for identification and matching of the signals, the four "kks" fields, "chan" and "desc" which will be checked for changes and finally the field "status" as the last input for the notification mapping. We chose a small amount of four signals for the first check-in. The engineer starting the check-in will select all changes and no notifications to trigger and as a result there will be four signals contained in the EDB after the first process execution and no notifications will be triggered.

Test data for second process

Table 7.2 depicts the input signal data for the second process execution. The columns and amount of signals are identical, only three values of three signals were changed which can be clearly seen by the values "changed" to trigger different notifications. Again the engineer who started the check-in will select all changes but this time he will not omit any notifications. In total, three different notifications will be triggered, based on three different rules and sent to three different stakeholders.

Process 1 execution

For each user interaction during the process we will present the parameters chosen by the user as described in Chapter 6 section 6.3. Listing 7.1 shows the test parameters for the first user task. The user selects the local file to upload (rawData), the transformer for the transformation of the

#	oid	version	kks0	kks1	kks2	kks3	...	chan	desc	status
1	7c84a263	1	1	ACA10	CE111	XQ01	...	1	description 1	6
2	e0e21ae2	1	1	ACA10	CE112	XQ01	...	1	description 2	6
3	2249326b	1	1	ACA10	CE113	XQ01	...	1	description 3	3
4	aa78522c	1	1	ACA10	CE120	XQ01	...	1	description 4	

Table 7.3: Result of EDB query after the first process execution.

file (transformer), the tool from which the uploaded file originated (tool) and the fields by which a signal can be uniquely identified (idFields).

Listing 7.1: The upload task user input parameters of the first process execution.

```
1  {
2    "taskId": "uploadTask",
3    "userId": "process_engineer",
4    "params": {
5      "rawData": "transformers/samples/DA-OPM-TC-1-1.csv",
6      "transformer": "transformers/DA-OPM-TC-1.zip",
7      "tool": "OPM",
8      "idFields": "region, componentNumber, cpuNumber, rackId",
9      "queryParams": ""
10   }
11 }
```

Since the first process is only used to get some pre-existing data into the EDB, the engineer selects all default values determined in the compare task and unselects the notifications of other users which can be seen in 7.2.

Listing 7.2: The select task user input parameters of the first process execution.

```
1  {
2    "taskId": "selectTask",
3    "userId": "process_engineer",
4    "params": {
5      "unSelectNew": "",
6      "unSelectUpdated": "",
7      "unSelectDeleted": "",
8      "unSelectNotifications": "0,1"
9    }
10 }
```

After this selection, the process persists the uploaded and transformed data in the EDB and finishes as expected. The EDB now contains one commit with four signal containing exactly the values shown in 7.1 with the addition of a Object ID (OID) for each signal which is used for faster internal identification of signals and an version for each signal which indicates the number of times a signal has been changed. Running a query on the current data in the EDB would result in Table 7.3.

Process 2 execution

Next we will have a more detailed look at the execution of the second process, which is much more interesting. The process starts again with the upload user task of the process engineer as

can be seen in 7.3. With the exception of the uploaded file, the parameters are identical to the ones used in the upload task of the first process.

Listing 7.3: The upload task user input parameters of the second process execution.

```
1 {
2   "taskId": "uploadTask",
3   "userId": "process_engineer",
4   "params": {
5     "rawData": "transformers/samples/DA-OPM-TC-1-2.csv",
6     "transformer": "transformers/DA-OPM-TC-1.zip",
7     "tool": "OPM",
8     "idFields": "region, componentNumber, cpuNumber, rackId",
9     "queryParams": ""
10   }
11 }
```

After submitting the initial parameters and uploading the file, the file is again transformed using the specified transformer and the data is compared with the existing EDB data. Since the uploaded signals do not have an oid yet, they are matched using the fields specified in the ”idFields” parameter: two signals are identical, if their ”idFields’ match.

This time the engineer performing the check in accepts all default settings and does not unselect any notifications. This results in three notification subprocesses being triggered for three different recipients:

Notification to Customer The customer is notified about the changes to signal 3, due to the changed field ”desc” and status ”3”. He decides to accept the change, which is reflected by the input parameters for his decision task in listing 7.4.

Listing 7.4: The decide task user input parameters of the second process execution.

```
1 {
2   "taskId": "decideTask",
3   "userId": "electrical_engineer",
4   "params": {
5     "declineNew": "",
6     "declineUpdated": "0",
7     "declineDeleted": ""
8   }
9 }
```

Notification to Manufacturer The manufacturer receives a notification about signal 2, due to the changes to field ”chan” and status ”6”. Just like the customer he accepts the changes which is reflected by the the input parameters of his task in listing 7.5.

Listing 7.5: The decide task user input parameters of the second process execution.

```
1 {
2   "taskId": "decideTask",
3   "userId": "manufacturer",
4   "params": {
5     "declineNew": "",
6     "declineUpdated": "",
7     "declineDeleted": ""
8   }
9 }
```

Notification to Electrical Engineer The electrical engineer receives a notification about two changed signals: signal 1 due to status ”6” and the changed field ”kks3”, and signal 2 due to status ”6” and the changed field ”chan”. He accepts the changes to signal 1, but denies signal 2. Note that this will override the decision of the manufacturer, who decided to accept signal 2.

Listing 7.6: The decide task user input parameters of the second process execution.

```
1 {
2   "taskId": "decideTask",
3   "userId": "customer",
4   "params": {
5     "declineNew": "",
6     "declineUpdated": "",
7     "declineDeleted": ""
8   }
9 }
```

After all three stakeholders have submitted their decisions the results of their decisions are merged. Signals which were denied are removed from the pending commit and the remaining signals are persisted. The final result can be seen in Table 7.4.

#	oid	version	kks0	kks1	kks2	kks3	...	chan	desc	status
1	7c84a263	2	1	ACA10	CE111	changed	...	1	description 1	6
2	e0e21ae2	1	1	ACA10	CE112	XQ01	...	1	description 2	6
3	2249326b	2	1	ACA10	CE113	XQ01	...	1	changed	3
4	aa78522c	1	1	ACA10	CE120	XQ01	...	1	description 4	

Table 7.4: Result of EDB query after the second process execution.

Running this test scenario with the implemented artifact yielded the expected result.

7.1.2 Evaluation of Status Update Process

After having shown that the Check-In process works as intended, we are going to demonstrate that the status update process is also correctly implemented. To accomplish this we first use the Check-In process to fill the product database with a data set and then run the status process to change the status of some signals. Figure 7.2 shows the scenario schematically.

As input data for the initial Check-In we use the same data as in the previous section in table 7.1. Next the Status Update Process is started. The process contains two user tasks for which input has to be provided. For the first user task, the ”Set Query Params Task”, the query parameters have to be provided which allows us to narrow down the set of signals for which we want to update the status. Listing 7.7 shows the task configuration for this task. In this case we only want to change the status of the last signal and so we select it by setting ”kks2” to ”CE120”. The second and last user task in this process is the ”Select Status Task” and the test configuration of this task can be seen in 7.8. In this task the user can select the status that should be applied to the signals selected by the query. In this case the status is set to ”changed” for the last signal.

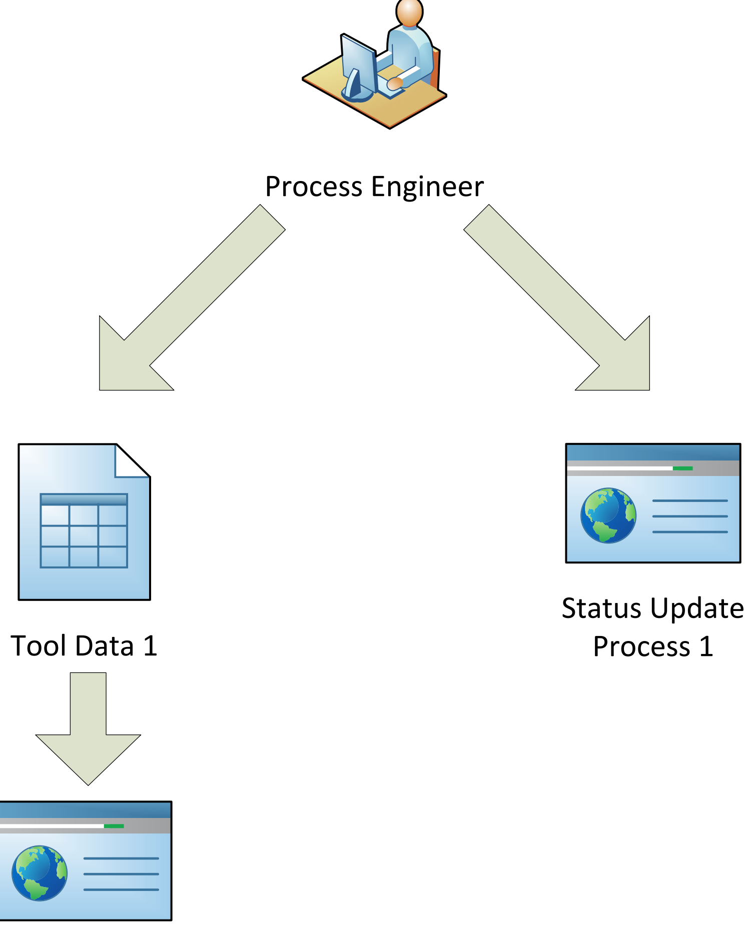

Figure 7.2: Overview of the second evaluation test scenario.

Listing 7.7: The "Set Query Params Task" user input parameters of the status process execution.

```
1 {
2   "taskId": "setQueryParamsTask",
3   "userId": "process_engineer",
4   "params": {
5     "queryParams": "kks2:CE120"
6   }
7 }
```

Listing 7.8: The "Select Status Task" user input parameters of the status process execution.

```
1 {
2   "taskId": "selectStatusTask",
3   "userId": "process_engineer",
4   "params": {
5     "status": "changed"
6   }
7 }
```

#	oid	version	kks0	kks1	kks2	kks3	...	chan	desc	status
1	7c84a263	1	1	ACA10	CE111	XQ01	...	1	description 1	6
2	e0e21ae2	1	1	ACA10	CE112	XQ01	...	1	description 2	6
3	2249326b	1	1	ACA10	CE113	XQ01	...	1	description 3	3
4	aa78522c	2	1	ACA10	CE120	XQ01	...	1	description 4	changed

Table 7.5: Result of EDB query after the status update process execution.

Finally, after the execution of the status process, we execute an EDB query to verify the correct update of the status in the database. Table 7.5 shows the result. As expected, only the status of the last signal was changed.

7.2 Case Study

To demonstrate the solution in a realistic environment we have performed a small case study with real data provided by the industry partner. We got three consecutive tool exports from Eplan from the same project, spanning a period over a year. The first file contains 1636 signals, the second 1936 and the third 1960. For each check-in we altered the date and set each of them one month apart from each other. We left the original files semantically unaltered and only did some syntactical adaptions. We have split the case study into two parts: in the first part we will run three check-in processes without setting any signal states and measure product specific key performance indicators and some basic process specific KPIs. In the second part of the study, we will change the state of a subset of signals to measure more process specific KPIs.

7.2.1 Case Study without Status Updates.

Figure 7.3 shows the three processes CI 1, CI 2 and CI 3 which were used in the test run.

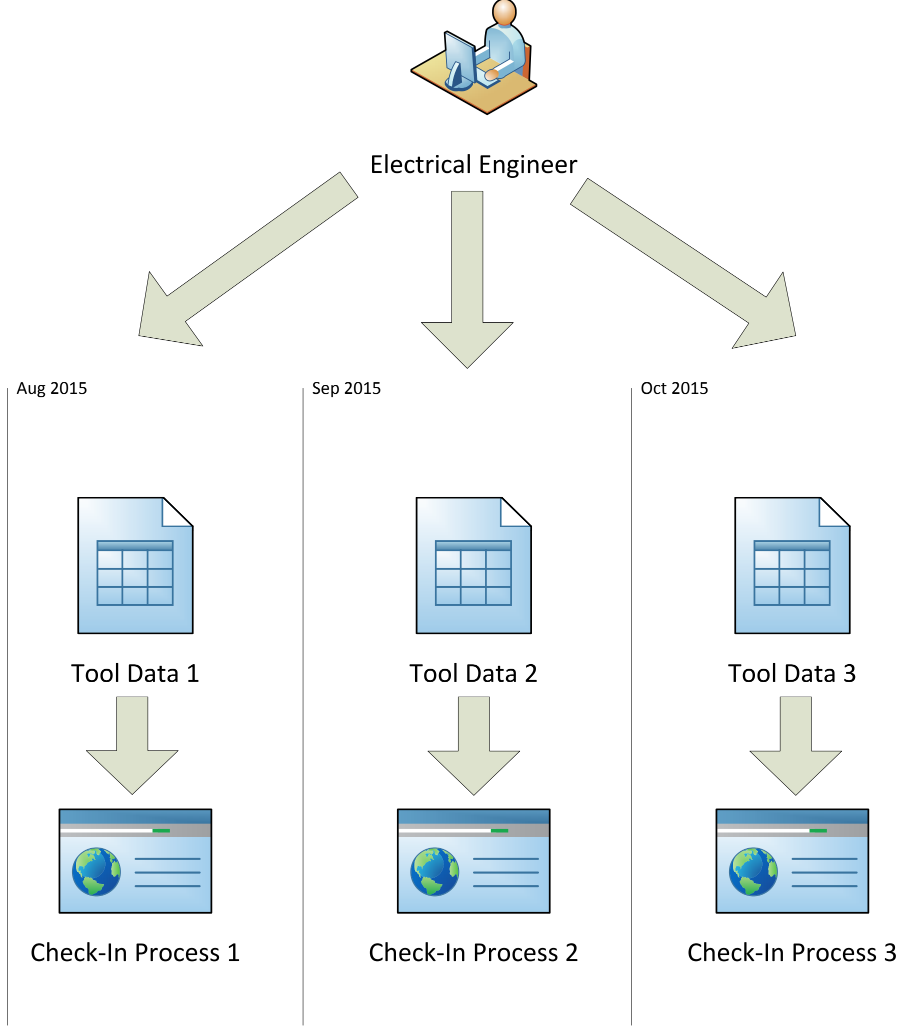

Figure 7.3: Overview of the first case-study scenario.

Basic Process Metrics.

Next we are evaluating the capabilites for process analysis of the solution, starting with basic process metrics.

Amount of Signal Operations per Process. Figure 7.4 shows the amount of signal operations per process. The exact values can be seen in table 7.6 which also shows the total amount of signals in the product database recorded after each process run. The amount of signal operations per commit can be obtained by the query in listing 7.9. This query returns the amount of all new, updated, unchanged and deleted signals for this commit.

	CI 1	CI 2	CI 3
new	1636	752	24
changed	0	796	141
unchanged	0	388	1795
deleted	0	452	0
DB	1636	1936	1960

Table 7.6: The number of signal operations detected in each process and the total amount of signals in the product database after each run.

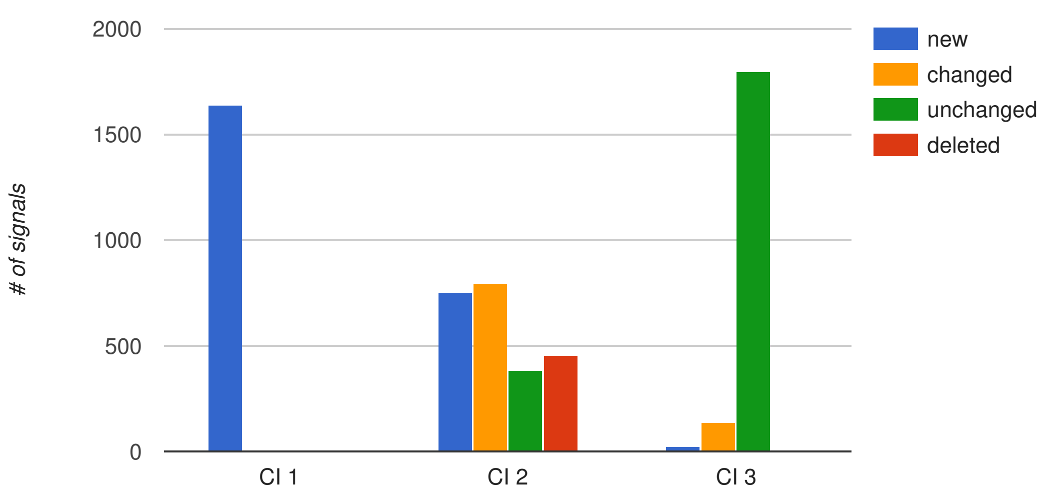

Figure 7.4: The number of signal operations detected in each process. The exact values can be seen in table 7.6.

Listing 7.9: The SQL query returning the amount and type of signal operations for all processes.

```
1 SELECT PROC_INST_ID_,
2        NAME_,
3        TEXT_
4 FROM ACT_HI_VARINST
5 WHERE NAME_='amountOfAdded' OR NAME_='amountOfUpdated' OR NAME_='amountOfUnchanged' OR NAME_='
      amountOfDeleted'
```

Process Element Durations. Table 7.7 shows the process elements of the executed process and their durations in ms. They are stored in the tables *ACT_HI_PROCINST* and *ACT_HI_ACTINST*

Activity	CI 1	CI 2	CI 3	μ	σ
start	6	0	0	2	2
uploadTask	86	15	15	39	33
transformTask	5880	4994	3361	4745	1043
queryTask	130	4076	3275	2494	1703
compareTask	1083	10398	6752	6078	3832
checkNotificationsTask	3	13	1	6	5
selectTask	249	661	121	344	230
madeChanges	120	11	2	44	53
needNotifications	7	3	0	3	2
mergeTask	4	310	38	117	136
persistTask	11232	72015	36534	39927	24930
end	0	0	0	0	0
total	18802	92497	50100	53800	30199

Table 7.7: The process elements and their durations in ms as recorded by Activiti.

in the database. On the vertical axis we have the process elements (start, uploadTask, transformTask, queryTask, compareTask, checkNotificationsTask, selectTask, madeChanges, needNotifications, persistTask, end) and on the horizontal axis the durations for each of the three processes, as well as the average duration and the standard deviation. As can be seen not only activities like user and service tasks are recorded in this table, but also events and gateways. To obtain this information we have used two SQL statements. The first statement returns all process elements and their durations for all processes and can be seen in listing 7.10. The second statement, as shown in listing 7.11, is used to collect all average durations and standard deviations for each type of activity. Note that *stddev_pop* is an h2 specific aggregate function and not included in the SQL standard and therefore this may not work with other SQL databases.

Listing 7.10: The SQL query to obtain the list and duration of all process elements.

```
1 SELECT PROC_INST_ID_,
2        ACT_ID_,
3        DURATION_
4 FROM ACT_HI_ACTINST
5 GROUP BY PROC_INST_ID_, ACT_ID_
6 ORDER BY START_TIME_, END_TIME_ ASC
```

Listing 7.11: The SQL query to obtain the average duration and standard deviation of process elements.

```
1 SELECT ACT_ID_,
2        AVG(DURATION_),
3        ROUND(STDDEV_POP(DURATION_),2)
4 FROM ACT_HI_ACTINST
5 GROUP BY ACT_ID_
```

The data for the last line was derived with two more queries shown in listings 7.12 and 7.13. They are quite similar to the first two queries, except that the table *ACT_HI_PROCINST* is queried.

Listing 7.12: The SQL query to obtain the list and duration of all processes.

```
1 SELECT PROC_INST_ID_,
2        DURATION_
3 FROM ACT_HI_PROCINST
4 ORDER BY START_TIME_
```

Listing 7.13: The SQL query to obtain the average duration and standard deviation of processes.

```
1 SELECT AVG(DURATION_),
2        ROUND(STDDEV_POP(DURATION_),2)
3 FROM ACT_HI_PROCINST
```

Looking at the data we can identify 4 processes which have a duration of more than a second, namely the "transformTask", "queryTask", "compareTask" and "persistTask".

TransformTask Observations. The "transformTask" is the first activity with a duration greater than one second with an average of 4745 ms and a standard deviation of 1043.34 ms. Interestingly the duration decreases with each process execution which might be caused by some preparations the OpenEngSB services have to do before execution or some internal optimizations in the component that executes the transformation (Smooks [1]). Overall the performance seems to be acceptable.

QueryTask Observations. The second notable task is the "queryTask" with an average duration of 2494 ms and a standard deviation of 1703.05 ms. The first execution of the "queryTask" is very short with 130 ms because the product database is empty at this point and nothing has to be returned, the other two queries are equal with the third query (3275 ms) being faster than the second (4076 ms), despite having to return more results than the first. This could be due to JPA or H2-internal optimizations. Overall the performance seems to be acceptable.

CompareTask Observations. The third notable task is the "compareTask" with an average duration of 6078 ms and a standard deviation of 3832.61 ms. In the first process execution there is nothing to compare yet and therefore it is quite fast with 1083 ms. This increases significantly in the second process execution to 10398 ms where an actual comparison takes place and many different operations are detected (as we have already shown in table 7.6). In the third process execution it is faster with 6752 ms even though the amount of signals that are compared is almost equal to the amount in the second execution, however there are far more unchanged signals this time. Since the detection whether a signal is changed or unchanged should be equal, this indicates that changed signals cause a significant overhead during this task which could be potentially optimized. Overall the performance is acceptable though.

PersistTask Observations. The last notable task is the "persistTask" with an average duration of 39927 ms and a standard deviation of 24930.27 ms. This is by far the task with the highest duration and standard deviation and it can be clearly seen that it is a performance bottleneck, as

[1] Smooks: http://www.smooks.org

the first task has a duration of 11 s (commit with 1636 signals: 1636 new), whereas the second has a duration of 72 s (commit with 2000 signals: 752 new, 796 changed and 452 deleted) and the third 40 s (commit with 165 signals: 24 new and 141 changed). So the last commit takes approximately four times as long as the first one despite having only a tenth of the database operations. This has to be investigated further as it seems that changes and deletes cause a massive increase in commit duration.

Database-Agnostic Querying It is also possible to obtain all data shown in tables 7.7 and 7.9 in a database-agnostic way by using Activiti's HistoryService which allows to query for all activities and processes based on specific parameters. However, because the return types of the HistoryService methods are limited to the types "HistoricProcessInstance" and "HistoricActivityInstance", it is required to extract all information from this lists programmatically and also necessary to implement the aggregate functions (avg, stddev_pop) which we have used earlier and are provided by h2. As this would yield little benefits in the scope of this work, we did not explore this path further.

Product Key Performance Indicators

In this section we will evaluate the capabilities of the artifact to measure various product specific key performance indicators (KPI). Based on the KPIs introduced in chapter 5, we will present SQL queries which can be used to derive the KPIs from the product database.

Amount of signals per month. To compute the amount of signals per month in a specific project, we have to query the history table of that project, in this case "HISTORY_2F6132CB49" as can be seen in listing 7.14. The query consist on a subtraction of two subqueries (lines 2 - 5 and 9 - 12), one for computing the total amount of signals with the "INSERT" operation up to a specific date (line 4), minus all "DELETE" operations at the same date. The result is the total amount of signals that represented the product at this date (line 11).

Listing 7.14: The SQL query returning the amount of signals at the end of the last month.

```
1 SELECT (
2   SELECT COUNT(*)
3   FROM HISTORY_2F6132CB49
4   WHERE DATEDIFF('MONTH', REV_TIMESTAMP, current_date) >= 1
5   AND REV_OPERATION = 'INSERT'
6 )
7 -
8 (
9   SELECT COUNT(*)
10  FROM HISTORY_2F6132CB49
11  WHERE DATEDIFF('MONTH', REV_TIMESTAMP, current_date) >= 1
12  AND REV_OPERATION = 'DELETE'
13  )
14 AS diff
```

Amount of components per month. The computation of the amount of specific components is quite similar to the total amount of signals, except that the "WHERE"-clause of each subquery also has to contain the component that should be queried, as can be seen in listing 7.15 in lines 6

and 14. Analogously, we can easily drill further down to smaller parts of the product by adding more and more parameters to the ”WHERE”-clause as needed.

Listing 7.15: The SQL query returning the amount of components with value ”001” at the end of the last month.

```
1  SELECT (
2    SELECT COUNT(*)
3    FROM HISTORY_2F6132CB49
4    WHERE DATEDIFF('MONTH',REV_TIMESTAMP,current_date) >= 1
5    AND REV_OPERATION = 'INSERT'
6    AND COMPONENTNUMBER = '001'
7  )
8  -
9  (
10   SELECT COUNT(*)
11   FROM HISTORY_2F6132CB49
12   WHERE DATEDIFF('MONTH',REV_TIMESTAMP,current_date) >= 1
13   AND REV_OPERATION = 'DELETE'
14   AND COMPONENTNUMBER = '001'
15 )
16 AS diff
```

Amount of signal operations per month. The next query collects the amount of signal operations in a specific month. Operations include the addition of new signals, updates of existing signals and deletion of existing signals. The query can be seen in listing 7.16.

Listing 7.16: The SQL query returning the amount and type of signal operations for a specific month.

```
1 SELECT COUNT(*) AS count,
2        CAST(REV_OPERATION AS VARCHAR ) AS operation,
3        MONTH(REV_TIMESTAMP) AS month
4 FROM HISTORY_2F6132CB49
5 WHERE DATEDIFF('MONTH', REV_TIMESTAMP, current_date) = 1
6 GROUP BY operation, month
```

Amount of signal operations per month and user. This query is identical to the query for all signal operations, except that another parameter for a specific user is added. The query can be seen in listing 7.17.

Listing 7.17: The SQL query returning the amount and type of signal operations for a specific month and user.

```
1 SELECT COUNT(*) AS count,
2        CAST(REV_OPERATION AS VARCHAR) AS operation,
3        MONTH(REV_TIMESTAMP) AS month
4 FROM HISTORY_2F6132CB49
5 WHERE DATEDIFF('MONTH',REV_TIMESTAMP,current_date) = 1
6 AND REV_USER='admin'
7 GROUP BY operation, month
```

Amount of signal operations per month and phase. This query is similar to the query for all signal operations, except that the results are grouped by phases (or states). The query can be seen in listing 7.18.

Listing 7.18: The SQL query returning the amount of signal operations for all phases in a specific month.

```
1  SELECT h1.status AS status,
2         COUNT(distinct(h1.UUID)) AS count,
3         FORMATDATETIME(max(h1.REV_TIMESTAMP), 'YYYY-MM') AS month
4  FROM HISTORY_2F6132CB49 h1
5  INNER JOIN (SELECT UUID, max(REV_TIMESTAMP) AS max
6              FROM HISTORY_2F6132CB49
7              WHERE REV_OPERATION != 'DELETE'
8              AND DATEDIFF('MONTH',REV_TIMESTAMP,current_date) >= 1
9              GROUP BY UUID) h2
10 ON h1.UUID = h2.UUID
11 AND h1.REV_TIMESTAMP = h2.max
12 WHERE h1.UUID NOT IN (SELECT UUID
13                        FROM HISTORY_2F6132CB49
14                        WHERE REV_OPERATION = 'DELETE'
15                        AND DATEDIFF('MONTH',REV_TIMESTAMP,current_date) >= 1)
16 GROUP BY status
```

7.2.2 Case study with Status Updates.

The plain case study showed the effectiveness of the solution solving and analyzing real scenarios. Now we will extend and repeat this scenario to show a greater potential, by using status updates to trigger notifications. We are using the same tool data files as in the original case study, except this time we will change signal states in between check-ins to trigger notifications for updated and deleted files. Figure 7.5 shows the five processes and their order used in this scenario. The three check-ins are CI 1, CI 2 and CI 3 and the two status updates are SU 1 and SU 2. The status updates are updating all signals which have region=10, cpuNumber=001 and rackId=01. The first status update sets all states for the selected signals to *3* whereas the second status update sets them to *4*. By looking at the notification tables in the requirements chapter 5 we can see that these states should trigger a notification for (1) the customer in case of state *3* and (2) the process engineer in case of state *4*.

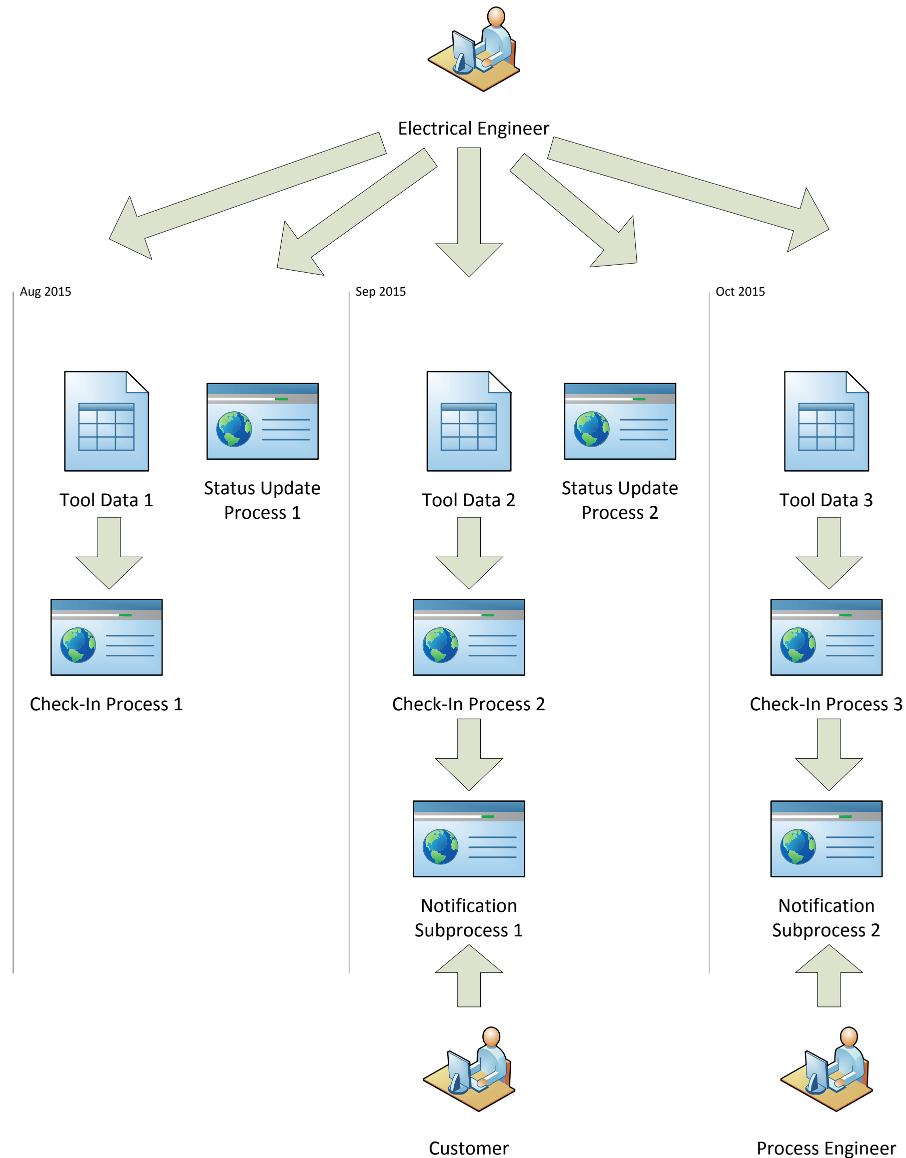

Figure 7.5: Overview of the second case-study scenario.

Basic Process Metrics.

We again start the analysis with the basic process metrics.

Signal Operations per Process. Figure 7.4 shows the amount of signal operations per process. The exact values can be seen in table 7.8 which also shows the total amount of signals in the product database after each process run. The check-in numbers are identical as in 7.6. The status update process SU 1 updates the state of 600 signals, whereas process SU 2 740 signals. Both do not change the total amount of signals in the product database.

	CI 1	SU 1	CI 2	SU 2	CI 3
new	1636	0	752	0	24
changed	0	600	796	740	141
unchanged	0	0	388	0	1795
deleted	0	0	452	0	0
DB	1636	1636	1936	1936	1960

Table 7.8: The number of signal operations detected in each process and the total amount of signals in the product database after each run.

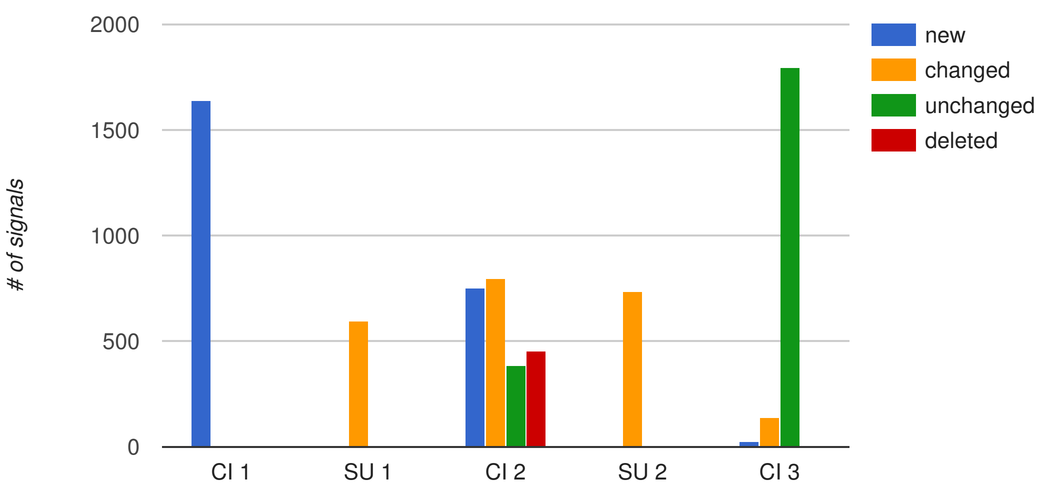

Figure 7.6: The number of signal operations detected in each process. The exact values can be seen in table 7.8.

Process Element Durations. Next we will have again a look at the process element durations and standard deviations. Table 7.9 shows the results obtained with the queries presented in the first part of the case study. As we can see the amount of elements has increased significantly from 12 to 20 due to the elements of the SU processes and the Notification subprocesses which are now triggered as we had anticipated. The new element durations do not add any noteworthy anomalies or bottlenecks. What can be seen however is a further increase in the duration of activities which already required a notable amount of time to complete, especially the ”transformTask”, ”queryTask” and the ”persistTask” activity which we have already previously identified as a performance bottleneck. The average duration of the ”transformTask” increased

Activity	CI 1	SU 1	CI 2	SU 2	CI 3	μ	σ
start	14	0	0	0	1	3	5
uploadTask	119	-	13	-	19	50	48
setQueryParamsTask	-	32	-	19	-	26	6
transformTask	7579	-	5326	-	4092	5666	1443
queryTask	233	2245	3126	3124	6949	3135	2179
compareTask	2128	-	11026	-	7376	6843	3652
selectStatusTask	-	55	-	45	-	50	5
applyStatusTask	-	56	-	35	-	46	10
checkNotificationsTask	5	-	77	-	7	30	33
selectTask	447	-	545	-	129	374	177
madeChanges	120	-	8	-	1	43	54
needNotifications	10	-	24	-	6	13	7
notifySubprocess	-	-	108	-	76	92	16
notifyTask	-	-	1	-	0	1	0
decideTask	-	-	81	-	59	70	11
applyDecisionTask	-	-	16	-	9	13	3
notifyEnd	-	-	0	-	0	0	0
mergeTask	9	-	291	-	40	113	126
persistTask	12339	84543	76110	226172	58516	91536	71800
end	0	0	0	0	0	0	0
total	23005	86932	96659	229397	77214	102641	68319

Table 7.9: The process elements and their durations in ms as recorded by Activiti.

from 4745 to 5666 ms and the standard deviation from 1043.34 to 1443.68 ms. Interestingly, we can again see that the duration decreases with each process execution. The average duration of the ”queryTask” from 2494 to 3135 ms and the standard deviation from 1703.05 to 2179.33 ms. While the first ”queryTask” executions are roughly equal, the last one took more than twice as much with 6949 ms. The most significant increase can however be seen with the ”persistTask” which we had already previously identified as a performance bottleneck. The average duration increased from 39927 to 91536 ms and the standard deviation from 24930.27 to 71800 ms, further displaying the severity of this bottleneck. The last task that took more than a few seconds to run, the ”compareTask” showed only a small increase in its average duration from 6078 to 6843 ms and even a decrease in its standard deviation from 3832.61 down to 3652.07 ms.

Process Key Performance Indicators.

Next, we will evaluate if the solution is capable of measuring the process key performance indicators by formulating queries.

Amount of Notifications per User.

The first query tested in the extended case study is the amount of notifications per user. The query can be seen in listing 7.19. The result can be completely derived from the *ACT_HI_ACTINST* table from the Activiti history by counting the number of *decideTasks* for each assigned user. These tasks only exist when a notification is triggered and are assigned to the recipient of the notification.

Listing 7.19: The SQL query to obtain the amount of notifications per user.

```
1 SELECT ASSIGNEE_,
2        COUNT(*)
3 FROM ACT_HI_ACTINST
4 WHERE ACT_ID_ = 'decideTask'
5 GROUP BY ASSIGNEE_
```

Average Duration of Notifications per User.

The second query returns the average time each notified user needs to complete his task. This query is quite similar to the previous except instead of the count function we are using the average function. The query can be seen in listing 7.20. Note that while a notification subprocess executes more than one task, we are only interested in the durations of the *decideTask*, because it is the only user task. The other tasks are system tasks and therefore not relevant.

Listing 7.20: The SQL query to obtain the average duration of notifications per user.

```
1 SELECT ASSIGNEE_,
2        AVG(DURATION_)
3 FROM ACT_HI_ACTINST
4 WHERE ACT_ID_ = 'decideTask'
5 GROUP BY ASSIGNEE_
```

List of Signals that triggered Notifications.

The third query returns the lists of signals that triggered notifications. The query can be seen in listing 7.21. Here we are querying Activiti's *ACT_GE_BYTEARRAY* table that stores serialized objects, in this case the process variable "hist.var-notifications". The result has to be deserialized into NotificationSet objects which contain the signals. While this approach can be used to obtain a complete list, it has some drawbacks which we are going to discuss further in the "limitations" section in chapter 8 and suggest solutions in the "future work" section of chapter 9.

Listing 7.21: The SQL query to obtain all signals which have triggered notifications.

```
1 SELECT BYTES_
2 FROM ACT_GE_BYTEARRAY
3 WHERE NAME_ = 'hist.var-notifications'
```

Amount of Status Updates per Month.

The next query shows how we can use both the process and product database in a single query to get the amount of status updates per month. The query can be seen in listing 7.22. Status

updates are stored like normal signal changes in the product database (this circumstance is layed out in more detail in the discussion chapter 8). The only (efficient) way to distinguish between status updates and other changes is by querying the process type. Therefore we have to select this process type from the Activiti process history table *ACT_HI_PROCINST*, join it with the variable instance table *ACT_HI_VARINST* which contains the commitId which links to the signals in the EDBI's *HISTORY* table that were updated. Therefore we join the *HISTORY* table via *REV_COMMIT*.

Listing 7.22: The SQL query to obtain the amount of all status updates.

```
1 SELECT COUNT(*)
2 FROM ACT_HI_PROCINST p
3 JOIN ACT_HI_VARINST v ON (p.PROC_INST_ID_ = v.PROC_INST_ID_)
4 JOIN HISTORY_2F6132CB49 h ON(v.TEXT_ = h.REV_COMMIT)
5 WHERE p.PROC_DEF_ID_ like 'statusProcess%'
6 AND v.NAME_ = 'commitId'
7 AND DATEDIFF('MONTH',REV_TIMESTAMP,current_date) = 1
```

Amount of changes without Status Updates.

This query is similar to the previous one, except this time we want to compute the amount of changes without status updates. The query can be seen in listing 7.23. This time we are selecting all signals in the EDBI which belong to the *checkinProcess* and with the *REV_OPERATION* "UPDATE", to exclude "INSERT" and "DELETE" operations.

Listing 7.23: The SQL query to obtain the amount of all changes without status updates.

```
1 SELECT COUNT(*)
2 FROM ACT_HI_PROCINST p
3 JOIN ACT_HI_VARINST v ON (p.PROC_INST_ID_ = v.PROC_INST_ID_)
4 JOIN HISTORY_2F6132CB49 h ON(v.TEXT_ = h.REV_COMMIT)
5 WHERE p.PROC_DEF_ID_ like 'checkinProcess%'
6 AND v.NAME_ = 'commitId'
7 AND h.REV_OPERATION = 'UPDATE'
8 AND DATEDIFF('MONTH',REV_TIMESTAMP,current_date) = 1
```

Detection and Visualization of Process Path

Finally, we demonstrate the applicability of process mining by visualizing the path executed by a process instance with the process mining algorithm by [Bayraktar, 2011] in ProM. To do this, we extract the collected event log data from table *ACT_HI_ACTINST* and transform it to an event log in XES-format. We only removed the events recorded by gateways, because otherwise they will be detected as normal activities by the algorithm. We loaded the event log data into ProM and selected the "BPMN Analysis (using Causal Net Miner)" with the proposed default settings. The result was the path taken through the implemented Check-In process as a BPMN process as seen in figure 7.7. This can now be compared with the originally modeled BPMN process (and subprocess) in figures 6.2 and 6.3.

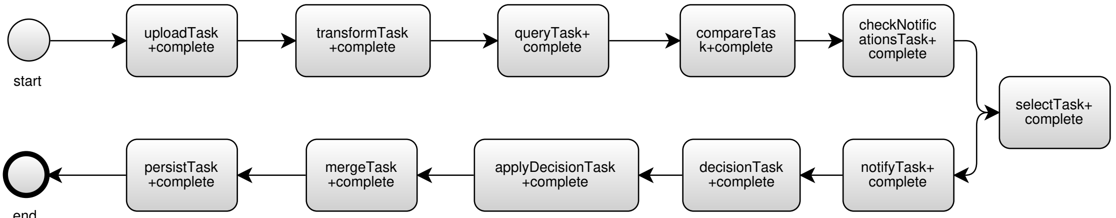

Figure 7.7: Discovered process as path through the processes in 6.2 and 6.3.

Detection and Visualization of Social Networks

Next we also tried the social network discovery algorithm as introduced in [van Der Aalst et al., 2005] and implemented in [Song and van der Aalst, 2008]. For this we added the ”org:resource” key to the user tasks, start and end events in the event log and set it to the user who started the process or provided input for the activity. We removed all the non-user tasks as well. To provide a more insightful result we added a second trace by the same process initiatior (Electrical Engineer) with additional Notification recipients (Manufacturer and Process Engineer). We selected the ”Hand over Network” algorithm in ProM. The result can be seen in figure 7.8. The circles are the four different actors encountered in the scenario (electrical engineer, process engineer, customer and manufacturer) the directed edges indicate the succession (or handover of work). The size of the spheres is based on the number of edges (degree) going in and out of the circles. As can be seen the alogrithm provides and interesting overview of both cases and can be used to identify important stakeholders. However, the algorithm does not consider tasks executed in parallel. While Case 1 correctly depicts the direct handover of work between the stakeholders (Electrical Engineer and Customer), in Case 2 we can see that the work is passed between all three stakeholders, from the Electrical Engineer, to the Manufacturer, to the Process Engineer and then back to the Electrical Engineer. However, in the real process the Manufacturer and Process Engineer make their decisions in parallel and independently and are only preceded and succeeded by the Electrical Engineer.

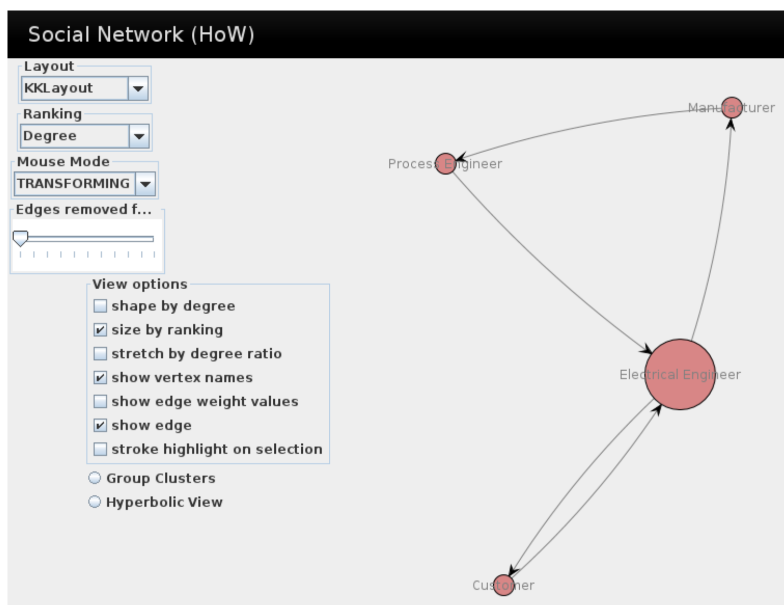

Figure 7.8: Discovered Handover-of-work network.

In this case the "Mine for a Similar-Task Social Network" algorithm provides a more accurate insight in the scenario at hand, as can be seen in figure 7.9. Here the arrangement of the edges is similar to the actual scenario. The connections between the stakeholders are only between the Electrical Engineer, who started the processes, and the notified stakeholders.

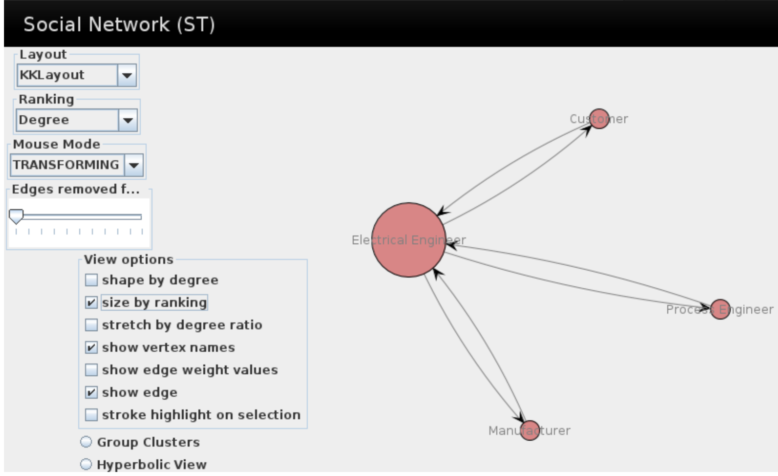

Figure 7.9: Discovered Similar-Task Social Network.

7.3 Summary of Key Performance Indicators

To conclude the evaluation chapter we will summarize and link the key performance indicators we measured separated into process and product KPIs.

7.3.1 Product Key Performance Indicators

The key performance indicators we successfully measured with the artifact are:

- amount of signals per month 7.14
- amount of components per month 7.15
- amount of signal operations per month 7.16
- amount of signal operations per month and user 7.17
- amount of signal operations per month and phase 7.18

7.3.2 Process Key Performance Indicators

The key performance indicators we successfully measured with the artifact are:

- amount of notifications per user 7.19
- average duration of notification per user 7.20
- list of signals that trigger notifications 7.21
- amount of status updates per month 7.22
- amount of changes per month without status updates 7.23

CHAPTER 8

Discussion

In this chapter we will discuss findings and results of the thesis. The chapter consists of two sections: in the first section we will discuss the research issues introduced in chapter 3, elaborate on how we solved them and why we solved them the way we solved them. In the second section we will discuss limitations of the created artifacts as well as limitations of the evaluation of these artifacts.

8.1 Discussion of Research Issues

In this section we will discuss how we solved the research issues given in chapter 3.

8.1.1 RI-1 Process definition and implementation

To satisfy all the requirements given in the Use Case chapter 5, we had to define and imlement a total of three processes, the Check-In process 6.2, its associated subprocess that handles notifications 6.3 and the process to update states of signals throughout their process phases 6.4. In detail we laid out the elements of the processes, especially the tasks and the input and output parameters of each and every task in chapter 6. We have also introduced the underlying architecture 6.1, describing the context in which the processes run in the Engineering Service Bus, and the major links to other utilized services.

Compared with previous process definitions (see figures 5.3, 5.4 and 5.5 in chapter 5) the process was simplified by increasing the abstraction by moving the signal comparison subtask to a service instead of modeling the complete business logic in BPMN. The original subtask only contained a minimum of business logic required to meet all requirements and therefore an implementation in BPMN was not feasible. We also removed all events, as they are no longer explicitly required (see also RI-2.2). As a strict separation between user and system tasks is required, we also had to decrease the abstraction in some cases and added more system tasks: For example the ”Checkin csv” task in 5.4 was split up into an ”Upload Product Data” task (for

user input) and a ”Transform Product Data” task (for the transformation of the uploaded data to the application internal model).

8.1.2 RI-2 Bridging between process and product data

To link process data with product data some kind of bridging is required. This raises the question on how to store process and product data as there are three possibilities to solve this problem:

1. storing product data in the event log
2. storing event log data in the product database
3. storing process and product data in separate locations

Storing product data in the event log. Storing all changes to product data to the event logs is impractical due to the large amount of product data that usually needs to be tracked during a process execution, which implicates creating a performance bottleneck. Additionally, the model of the product data can vary greatly: in the most simple cases it is only a simple collection where each element contains multiple key-value pair entries. In more complex cases the product can be represented by a large object tree. Sometimes it can be a mixture of different product models (if different types of products need to be supported by the same process) or in the worst case, product data models can even change over time due to new arising requirements, enforcing an evolution of the event log model as well.

Storing event log data in the product database. For similar reasons this approach is unpractical as the first one. Additionally, not every process execution leads to changes in a product, but still process events occurred that need to be recorded. Further problems arise when multiple product databases are used, then the process events are spread out as well and if multiple product databases are used during a single process execution then the problem arises in which of these the process data should be stored or maybe they are stored in all databases, leading to redundant information.

Storing process and product data in separate locations. Due to the disadvantages of the two already mentioned approaches, this seems to be the only reasonable one. Keeping process and product data separate allows to change and optimize them separately, without side-effects. Fortunately the used components, the Activiti process engine and the Engineering Database (EDB) support this approach as we are going to examine next.

RI-2.1 Structure of the Process and Product Data Model

In the main research issue we have already laid out the reasons for a strict separation of the process and product data models. Next we are going to explain how we did this in our solution. Concerning the product model we were bound to use the Engineering Database (EDB) in combination with the Engineering Database Index (EDBI) as the former is required for the latter due to technical restrictions at the time being. From our analysis’ point of view only the

product model of the EDBI is important and we have introduced that in chapter 6 in table 6.8. For the process model we have found the one provided by Activiti 6.7 to be well suited. These models allow us to keep process and product data separated. The only exception is the ”status” information which is semantically a process specific attribute which is stored as a single field similar to product attributes in the product model. Since the status of every single data point of the product can be changed individually (just like product attributes) and has to be frequently available for product data queries (just like product attributes) it is arguably reasonable to store it alongside of other product data. Nevertheless, we suggest a closer investigation of the benefits of a separation of the status information from the product model for future work.

RI-2.2 Placement of Bridging Events

After the implementation of the process we had to decide which and where bridging events should be set which connect the separated process and product data. Whenever product data is persisted, the Engineering Database creates a commit which contains a list of new, deleted and updated product elements. This commit has a generated Universally Unique Identifier (UUID) [Leach et al., 2005] (field UUID in the tables of 6.8. We found that in the actual process implementation the variable context of Activiti is the best place to store the commit's UUID instead of using separate events. This creates an entry in table *ACT_HI_VARINST* (see fig 6.7) which can be easily joined in an SQL-query with the EDBI tables. The variable is created after the commit has been successfully completed in the ”Persist changes” task of both implemented workflows (see figs 6.2, 6.4).

8.1.3 RI-3 Process efficiency and effectiveness

This research issue dealt mainly with the evaluation part and we have split it into two sub-issues which we are going to discuss next:

RI-3.1 Initial effort and correctness

Concerning the effort the engineers only have to set up their tool-to-domain transformers once for each project (or can even reuse and adapt old transformers from other projects) which is a matter of minutes for experienced users. After that they simply have to export the data from their tools, start the process, upload their data and select the changes. They do no longer have to notify stakeholders, as the system is capable of detecting automatically when someone should be notified, can notify them and handle the decisions of the recipients. Estimates by the industry partner for the synchronization process alone (without notifications) was an average of 30 minutes. Our synchronization process (with simulated and instantaneous user input) in the conducted case study executes in 65 seconds on average. By fixing the EDB-performance problem this time span could be reduced even further (see also limitations 8.2). Of course, with a user interface (which was not part of this work) and real user input the process will take longer, however we have shown that the fundamental parts of the process can already improve the stakeholders workflow significantly.

And for the correctness of our implementation, we have presented artificial test scenarios in the evaluation chapter 7 which trigger all of the available features of the processes. We could verify that the data was checked in correctly and notifications were handled as expected.

RI-3.2 Formulating queries

In the requirements in chapter 5 section 5.1.3 we have presented Key Performance Indicators (KPI) which were defined together with the industry partner. In the evaluation chapter 7 section 7.3 we have shown that our solution is capable of querying these KPIs. We have also added further process specific queries to the mix which can be queried with our solution. Table 8.1 shows the queries with the timings we have measured. All queries execute within fractions of a second, except for the query for the amount of signal operations per month and phase which returns the results for all months instead of a specific month. Since the result for past months is not expected to change, it could be cached to improve performance. Regardless, this is a significant improvement over the current approach, as it would take several days to collect and evaluate data at the industry partner for most queries.

Key Performance Indicator	Duration
amount of signals per month	75
amount of components per month	81
amount of signal operations per month	60
amount of signal operations per month and user	60
amount of signal operations per month and phase	8500
amount of notifications per user	1
average duration of notification per user	3
list of signals that trigger notifications	28
amount of status updates per month	71
amount of changes per month without status updates	174

Table 8.1: Durations of queries in ms for the KPIs. Each query returns the results for a specific month, except for query 5 which return the results for all months.

8.2 Limitations

In this section we discuss limitations of the implemented artifacts that were identified during the implementation and evaluation, moreover limitations of the case study that we have conducted. These limitations cannot be overcome in the scope of this work and have to be addressed in future work (see also section ”Future Work” in the Conclusion chapter 9).

8.2.1 Artifact Limitations.

First we will discuss the limitations of the implemented artifacts.

Only one active process per project supported. The implemented processes (check-in and status update) only allow for one active process at a time for a specific project. Running two check-in processes, two status update processes or one of each at the same time for the same project are not supported. This is because each process has to be seen as a long running transaction which has to commit before the next transaction can start. A process (p1) performs a read (r1) at the beginning of the transaction and in most cases a write (w1) at the end which is based on the result of r1. If another process (p2) would perform a write operation (w2) between r1 and w1 the result of r1 could be outdated due to the following reasons:

- w2 may have added new signals which were not returned by r1.
- w2 may have changed signals which were returned by r1 with different values.
- w2 may have deleted signals which were returned by r1.

If one or more of these scenarios happens, the decisions made by the process-stakeholders is based on outdated data which may lead to decision and a database write w1 with inconsistent data. Being able to have only one process running at a time is a drawback of the solution as a single process may run for several hours or even days if it is waiting on input from process stakeholders. This means that the project has to be locked while a process is running and only one engineer can perform a check-in while the others have to wait until the first one is finished, even if their process would finish faster. Depending on the size of the project team and the number of engineers who want to start processes this may get increasingly problematic. While small teams with two to three engineers may be able to circumvent this problem well by good coordination and communication it gets increasingly difficult the larger the teams get. There are no limitations for multiple projects; however, when there are ten different active projects, ten check-in processes can run in parallel and without interference. In early phases of a project this is less of a problem, because in early phases there are no notifications required. A check-in process can be completed by a single engineer and other engineers simply have to wait until he is finished. In later project phases it might be possible that less changes occur. Our test samples showed a decreasing amount of changes with each check-in, tough the amount of data samples we had is too small to draw any definitive conclusions of this kind. The test samples also lacked the signal states (and we had to assume some states to demonstrate that it works) and therefore we also cannot determine the phase in which the changes occurred.

There are different strategies to solve this problem, e.g. it could be overcome by some form of inter-process synchronization, where process p1 is able to react to writes performed by other processes, by re-reading from the database again and re-computing the change and notification sets. The engineer of p1 could then be notified about this and review the synchronized process and re-select his changes and notifications. Another possible solution could be on the project management level, e.g. by discussing and coordinating in weekly or even daily meetings who is going to check-in his data. However, further research is required on how this problem can be solved best and it cannot be tackled in the scope of this work.

Limited notification query capabilities. While NotificationSets and the contained signals are persisted in a binary format by Activiti's history, it would be more useful to have them stored

in a way that allows full query capabilities on all the properties in textual form. In the current implementation when we want to query for all signals that triggered notification, we have to first query the *ACT_GE_BYTEARRAY* table as shown in listing 7.21 and then deserialize the results into multiple NotificationSets which we then have to extract the signals from and which cause additional coding effort for the developer and computation effort for the system. These extracted signals then have to be cleaned of duplicates (e.g. when one signal triggered multiple notifications) or when wanting to count the number of signals that triggered notifications, filter out the unique object IDs from this list. It would be much easier to perform all these functions using SQL, just as in the other queries, but this is not possible as long as the data is stored in a binary format in the database.

Performance issues. While the new components that were created during this work did not cause any performance issues, some already existing and reused components do limit the usefulness of the whole solution by causing performance issues. By far the biggest bottleneck is the Engineering Database (EDB) which turned out to not scale well when performing updates. While the EDB may be useful for a research prototype due to its flexibility in defining and handling different model types, it is unusable with large amounts of models, as required in some research and practically all production scenarios, where ten thousands to millions of signals may be handled. Some of these performance issues were already noted before the creation of this work and as a result, the Engineering Database Index was created, which greatly improves product data query and storage performance while sacrificing some of the flexibility of the EDB.

Only single specific users supported for notification rules. Currently the notification rules in the notification mapping only allow the assignment to a single specific user for each rule and project. This can cause delays and problems if the user is notified and unable to respond to the notification. For example the user could be sick or on vacation.

8.2.2 Evaluation Limitations.

There are also a number of limitations with regards to the evaluation:

Limited Number of Check-Ins. The test data for our case study contained only three data samples for a single project. This makes it much less reliable to derive accurate project progress characteristics, like the number of changes over time and if the changes are increasing, decreasing or remaining approximately the same over time and how many changes occur in each project phase. We have shown however, that our solution could easily derive this, if the necessary data is at hand.

Limited Number of Signals. The amount of signals in the test sample reflected only a single component in a project. As a result the amount of signals was relatively low. The test samples contained a maximum of 1636 new, 796 changed and 452 deleted signals. In a full project there could be up to 100.000 signals. However, testing with a higher amount of signals is not realistic

anyway, since the performance bottleneck posed by the Engineering Database has to be resolved first.

No simultaneous Check-Ins tested. In a real world scenario there can be multiple check-ins at the same time. We did not test this, neither multiple simultaneous check-ins in a single project, nor multiple simultaneous check-ins in different projects.

Duration of User Tasks do not reflect real durations. We did not simulate any user task durations. While it would be possible to estimate the average time a user needs to complete a task, we did not have any real world data to base our assumptions on. Also it would be impractical to add extra durations to the test scenarios as it would artificially increase the duration of test executions with no real benefits and the drawback of having to wait some extra time for a test scenario to finish. While this could be circumvented by changing the system clock, we did not see it being worth the effort.

CHAPTER 9

Conclusion and Future Work

In this chapter we will give our conclusion about this work, spanning the arc to the introduction and also provide some ideas for future work which were brought up during the creation of this thesis and could not be solved within the scope of this thesis but should be at least investigated in the future.

9.1 Conclusion

In heterogeneous engineering environments multiple tools from different disciplines are used which also provide limited interoperability, causing noticeable overheads in day-to-day work processes for all involved stakeholders, such as engineers, manufacturers, customers and project managers. These work processes involve manual synchronization efforts which are time consuming and prone to errors. Insights into ongoing and completed processes are difficult to obtain, since most communication and work synchronization is done in different systems and via e-mail.

In this work we have demonstrated a BPMN-process based approach using the Activiti process engine, based on the input of an industry-partner in the area of power plant engineering and provided a process implementation in the environment of the Engineering Service Bus (EngSB) [Biffl and Schatten, 2009], including a test framework which can be parameterized and that allows the simulation of various scenarios. Extending the approach of the Project Observation and Analysis Framework (POAF) [Sunindyo, 2012], we also incorporated product data in our analysis enabling combined queries over both process and product data. During process execution process data is recorded which allows to measure various process metrics, such as run times of processes, automated tasks and user tasks, as well as the number of changes, involved users and notifications. Together with the industry partner we have identified several key performance indicators (KPI) which we could successfully measure with our solution as the evaluation showed.

9.1.1 Improvements for Stakeholders

Finally we want to highlight the improvements the stakeholders can expect integrating our solution in their work process:

Engineers With our solution engineers can reduce their effort for data synchronization, allowing for faster and more frequent synchronizations. This means that they can work with more up-to-date product data and it also reduces the risk of errors compared to manual synchronization. They no longer have to send e-mails to other collaborators (engineers from other disciplines, manufacturers, customers) informing them of their changes as the system is configured to detect if and which collaborators should be notified and also provides decision mechanisms for them. Additionally engineers are able to transition product components through the project states defined by the project managers. The states can be applied and stored in the system and are available to be used in project statistics.

Customers and Manufacturers The customers' and manufacturers' decision effort is reduced as they can now submit their decisions about product changes directly in the system. They no longer have to communicate exclusively by e-mail with the engineers when changes occur. Their decisions are now also automatically recorded by the system and available to be used in project statistics.

Project Managers Our solution enables project managers to more effectively monitor projects by having access to data collected during process executions from other stakeholders in the project. They gain deeper insights into processes and the development of the product, which they could not have obtained before, or with an effort of at least several days. We have proven this with various KPIs for which we have presented queries to measure them within fractions of seconds.

9.2 Future work

During this work we identified several points, which could be interesting for future work.

User Interface. Our contribution did not include a user interface (UI) or even a UI-prototype, as only general concepts and the potential of our solution were tested. Furthermore, while former UI-implementations were based on Apache Wicket[1] a new version of the OpenEngSB was planned and implemented parallel to this work, and a decision about the future UI-framework had not been made yet.

Separate Storage for Notification Data. Activiti stores all objects in a serialized, binary format in the *ACT_GE_BYTEARRAY* table. While this is good because it can also store the notification data accumulated during process execution, it is not possible to directly query parts of the data. Therefore we suggest a separate storage that supports such queries.

[1] Apache Wicket: http://wicket.apache.org

Support for Standardized Data Exchange Formats. Currently tool data is directly integrated in a non-standardized way via domain models in the EngSB. During the last years various standardized formats have been developed, like AutomationML [Drath,], CAEX [Schleipen et al., 2008], PLCopen [Riedl et al., 2009] and COLLADA [Barnes and Levy Finch, 2008] which could also be used. This would also require new product data storage components which would replace the EDB/EDBI combination used in this work.

Memory footprint measurement. Our evaluation did not include the measurement of a memory footprint during process execution. To fully understand all performance factors, tests should not only include the duration of process executions but also include a memory footprint. Some independent tests have shown that the decreasing performance of the Engineering Database (EDB) correlates with an increasing usage of memory. Artificial increases and decreases of the available memory had a noticeable impact on the performance. Having a memory footprint at hand, would help to identify similar problems in other components.

Process mining export component. Event logs used for process mining tasks in this work were crafted manually. An auxiliary service that helps with converting process data, collected with Activiti to the XES-format for analysis, with process mining tools like ProM, will be indispensable if process mining will be employed in other future work, even in a finished product.

Social Network Algorithm with Parallel Task Support. Another interesting approach for project managers could be social networks which visualize the interactions between different stakeholders and additionally the frequency of these interactions. We have already performed first tests with existing social network process mining algorithms, but they are challenged by the parallel subprocess we use in our check-in process. Unfortunately, independent and parallel interactions are detected as subsequent interactions which does not reflect the reality. The development of a social network process mining algorithm that supports independent parallel tasks would be quite useful.

Investigation of Extendability of Processes and Adaptability to Process change. It should be investigated how well our solution supports the extension and adaption of real world processes. This includes the addition, removal or moving of process activities, including also changes to the whole test setup we have created. During the development of this thesis we already had to perform such extensions and adaptions as new requirements for the processes had to be considered. However, we did not measure or document these adaptions in a systematic way, since we were only dealing with one major use case in this thesis.

Bibliography

[Barnes and Levy Finch, 2008] Barnes, M. and Levy Finch, E. (2008). COLLADA – Digital Asset Schema Release 1.5.0 Specification. *Elements*, (March).

[Bayraktar, 2011] Bayraktar, I. (2011). The Business Value of Process Mining Bringing It All Together. Master's thesis.

[Bergenthum et al., 2007] Bergenthum, R., Desel, J., Lorenz, R., and Mauser, S. (2007). Process Mining Based on Regions of Languages. In *Business Process Management*, pages 375–383.

[Biffl and Schatten, 2009] Biffl, S. and Schatten, A. (2009). A platform for service-oriented integration of software engineering environments. In *SoMeT*, pages 75–92.

[Biffl et al., 2009] Biffl, S., Schatten, A., and Zoitl, A. (2009). Integration of heterogeneous engineering environments for the automation systems lifecycle. In *2009 7th IEEE International Conference on Industrial Informatics*, pages 576–581. IEEE.

[Biffl et al., 2010] Biffl, S., Sunindyo, W. D., and Moser, T. (2010). A Project Monitoring Cockpit Based On Integrating Data Sources in Open Source Software Development. In *Proc. Twenty-Second International Conference on Software Engineering and Knowledge Engineering (SEKE 2010)*, pages 620–627.

[Chappell, 2004] Chappell, D. A. (2004). *Enterprise Service Bus*. O'Reilly.

[Cook and Wolf, 1999] Cook, J. E. and Wolf, A. L. (1999). Software process validation: Quantitatively measuring the correspondence of a process to a model. *ACM Trans. Softw. Eng. Methodol.*, 8(2):147–176.

[Databases et al., 2003] Databases, P., Kiepuszewski, B., Barros, A. P., and Dogac, A. (2003). Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51.

[Date and Darwen, 1997] Date, C. J. and Darwen, H. (1997). *A Guide to the SQL Standard (4th Ed.): A User's Guide to the Standard Database Language SQL*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[De Medeiros et al., 2007] De Medeiros, a. K. a., Weijters, a. J. M. M., and van Der Aalst, W. M. P. (2007). Genetic process mining: An experimental evaluation. *Data Mining and Knowledge Discovery*, 14(2):245–304.

[Doan et al., 2004] Doan, A., Noy, N. F., and Halevy, A. Y. (2004). Introduction to the Special Issue on Semantic Integration. *SIGMOD Record*, 33(4):11–13.

[Drath,] Drath, R. *Datenaustausch in der Anlagenplanung mit AutomationML: Integration von CAEX, PLCopen XML und COLLADA*.

[Fay et al., 2013] Fay, A., Biffl, S., Winkler, D., Drath, R., and Barth, M. (2013). A method to evaluate the openness of automation tools for increased interoperability. In *Proceedings of the 39th Annual Conference of the IEEE Industrial Electronics Society (IECON 2013)2*, pages 6842–6847, Vienna, Austria. IEEE.

[Günther and Verbeek, 2014] Günther, C. and Verbeek, E. (2014). Xes standard definition. *Fluxicon Process Laboratories*, pages 0–24.

[Havey, 2005] Havey, M. (2005). *Essential Business Process Modeling*.

[Hevner, 2007] Hevner, A. R. (2007). A Three Cycle View of Design Science Research A Three Cycle View of Design Science Research. 19(2).

[Hevner et al., 2004] Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105.

[Kalenkova et al., 2014] Kalenkova, A. a., Leoni, M. D., and van der Aalst, W. M. P. (2014). Discovering , Analyzing and Enhancing BPMN Models Using ProM. *CEUR Workshop Proceedings*, 1295:36–41.

[Kalenkova et al., 2015] Kalenkova, A. a., van der Aalst, W. M. P., Lomazova, I. a., and Rubin, V. (2015). Process Mining Using BPMN : Relating Event Logs and Process Models.

[Kloppmann et al., 2007] Kloppmann, M., Koenig, D., Leymann, F., Pfau, G., Rickayzen, A., Schmidt, P., and Trickovic, I. (2007). *WS-BPEL Extension for People*.

[Leach et al., 2005] Leach, P., Mealling, M., and Salz, R. (2005). A universally unique identifier (uuid) urn namespace. RFC 4122.

[Mayerhuber, 2011] Mayerhuber, F. (2011). Versioning of tool data in service orientated architectures. Technical report, TU Wien.

[Mordinyi et al., 2015] Mordinyi, R., Winkler, D., Waltersdorfer, F., Scheiber, S., and Biffl, S. (2015). Integrating Heterogeneous Engineering Tools and Data Models : A Roadmap for Developing Engineering System Architecture Variants. pages 1–19.

[Moser, 2009] Moser, T. (2009). Semantic Integration of Engineering Environments Using an Engineering Knowledge Base. *Faculty of Informatics*, PhD Thesis(0):226.

[Moser et al., 2011] Moser, T., Mordinyi, R., Winkler, D., and Biffl, S. (2011). Engineering project management using the Engineering Cockpit: A collaboration platform for project managers and engineers. In *IEEE 9th International Conference on. Industrial Informatics (INDIN'2011)*, pages 579–584.

[Mulyar, 2005] Mulyar, N. a. (2005). Pattern-based Evaluation of Oracle-BPEL (v.10.1.2). *Technology*.

[Murata, 1989] Murata, T. (1989). Petri nets: properties, analysis and applications.

[OASIS, 2007] OASIS (2007). *Business Process Execution Language 2.0 (WS-BPEL 2.0)*.

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

[OMG, 2013] OMG (2013). Business Process Model and Notation (BPMN) Version 2.0.2. 95(December).

[Peffers et al., 2007] Peffers, K. E. N., Tuunanen, T., Rothenberger, M. a., and Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3):45–77.

[Pieber and Biffl, 2010] Pieber, A. and Biffl, S. (2010). Flexible Engineering Environment Integration for (Software+) Engineering Teams. *Proceedings of the Junior Scientist Conference 2010*, 2010(0):49–50.

[Pircher, 2013] Pircher, M. (2013). An ontology-based approach for transformer generation in a multi-disciplinary engineering environment. Technical report, TU Wien.

[Rademakers, 2012] Rademakers, T. (2012). *Activiti in Action : Executable business processes in BPMN 2.0*. Manning Publications, Shelter Island, NY, first edition.

[Rausch, 2014] Rausch, T. (2014). Efficient Querying of Versioned Tool Data in Data Integration Environments. Technical report, TU Wien.

[Recker, 2010] Recker, J. (2010). Opportunities and constraints: the current struggle with BPMN. *Business Process Management Journal*, 16(1):181–201.

[Riedl et al., 2009] Riedl, M., Baier, T., Atali-ringot, M., and Powers, L. (2009). PLCopen PLCopen Technical Committee 6 XML Formats for IEC 61131-3 PLCopen.

[Schatten et al., 2006] Schatten, A., Mustofa, K., Biffl, S., Tjoa, A. M., and Wahyudin, D. (2006). Introducing "health" perspective in open source web-engineering software projects, based on project data analysis. In *Proceedings of the International Conference on Information Integration, Web-Applications and Services*. Austrian Computer Society. Vortrag: IIWAS International Conference on Information Integration, Web-Applications and Services, Yogyakarta Indonesien; 2006-12-04 – 2006-12-06.

[Schleipen et al., 2008] Schleipen, M., Drath, R., and Sauer, O. (2008). The system-independent data exchange format CAEX for supporting an automatic configuration of a production monitoring and control system. *IEEE International Symposium on Industrial Electronics*, pages 1786–1791.

[Shafranovich, 2005] Shafranovich, Y. (2005). Common format and mime type for comma-separated values (csv) files. RFC 4180.

[Song and van der Aalst, 2008] Song, M. and van der Aalst, W. M. P. (2008). Towards comprehensive support for organizational mining. *Decision Support Systems*, 46(1):300–317.

[Sunindyo et al., 2011a] Sunindyo, W., Moser, T., Winkler, D., Mordinyi, R., and Biffl, S. (2011a). Workflow validation framework in distributed engineering environments. In Meersman, R., Dillon, T., and Herrero, P., editors, *On the Move to Meaningful Internet Systems: OTM 2011 Workshops*, volume 7046 of *Lecture Notes in Computer Science*, pages 236–245. Springer Berlin Heidelberg.

[Sunindyo, 2012] Sunindyo, W. D. (2012). Project Observation and Analysis in Heterogeneous Software & Systems Development Environments. page 161.

[Sunindyo et al., 2011b] Sunindyo, W. D., Moser, T., Winkler, D., and Biffl, S. (2011b). Analyzing OSS Project Health with Heterogeneous Data Sources. *International Journal of Open Source Software and Processes*, 3(4):1–23.

[Sunindyo et al., 2012] Sunindyo, W. D., Moser, T., Winkler, D., and Dhungana, D. (2012). Improving Open Source Software Process Quality Based on Defect Data Mining. In *Proceedings of Software Quality Days 2012, Research Track*, pages 84–102.

[Sunindyo et al., 2013] Sunindyo, W. D., Moser, T., Winkler, D., and Mordinyi, R. (2013). Project Progress and Risk Monitoring in Automation Systems Engineering. In *Proc. 5th Software Quality Days. Lecture Notes in Business Information Processing 133 (SWQD 2013)*, pages 30–54, Vienna, Austria. Springer Berlin Heidelberg.

[Sunindyo et al., 2011c] Sunindyo, W. D., Moser, T., Winkler, D., Mordinyi, R., and Biffl, S. (2011c). Workflow validation framework in distributed engineering environments. In *OTM Workshops*, pages 236–245.

[van der Aalst et al., 2012] van der Aalst, W., Adriansyah, A., and Van Dongen, B. (2012). Replaying history on process models for conformance checking and performance analysis. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(2):182–192.

[van Der Aalst et al., 2004] van Der Aalst, W., Weijters, T., and Maruster, L. (2004). Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142.

[van der Aalst, 2011] van der Aalst, W. M. P. (2011). *Process Mining; Discovery, Conformance and Enhancement of Business Processes*. Springer Berlin Heidelberg: Berlin, Heidelberg, Berlin, Heidelberg.

[van der Aalst et al., 2011] van der Aalst, W. M. P., Adriansyah, A., de Medeiros, A. K. A., Arcieri, F., Baier, T., Blickle, T., Bose, R. P. J. C., van den Brand, P., Brandtjen, R., Buijs, J. C. A. M., Burattin, A., Carmona, J., Castellanos, M., Claes, J., Cook, J., Costantini, N., Curbera, F., Damiani, E., de Leoni, M., Delias, P., van Dongen, B. F., Dumas, M., Dustdar, S., Fahland, D., Ferreira, D. R., Gaaloul, W., van Geffen, F., Goel, S., Günther, C. W., Guzzo, A., Harmon, P., ter Hofstede, A. H. M., Hoogland, J., Ingvaldsen, J. E., Kato, K., Kuhn, R.,

Kumar, A., Rosa, M. L., Maggi, F. M., Malerba, D., Mans, R. S., Manuel, A., McCreesh, M., Mello, P., Mendling, J., Montali, M., Nezhad, H. R. M., zur Muehlen, M., Munoz-Gama, J., Pontieri, L., Ribeiro, J., Rozinat, A., Pérez, H. S., Pérez, R. S., Sepúlveda, M., Sinur, J., Soffer, P., Song, M., Sperduti, A., Stilo, G., Stoel, C., Swenson, K. D., Talamo, M., Tan, W., Turner, C., Vanthienen, J., Varvaressos, G., Verbeek, E., Verdonk, M., Vigo, R., Wang, J., Weber, B., Weidlich, M., Weijters, T., Wen, L., Westergaard, M., and Wynn, M. T. (2011). Process mining manifesto. In *Business Process Management Workshops (1)*, pages 169–194.

[van der Aalst et al., 2005] van der Aalst, W. M. P., Dumas, M., ter Hofstede, A. H. M., Russell, N., Verbeek, H. M. W., and Wohed, P. (2005). Life After BPEL? *Formal Techniques for Computer Systems and Business Processes*, 3670:35–50.

[van Der Aalst et al., 2005] van Der Aalst, W. M. P., Reijers, H. a., and Song, M. (2005). Discovering social networks from event logs. *Computer Supported Cooperative Work*, 14(6):549–593.

[van Dongen et al., 2005] van Dongen, B., Alves de Medeiros, a. K., Verbeek, H. M. W., Weijters, a. J. M. M., and van der Aalst, W. M. P. (2005). The ProM framework: A new era in process mining tool support. *Application and Theory of Petri Nets 2005*, (3536):444–454.

[van Dongen et al., 2007] van Dongen, B., Busi, N., Pinna, G. M., and van der Aalst, W. M. P. (2007). An Iterative Algorithm for Applying the Theory of Regions in Process Mining. *Workshop on Formal Aspects of Business Processes and Web Services*.

[Wahyudin et al., 2007] Wahyudin, D., Mustofa, K., Schatten, A., Biffl, S., and Tjoa, A. M. (2007). Monitoring the “health” status of open source web-engineering projects.

[Waltersdorfer et al., 2010] Waltersdorfer, F., Moser, T., Zoitl, A., and Biffl, S. (2010). Version Management and Conflict Detection Across Heterogeneous Engineering Data Models. In *Proc. 8th IEEE International Conference on Industrial Informatics (INDIN 2010)*.

[Weijters and van der Aalst, 2003] Weijters, A. and van der Aalst, W. (2003). Rediscovering Workflow Models from Event-Based Data using Little Thumb. *Integrated ComputerAided Engineering*, 10:151–162.

[Weijters and Ribeiro, 2011] Weijters, A. J. M. M. and Ribeiro, J. T. S. (2011). Flexible Heuristics Miner (FHM). *Computational Intelligence and Data Mining (CIDM), 2011 IEEE Symposium on*, pages 310–317.

[Winkler et al., 2011] Winkler, D., Moser, T., Mordinyi, R., Sunindyo, W. D., and Biffl, S. (2011). Engineering object change management process observation in distributed automation systems projects. In *18th EuroSPI Conference*.

[Winkler et al., 2014] Winkler, D., Schonbauer, M., and Biffl, S. (2014). Towards Automated Process and Workflow Management: A Feasibility Study on Tool-Supported and Automated Engineering Process Modeling Approaches. *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 102–110.

[Wohed et al., 2006] Wohed, P., van der Aalst, W. M. P., Dumas, M., ter Hofstede, a. H. M., and Russell, N. (2006). On the suitability of BPMN for business process modelling. *Proceedings of the 4th International Conference, BPM 2006*, pages 161–176.

[Yin et al., 2009] Yin, J., Chen, H., Deng, S., Wu, Z., and Pu, C. (2009). A Dependable ESB Framework for Service Integration. *Internet Computing, IEEE*, 13(2):26–34.

APPENDIX A

Used Technologies

This chapter provides a list of all technologies used during the creation of this work. The intention is to provide a complete overview for future reference and to help with reproducibility of the results presented. Thanks to all the companies providing free software or free student licenses.

Tool	Version	URL
Java	1.7.0	`http://www.java.com`
Maven	3.1.1	`http://maven.apache.org`
Git	1.8.3	`http://git-scm.com`
Inkscape	0.91	`https://inkscape.org`
Mendeley	1.14	`https://www.mendeley.com`
MiKTeX	2.9.5105	`http://miktex.org`
ProM	6.5	`http://www.promtools.org`
Texmaker	4.4.1	`http://www.xm1math.net/texmaker`
Visual Paradigm	11.1	`http://www.visual-paradigm.com`
Visio	2010	`http://microsoft.com`
draw.io	5.2.6.3	`http://www.draw.io`
Intellij IDEA	14.1.4	`https://www.jetbrains.com/idea/`
Squirrel SQL	3.6	`http://squirrel-sql.sourceforge.net/`

Table A.1: The tools used within the research prototype

Library	Version	URL
Guava	15.0	http://code.google.com/p/guava-libraries
Activiti	5.17.0	http://activiti.org
Wicket	6.9.0	http://wicket.apache.org
Junit	4.12	http://junit.org/
Mockito	1.9.5	http://code.google.com/p/mockito/
OpenEngSB	3.x	http://openengsb.org
pax-wicket		
H2	1.3.173	http://www.h2database.com
SLF4J	1.7.5	http://www.slf4j.org/
Log4J	1.2.17	http://logging.apache.org/log4j
Commons-Codec	1.8	http://commons.apache.org/codec
Commons-IO	2.4	http://commons.apache.org/io
Commons-Lang3	3.1	http://commons.apache.org/lang

Table A.2: The java libraies used within the research prototype

APPENDIX B

Test Environment

In this chapter we present the environment used to create and evaluate the artifacts. All test results were obtained on an HP ELiteBook 8440p with the following system specs:

- OS: ArchLinux 4.2.5-1-ARCH
- CPU: Intel i7-620M (2.67GHz)
- GPU: Nvidia NVS-3100M (@135MHz)
- RAM: 8GB
- SSD: Samsung 830 256GB

List of Figures

2.1 EngSB Domain and Connector Concept plus Workflow Engine Service. [Biffl and Schatten, 2009] 10
2.2 Project Observation and Analysis Framework by Sunindyo [Sunindyo, 2012] . . . 11
2.3 Workflow Observation Process [Sunindyo, 2012] 13
2.4 The process mining framework. [van der Aalst, 2011] 18
2.5 De jure petri net based on the event log in table 2.6. 20
2.6 Three basic types of process mining. [van der Aalst et al., 2011]. 21

3.1 Process and Product Analysis Framework based on [Sunindyo, 2012] 24

4.1 The design science research cycles [Hevner, 2007] 28
4.2 The design science research methodology [Peffers et al., 2007] 29

5.1 Identified Use Cases based on the requirements of the industry partner. 33
5.2 Signal Change Management Workflow [Winkler et al., 2011] 34
5.3 Signal Change Management Process [Winkler et al., 2014] 34
5.4 Signal Comparison Subtask [Winkler et al., 2014] 35
5.5 Human Interaction Subtask [Winkler et al., 2014] 35

6.1 The Component Architecture grouped into Process focused, Product focused and auxiliary components. 42
6.2 BPMN diagram of the implemented process showing the control and data flow. The data flow only shows the flow of signal data and not for all params. 46
6.3 The subprocess handling the notifications. 46
6.4 The process for status updates. 52
6.5 The class diagram of the NotificationMapper and associated classes. 57
6.6 The test model for parameterized testing. 60
6.7 The EER model of the Activiti history. 65
6.8 The EER model of the EDBI. 66

7.1 Overview of the first evaluation test scenario. 68
7.2 Overview of the second evaluation test scenario. 73
7.3 Overview of the first case-study scenario. 75
7.4 The number of signal operations detected in each process. The exact values can be seen in table 7.6. 76

7.5 Overview of the second case-study scenario. 82
7.6 The number of signal operations detected in each process. The exact values can be seen in table 7.8. 83
7.7 Discovered process as path through the processes in 6.2 and 6.3. 87
7.8 Discovered Handover-of-work network. 88
7.9 Discovered Similar-Task Social Network. 88

List of Tables

2.1 BPMN events [OMG, 2011] 14
2.2 BPMN activities [OMG, 2011] 15
2.3 BPMN flows [OMG, 2011] 16
2.4 BPMN Gateways [OMG, 2011] 16
2.5 BPMN other elements [OMG, 2011] 17
2.6 An event log fragment 19

5.1 Signal States 36
5.2 Ticket types 36
5.3 The Notification Table 37

6.1 Activiti History Levels [Rademakers, 2012] 44
6.2 The input parameters for the Upload Product Data Task 47
6.3 The parameters for the Transform Product Data Task 47
6.4 The parameters for the Query Existing Product Data Task 47
6.5 The parameters for the Compare Changes Task 48
6.6 The parameters for the Check Notifications Task 48
6.7 The input parameters for the Select Changes and Notifications Task 49
6.8 The parameters for the Handle Notifications Subprocess 49
6.9 The input parameters for the Decide Task 50
6.10 The parameters for the Apply Decision Task 50
6.11 The parameters for the Merge Task 51
6.12 The parameters for the Persist Task 51
6.13 The input parameters for the Set Query Parameters Task 52
6.14 The input parameters for the Select Status Task 53
6.15 The parameters for the Apply Status Task 53
6.16 The sql standard functions used. 66

7.1 Signal data for the first check-in process 69
7.2 Signal data for the second check-in process 69
7.3 Result of EDB query after the first process execution. 70
7.4 Result of EDB query after the second process execution. 72
7.5 Result of EDB query after the status update process execution. 74

7.6 The number of signal operations detected in each process and the total amount of signals in the product database after each run. 76
7.7 The process elements and their durations in ms as recorded by Activiti. 77
7.8 The number of signal operations detected in each process and the total amount of signals in the product database after each run. 83
7.9 The process elements and their durations in ms as recorded by Activiti. 84

8.1 Durations of queries in ms for the KPIs. Each query returns the results for a specific month, except for query 5 which return the results for all months. 94

A.1 The tools used within the research prototype 109
A.2 The java libraies used within the research prototype 110

List of Acronyms

AutomationML Automation Markup Language

API Application Programming Interface

ASB Automation Service Bus

BIM Building Information Modeling

BPEL WS-Business Process Execution Language

BPMN Business Process Model and Notation

CAD Computer Aided Design

CAEX Computer Aided Engineering Exchange

CDL-Flex Christian Doppler Laboratory for Software Engineering Integration for Flexible Automation Systems

COLLADA Collaborative Design Activity

CSV Comma-Separated Values

EAI Enterprise Application Integration

ESB Enterprise Service Bus

GUID Globally Unique Identifier

GUI Graphical User Interface

IEEE Institute of Electrical and Electronics Engineers

ISO International Organization for Standardization

J2EE Java Platform Enterprise Edition

JPA Java Persistence API

JSON Java Script Object Notation

JVM Java Virtual Machine

KKS Kraftwerk Kennzeichensystem - Identification System for Power Stations

MXML Mining eXtensible Markup Language

OASIS Organization for the Advancement of Structured Information Standards

OMG Object Management Group

OpenEngSB Open Engineering Service Bus

OPM Output Point Module

ORM Object-Relational-Mapping

OSGi OSGi Alliance (formerly Open Services Gateway Initiative

PAIS Process-Aware Information System

PDF Portable Document Format

PLC Programmable Logic Controller

POAF Project Observation and Analysis Framework

SAX Simple API for XML

SA-MXML Semantically Annotated MXML

SCM Source Code Management

SVN Subversion

SQL Structured Query Language

UML Unified Modeling Language

URL Uniform Resource Locator

VM Virtual Machine

XES eXtensible Event Stream

XML Extensible Markup Language

XPath XML Path Language