FH JOANNEUM - University of Applied Sciences

Process Mining of Business Process Choreographies

Master thesis

submitted at the Master Degree Programme Information Management

for the degree „Diplomingenieur für technisch-wissenschaftliche Berufe“

Author:

Matthias Eduard Geisriegler, BSc

Supervisor:

FH-Prof. Mag. Dr. Robert Singer

Graz, 2017

FH | JOANNEUM
University of Applied Sciences

Declaration

I hereby declare that the present master's thesis was composed by myself and that the work contained herein is my own. I also confirm that I have only used the specified resources. All formulations and concepts taken verbatim or in substance from printed or unprinted material or from the Internet have been cited according to the rules of good scientific practice and indicated by footnotes or other exact references to the original source.

The present thesis has not been submitted to another university for the award of an academic degree in this form. This thesis has been submitted in printed and electronic form. I hereby confirm that the content of the digital version is the same as in the printed version.

I understand that the provision of incorrect information may have legal consequences.

_____

Matthias Eduard Geisriegler, BSc Graz, 18/09/2017

Contents

List of Figures

Listings

Abbreviations

BPM	Business Process Management
CSV	Comma-Seperated Values
FSM	Finite-State Machine
JSON	JavaScript Object Notation
S-BPM	Subject-Oriented Business Process Management
SBD	Subject Behaviour Diagram
SID	Subject Interaction Diagram
MXML	Mining eXtensible Markup Language
oWFN	Open Workflow Nets
OWL	Web Ontology Language
PNML	Petri Net Markup Language
REST	Representational State Transfer
XES	Extensible Event Stream
XML	Extensible Markup Language

Abstract

Business Process Management (BPM) and business process modelling contribute to the success of organisations. Apart from process orchestration modelling, process choreography modelling has become more and more important due to the increasing collaboration and interaction between business partners. In many cases, processes modelled in the most common modelling notation BPMN (Business Process Management Notation) focus either on the orchestration or the choreography. The subject-oriented notation S-BPM (Subject-Oriented Business Process Management), by contrast, combines the concepts of process orchestration and choreography. However, process models mostly illustrate an idealised view of the process. For that reason Process Mining techniques were developed. These techniques can be used to discover actual and to improve existing process models. However, common Process Mining methods focus on process orchestrations.

This thesis aims at creating solutions for the generation of process choreographies by means of Process Mining. Moreover, this work is to show how Process Mining methods can be aligned with the principles of S-BPM. A prototypical algorithm is presented and implemented into the existing S-BPM Execution Platform, dedicated to support Process Mining efforts in terms of discovering process choreographies.

First, the principles of business process modelling and different modelling notations are discussed. Second, the idea of Process Mining is presented and an event log structure suitable for reconstructing process choreographies is developed. After illustrating the features of the S-BPM Modelling and Execution Platform briefly, the adaptations and implementations are presented. All implementations in this thesis follow the existing microservice architecture of the platform, based on Spring Boot and Java 8. A new service is created, which is responsible for logging process events and for transforming discovered process orchestrations into process choreographies. Finally, all features implemented are tested and the discovery of a sample process choreography is illustrated.

Based on the findings and the implementations of this thesis, it is shown how business process choreographies can be discovered by using Process Mining methods. Furthermore, it is documented how the existing S-BPM platform is extended by features that support the discovery of process choreographies.

The main conclusion drawn from this thesis is, that it is possible to generate process choreographies by means of Process Mining. Additionally, it is proved that the advantages of Process Mining and the benefits of S-BPM can be combined.

Kurzfassung

Die Modellierung und das Management von Geschäftsprozessen tragen wesentlich zum Erfolg eines Unternehmens bei. Durch die steigende Zusammenarbeit zwischen Geschäftspartnern wird es zunehmend auch wichtiger neben Prozessorchestrierungen auch Prozesschoreographien zu modellieren. Prozesse, die mittels BPMN (Business Process Management Notation) konstruiert werden, bilden häufig entweder nur die Orchestrierung oder die Choreographie ab. Der subjekt-orientierte Ansatz der S-BPM (Subject-Oriented Business Process Management) Notation hingegen, verbindet die Konzepte der Prozessorchestrierung und -choreographie miteinander. Prozessmodelle bilden meist jedoch nur eine idealisierte Sicht eines Prozesses ab. Um dieses Problem zu lösen, können Process Mining Methoden angewandt werden, die die tatsächlichen Abläufe eines Prozesses darstellen und bestehende Modelle verbessern können. Allerdings beziehen sich die bewährtesten Process Mining Methoden auf die Entdeckung und Verbesserung von Prozessorchestrierungen.

Ziel dieser Arbeit ist es, Lösungen zur Generierung von Prozesschoreographien mittels Process Mining zu entwickeln und darzustellen. Außerdem soll gezeigt werden, wie Process Mining Methoden mit den Prinzipien der subjekt-orientierten Prozessmodellierung im Zusammenspiel funktionieren können. Zu diesem Zweck wird ein prototypischer Algorithmus zur Entdeckung von Prozesschoreographien mittels Process Mining entwickelt und in die bestehende S-BPM Modellierungs- und Ausführungsplattform implementiert.

Zu Beginn werden die Prinzipien der Geschäftsprozessmodellierung und verschiedene Notationen erläutert. Danach werden die Grundlagen des Process Minings dargestellt und eine für Prozesschoreographien geeignete Event Log Struktur wird entwickelt. Neben einer kurzen Beschreibung und Darstellung der bestehenden Funktionalitäten der S-BPM Modelleriungs- und Ausführungsplattform, werden die Anpassungen und Implementierungen, die im Zuge dieser Masterarbeit erarbeitet werden, präsentiert. Alle neuen Funktionalitäten fügen sich nahtlos in die bestehende Mikroservice Architektur, basierend auf Spring Boot und Java 8, ein. Ein neuer Service, der Prozessevents während der Ausführung dokumentiert, wird implementiert. Außerdem wird dieser Dienst jene Aufgaben übernehmen, die notwendig sind, um entdeckte Prozessmodelle in Prozesschoreographien zu transformieren. Schlussendlich werden alle Funktionalitäten anhand einer beispielhaften Prozesschoreographie getestet und dargestellt.

Anhand der Ergebnisse und Implementierungen dieser Arbeit wird gezeigt, wie Prozesschoreographien mittels Process Minings entdeckt werden können. Außerdem wird dokumentiert, in welcher Form und mit welchen Funktionalitäten die S-BPM Platform diese Aufgabe unterstützen kann.

Abschließend kann gesagt werden, dass es möglich ist, Prozesschoreographien mittels Process Mining zu generieren. Außerdem kann gezeigt werden, dass Process Mining und die Prozessmodellierung mittels S-BPM erfolgreich kombiniert werden können.

1. Introduction

Business process management (BPM) is key for the success of organisations, as it refers to the continuous design of business processes. [6, p. 2]. Organisations design processes to perform and coordinate activities consistently in a convenient way for the customer, which results in a competitive advantage. The purpose of today's IT systems is to support business processes, making them agile and adaptive. However, it is unclear to what extent modern IT infrastructure enables organisations to gain competitive advantage [15, pp. 48-51].

Business process choreography modelling becomes more and more important since business collaboration between companies has increased. The interaction between business partners may involve a high number of messages exchanged. By modelling process choreographies, the internal view is abstracted and the focus is put on the conversation and communication between the partners [2, pp. 280-281].

The aim of designing business process models is to implement the strategy goals of a company. This is done by translating process descriptions into a graphical representation. Most commonly, BPMN (Business Process Management Notation) is the first choice for process modelling. External or internal consultants analyse, design and document the important processes of a company. Nevertheless, even perfectly drawn processes will mostly not be adapted to the reality. Moreover, it is difficult for people who are actually working with a process to design or understand a BPMN process model, since deep knowledge is needed for using this notation. Nevertheless, literature concerning business process management recommends BPMN for process modelling [15, pp. 51-52]. Studies showed that processes modelled in BPMN either focus on the orchestration or on the choreography, not both [1, p. 239].

Unlike other notations, Subject-Oriented Business Process Management (S-BPM) uses a small set of four symbols to model business processes. Moreover, S-BPM is message- and behaviour-oriented and processes can be modelled with natural language by using a subject, a predicate and an object. This approach makes it easier for the people who are actually involved in the processes to design and understand process models. Using the S-BPM methodology, organisations can possibly bridge the gap between business processes and information technology [15, pp. 54-56]. S-BPM coordinates parallel activities of multiple subjects via messages. Therefore, the subject orientation of S-BPM combines the concepts of process orchestration and choreography [6, p. 200].

As a matter of fact, hand-made models mostly illustrate an idealised version of the process and do not represent the reality. When process models are not aligned with reality, they provide little value for end users [17, p. 30]. This is where Process Mining comes in.

Today's business processes rely heavily on information systems that generate a huge amount of events. These events, often summarised by the term "Big Data", can be recorded and analysed. Moreover, event data can be used for process analysis by gaining information about bottlenecks, problems and policy violations as well as by

generating countermeasures and optimising processes. Therefore, companies and organisations should strive hard for generating value from their event data produced by their information systems. This can be achieved by Process Mining which "*aims to exploit event data in a meaningful way*" [17, pp. 3-5].

Process Mining is a sub-discipline of Data Science, which includes extraction, preparation, exploration and transformation of structured or unstructured data aiming at generating value. The growing amount of event data results in the four challenges of Big Data, the so called four V's: Volume, which refers to the incredible sizes of data. Velocity, which stands for the speed of change. Variety, which refers to the different types and sources of data. Veracity, which stands for the dimension of how uncertain or trustworthy the data is [17, pp. 9-12].

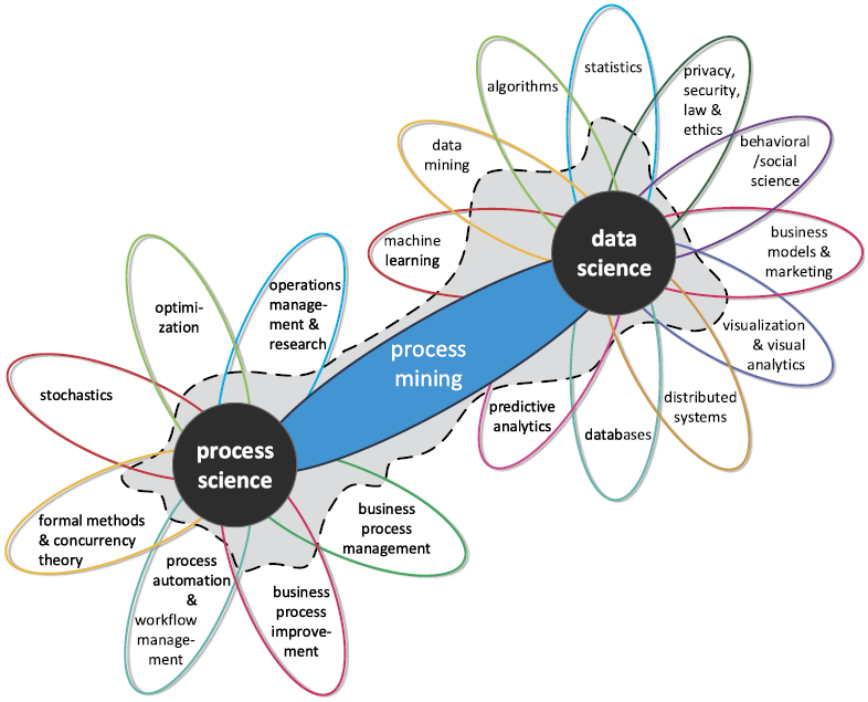

Figure 1.1.: Process Mining: the link between data science and process science [17, p. 18]

Figure 1.1 shows that the sub-disciplines of Data Science are overlapping. Process Mining is influenced by machine learning and data mining and builds a process-oriented perspective to those disciplines [17, pp. 12-13]. According to [9, p. 1] Data Mining is "*the analysis of (often large) data sets to find unsuspected relationships and to summarize the data in novel ways that are both understandable and useful to the data owner*". Machine Learning, however, is stated to be a discipline where computers learn from experience based on algorithms. In terms of Process Mining, event data is used to discover process models or for analysing process compliance and performance by replaying a model [17, p. 13].

Despite the fact that Process Mining is a sub-discipline of Data Science, many Big Data and Data Science initiatives lack a process perspective. Therefore, "*Process Mining can be seen as a means to bridge the gap between data science and process science*". The term Process Science refers to a combination of information technology and manage-

ment sciences striving for running and improving operational processes. The illustration in the lower left of Figure 1.1 shows the "main ingredients" of Process Science, which consists, similar to Data Science, of overlapping sub-disciplines. It illustrates that Business Process Management (BPM), which includes methods for designing, executing, controlling, measuring and optimising business processes, is part of Process Science. Moreover, it shows that also Process Mining is one of the sub-disciplines, which, unlike BPM, does not aim on process modelling itself, but on exploiting event data. As Figure 1.1 reveals, Process Mining is the link between Process Science and Data Science, since it combines the strengths of data mining, statistics and machine learning methods with powerful modelling techniques. Therefore, by using Process Mining approaches, event data is related to process models so that actual processes can be discovered, existing models can be evaluated and end-to-end processes can be improved [17, pp. 15-18, p. 30].

Combining the advantages and possibilities of Process Mining in terms of process model discovery and evaluation with the benefits and simplicity of S-BPM concerning process choreography modelling, great value can possibly be created for companies and organisations.

1.1. Motivation and Goals

Intensive research showed, that Process Mining methods mostly focus on retrieving choreographies. For that reason, this thesis aims at answering the question *"Is it possible to generate process choreographies by means of Process Mining?"*, For that purpose, the S-BPM notation, which is used to model process orchestrations and choreographies, and Process Mining methods will be discussed and combined in this thesis.

There are a number of different algorithms available to discover process models. These algorithms have different weaknesses and strengths and each focusses on a different target language. Several algorithms, including the α algorithm, use Petri nets to visualise discovered process models [17]. Therefore, a solution has to be found to discover S-BPM process models based on Process Mining techniques. For this purpose the open-source Process Mining tool ProM (see Section 4.1) is used.

Since S-BPM focusses on the behaviour of each subject of a process as well as on the interaction between subjects [6], a suitable event log structure, which is the basis for all Process Mining efforts, has to be developed. Moreover, an algorithm has to be found that takes the communication between multiple actors within a process into account.

The S-BPM Modelling and Execution Platform, as presented in Chapter 5, provides a possibility to model and execute S-BPM processes. However, it does not support the full BPM life-cycle [17] (see Figure 3.1), as there are no monitoring and diagnosis mechanisms for processes implemented. However, by using Process Mining methods and by implementing new features into the platform, the platform may be able to support the full BPM life-cycle. This includes the logging of events as well as the generation and extraction of event-logs in suitable formats.

This thesis is to discuss Process Mining methods and to present their applications by using ProM. Another goal of this thesis is to find and test suitable Process Mining

mechanisms for S-BPM. Furthermore, an algorithm for process choreography discovery has to be found, which is applicable to both ProM and the S-BPM Platform. Moreover, it aims to illustrate and document the implemented solution that enables the S-BPM Modelling and Execution Platform to facilitate S-BPM Process Mining efforts and to support the full BPM life-cycle.

Furthermore, it will be discussed to what extent common event log structures and Process Mining techniques support process choreographies.

1.2. Approach

First, this thesis will start with a brief introduction to the principles of process modelling and different process modelling notations will be discussed. Second, the idea and the basics of Process Mining will be illustrated, including a proposal for an event log structure suitable for S-BPM process models and choreographies. Moreover, process discovery and the alpha algorithm will be explained. Afterwards the Process Mining tool ProM will be presented and methods, appropriate for the generation of process choreographies, will be developed. Then, the S-BPM Modelling and Execution Platform will be illustrated. Besides, the functionalities, which have been implemented as part of this master's thesis, supporting the choreography mining efforts will be documented and explained. Furthermore, it will be exemplified how a process can be modelled, imported and executed in the platform. Additionally, the resulting event log will be exported and used for applying Process Mining methods to regenerate the process choreography. Finally, the findings and the results of this thesis will be discussed.

2. Process Modelling

Process modelling and its formalisms and notations provide the basis for process science (Figure 1.1). Organisations use process models to structure discussions, to document procedures or to analyse and enact operational processes. Most of the process models are hand-made, without any linking to existing process data. Due to the fact that today's business processes rely heavily on communication and telecommunication, process modelling has great impact on organisations and management. Today there is a variety of different process modelling notations available [17, pp.55-57].

2.1. Transition Systems

A transition system, consisting of *states* and *transitions*, builds the most basic process modelling notation. As shown in Figure 2.1, states are represented by black circles, identified by a unique label with no additional meaning. Two states are connected by transitions represented by arcs, which are labelled with an activity name. Transition systems have a set of *initial states* and a set of *final states*. The set of states can be infinite. However, in most cases it will be finite, which then results in a so called *Finite-State Machine (FSM)* [17, p. 58].

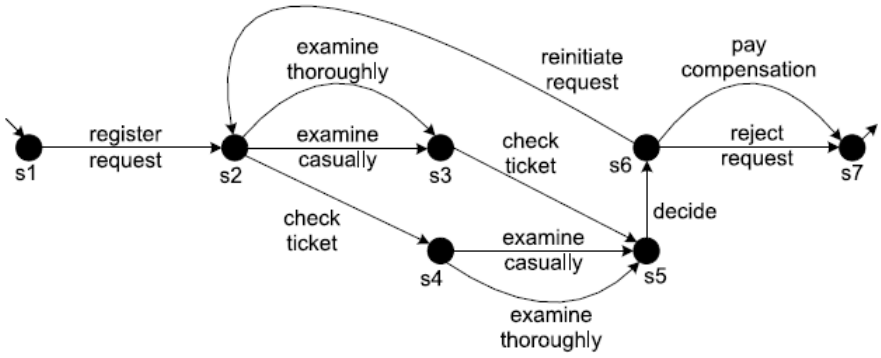

Figure 2.1.: Example of a transition system [17, p. 58]

Figure 2.1 shows a transition system, composed of a set of states from *s1* (initial state) to *s7* (final state). The first transition, for example, is formed by the state *s1* the activity *register request* and the state *s2*. Starting in one of the start states (in this case there is only one: *s1*) any path possible is referred to an *execution sequence* [17, pp. 58-59].

Since any process model can be expressed by a transition system, many notations can be translated to higher-level languages like Petri nets or BPMN. These notations provide the possibility to create more expressive models and overcome the problem of expressing concurrency in transition systems [17, p. 59].

2.2. Petri Nets

A Petri net, as shown in Figure 2.2, is a bipartite graph and a network structure that consists of a finite set of *places* and a finite set of *transitions*. *Tokens* that can flow through places of the graph are used to indicate the state of a Petri net, which is called the *marking* of the graph. Places and transitions are connected via directed arcs, called *flow relation*. Figure 2.2 shows a *marked* Petri net, since an initial marking (token) is located in the *start* place [17, pp. 59-60].

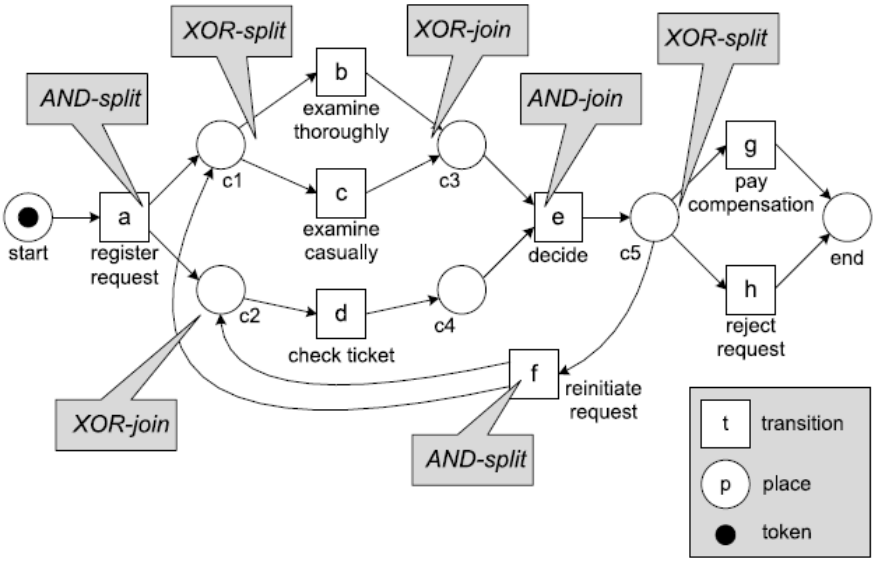

Figure 2.2.: Example of a Petri net [17, p. 60]

Petri nets have a dynamic behaviour by *enabled* transitions that can *fire*. When every input place of a transition contains a token, it is enabled and it can fire by consuming one token of each input place and producing one token for each output place [17, p. 61].

Figure 2.2 shows furthermore some logic structures: *AND* joins and splits as well as *XOR* joins and splits.

According to [17, p. 64] marked Petri nets may have some generic properties:

- *k-bounded*: A marked Petri net is bounded, if there is a finite number of tokens in each place. The number of tokens that a place can hold is referred to as *k*. If the maximum number of tokens that one place can hold is for example 25, than the Petri net is 25-bounded.
- *safe*: If the maximum number of tokens that one place can hold is 1, than the Petri net is 1-bounded and therefore *safe*.
- *deadlock free*: If every reachable marking enables at least one transition, a marked Petri net is called deadlock free.
- *live*: A marked Petri net is live if each transition can be enabled from every reachable marking.

2.3. Workflow Nets

Workflow nets (WF-nets) are Petri nets that have a dedicated start place (*source*) and a dedicated end place (*sinks*). WF-nets are used for business process modelling and Process Mining because process models describe the life-cycle of a case, including the creation (start) and the end (completion). Figure 2.2 shows also a WF-net, since the graph contains a start and an end place [17, p. 65].

A WF-net is *sound* if it is *safe* and *live* (cf. Section 2.2) [17, pp. 65-66].

2.4. Process Orchestration and Choreography

The modelled behaviour of a single actor in a business process is referred to as *orchestration*. However, typically a business process consists of multiple interacting partners and therefore understanding the behaviour of only one partner is often not sufficient. A process model that illustrates the behaviour and interaction of all partners in a process is called *choreography*. Figure 2.3 shows three individual process orchestrations (*P1, P2 and P3*), which form a choreography by linking the partners via dashed arrows [12, p. 618].

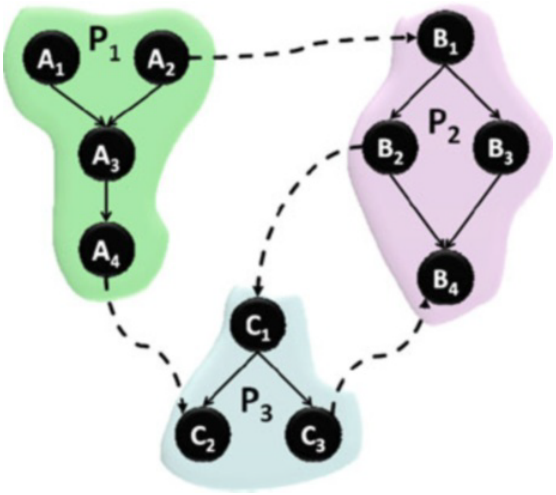

Figure 2.3.: Choreography and orchestrations [12, p. 619]

Choreographies derived from web service standardisation, as business partners have to coordinate activities. This collaboration via choreographies is based on the exchange of messages between partners. In BPMN, for example, collaboration diagrams, containing pools that depict the orchestration of each partner, are used to illustrate choreographies [2, p. 280].

2.4.1. Workflow Modules

As mentioned in 2.3, workflow nets are special Petri nets with a specified start and end place. A so-called *workflow module* can be used to model a process that is able to communicate with its environment. According to [13] an interface, which is "*a set*

of places representing direct message channels", is therefore added to a workflow net. A workflow module consists of *internal places*, *input places* and *output places*. Together with their transitions and directed arcs, the internal places form the *internal flow* and the input and output places form the *communication flow*. Moreover, a Petri net is a workflow module if the internal flow is a workflow net and if the transitions are not connected both to an input place and an output place. A workflow net within a module is called *internal process*, whereas the input and output places form the *interface*. To compose two workflow modules, the internal processes of both have to be disjoint and for each common place there has to be an output place provided in one module and an input place provided in the other module. This is referred to as *syntactically compatible*. However, it is not necessary that both interfaces match completely to be syntactically compatible. By composing two modules, the common places are merged and disjoint input and output places become the new interface of the composed module. Furthermore, two modules are semantically compatible only if the composed system is *weak sound*, which is a subset of *soundness* (cf. Section 2.3) allowing *dead* transitions [13].

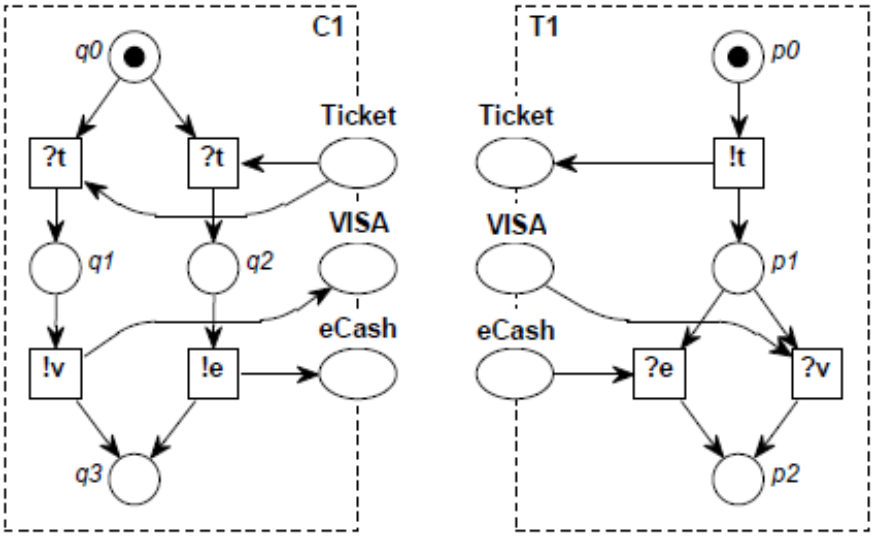

Figure 2.4.: Compatible Workflow Modules [13]

Figure 2.4 shows two workflow modules, representing a ticket service *T1* and a customer *C1*. The communication is started by *T1* by sending a *Ticket* to *C1*, which results in *T1* waiting for either *VISA* or *eCash* payment. After receiving the *Ticket*, *C1* chooses to either pay with *VISA* or *eCash* and sends the information back to *T1*. Afterwards, the corresponding payment transition fires in the module of *T1* and both processes end. The composed system of the two modules is weak sound and therefore both modules are semantically compatible. Figure 2.5 illustrates two modified modules, which are, by contrast, incompatible. After *T2* sending the ticket, the internal state would either change to *p1* or *p2* waiting for *VISA* or *eCash*. However, *C2* has the choice to either pay with *VISA* or *eCash*, although he does not know the internal state of *T2*. Therefore the composed system may end in a deadlock and it is not weak sound. A system composed of *C2* and *T1* is also weak sound and compatible, whereas a system composed of *C1* and *T2* is not compatible due to a possible deadlock [13].

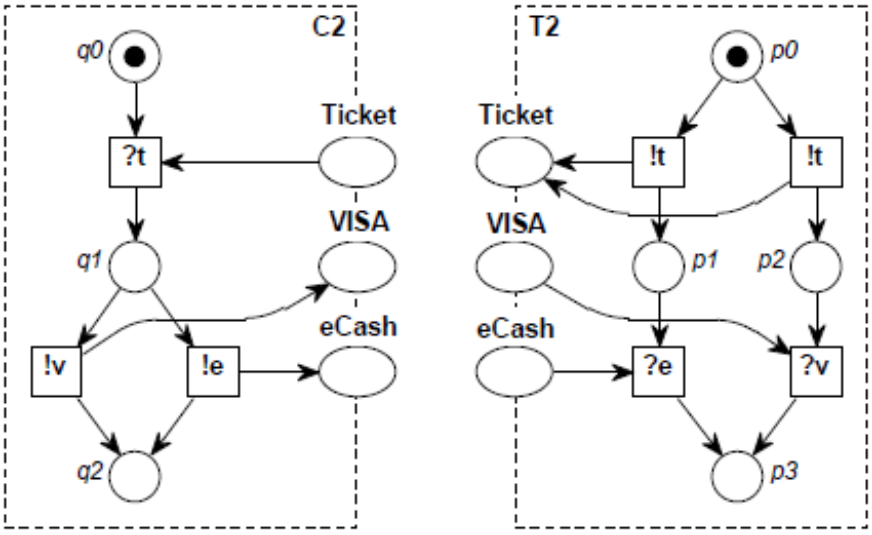

Figure 2.5.: Incompatible Workflow Modules [13]

2.4.2. Open Workflow Nets

A similar approach to workflow modules is called *Open Workflow Nets (oWFNs)* as described in [14]. oWFNs are "liberal" workflow nets that allow communication by using *communication places*. An open workflow net is defined as a Petri net that has *input* and *output places*, an *initial marking*, an a set of *final markings*. A *composition* illustrates the interaction of two oWFNs that share input- and output elements [14].

Since the demonstrations and implementations of this thesis are based on the α algorithm (see Section 3.3), all Process Mining efforts presented in the subsequent chapters generate a Petri net. Since S-BPM process models visualise the message exchange (see Section 2.5), the communication perspective has to be taken into account when Process Mining methods are applied in order to generate S-BPM process models. For this purpose, Open Workflow Nets or Workflow Modules are used in this thesis, as they provide a possibility to visualise the communication between actors of a process. As a result, S-BPM Process Models can be generated by transforming discovered Petri nets into oWFNs, as described in Chapter 4.

2.5. Subject-Oriented Business Process Modelling

According to [6, p. 26] subject-oriented business process modelling (S-BPM) understands a business process as "*a set of interrelated activities (tasks), which are handled by active entities (people or systems performing work tasks) in a logical (with respect to business) and chronological sequence, and which use resources (material and information) to work on a business object for the purpose of satisfying a customer need (to thus contribute an added value), and which have a defined start and input, as well as a defined end state and result.*" Business objects are defined as objects that are relevant for the exchange of messages between subjects as well as for the individual activities of subjects [6, p. 26].

S-BPM provides a framework that focuses on the subject of a process. Processes

are modelled in natural language and can be directly executed. Therefore all people involved in the processes of an organisation can learn how to model subject-oriented process models[6, pp. 2-6].

Complete sentences of natural language consist of a subject, a predicate and an object. These three elements are sufficient for the description of a business process, since processes consists of actors (subject) who perform actions (predicate) on certain objects (object). Moreover, these sentence building blocks answer the questions ”Who acts?” (subject), ”What does the subject?” (predicate) and ”What edits the subject?” (object). Figure 2.6 shows a business trip process described in natural language[6, pp. 16-17, p. 56].

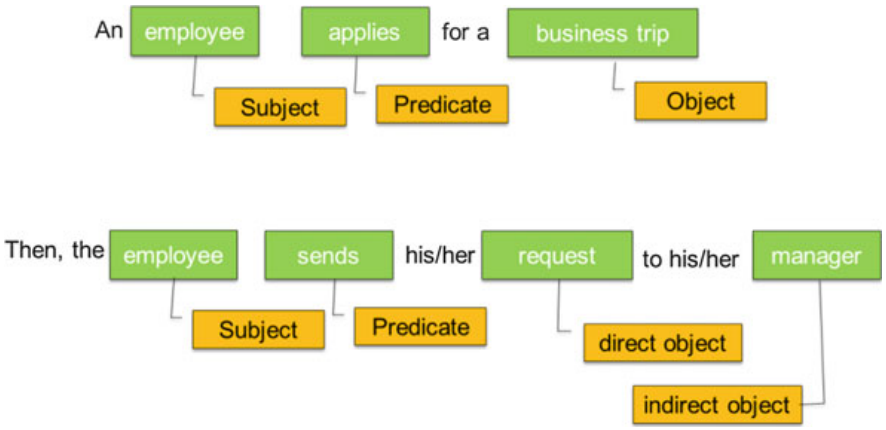

Figure 2.6.: Process description in natural language [6, p. 17]

In a first step, a subject-oriented description of a business process is composed of the subjects and the messages, along with required data that are exchanged between the roles involved [6, p. 19]. The interaction between subjects in a business trip process model is illustrated in Figure 2.7, which is called Subject Interaction Diagram (SID) [6, p. 73].

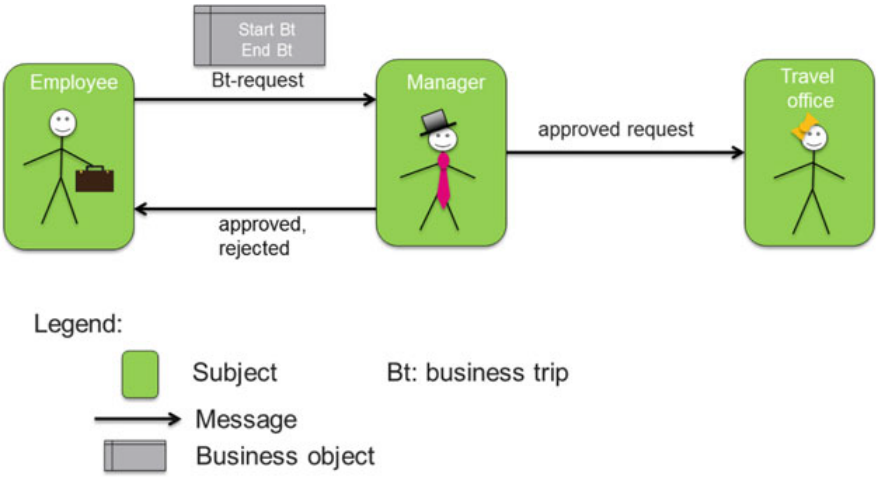

Figure 2.7.: Interaction between subjects involved in the application process [6, p. 19]

After describing the interaction between the subjects, the behaviour, including the activities and the interaction, of the individual subjects is described [6, p. 19]. The

description of the behaviour of the employee can be seen in Figure 2.8, which is called Subject Behaviour Diagram (SBD) [6, p. 80]. The SBD of a subject is a Finite State Machine (FSM) graph, describing the sequence of executing actions, sending and receiving messages [4, p. 1].

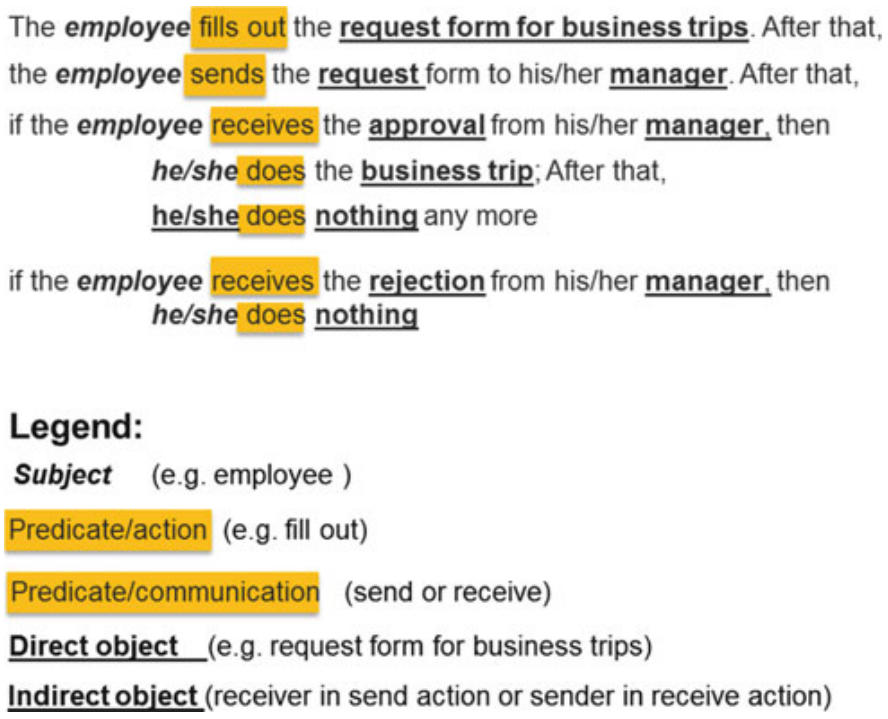

Figure 2.8.: Description of the behaviour of the employee in the business trip process [6, p. 19]

The description in natural language reveals that the process consists of multiple paths, which appears clearer in a graphical representation, as it is shown in Figure 2.9. This figure shows the order of actions, states and messages of the employee. A triangle in the upper left corner marks a start state, which, in this case, is a function state in which the application is filled out. After that a transition happens to a send state, marked by a triangle at the bottom, in which the request is sent to the manager. Then the employee enters a receive state, marked by a triangle at the top, in which the employee waits until a response from the manager is received. If the manager rejects the business trip, the process ends, which is marked by a triangle in the lower right corner. If the manager approves the business trip, the employee enters the next state, goes on the trip and then the application process ends. As it is illustrated in Figure 2.10, the behaviour of the employee in the business trip application process is complementary [6, p. 20].

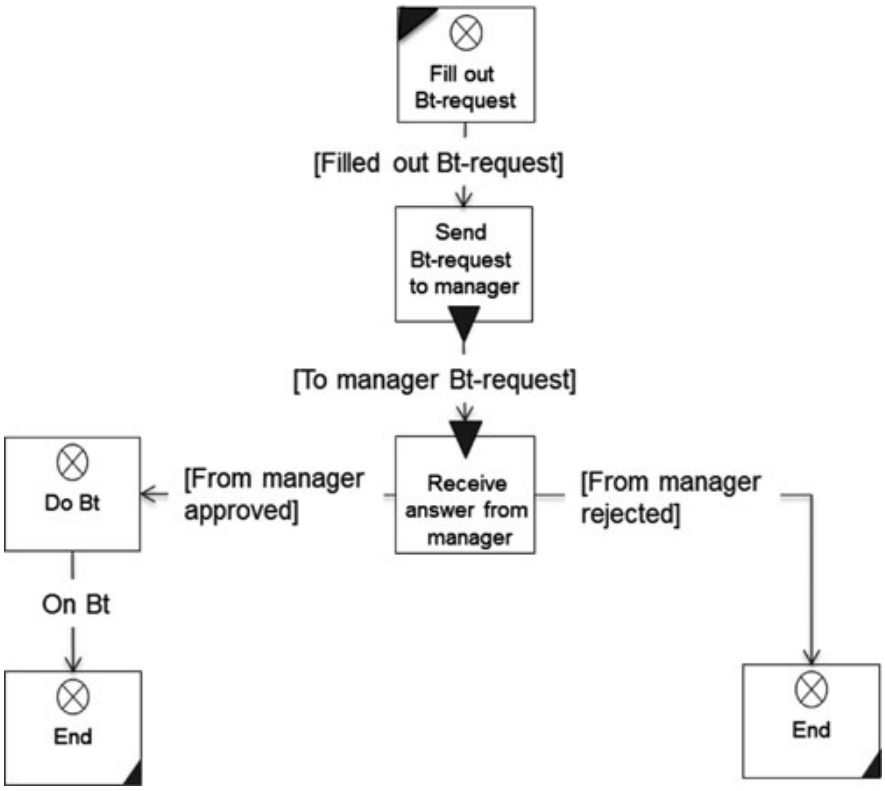

Figure 2.9.: Illustration of the behaviour of the employee in the business trip process [6, p. 20]

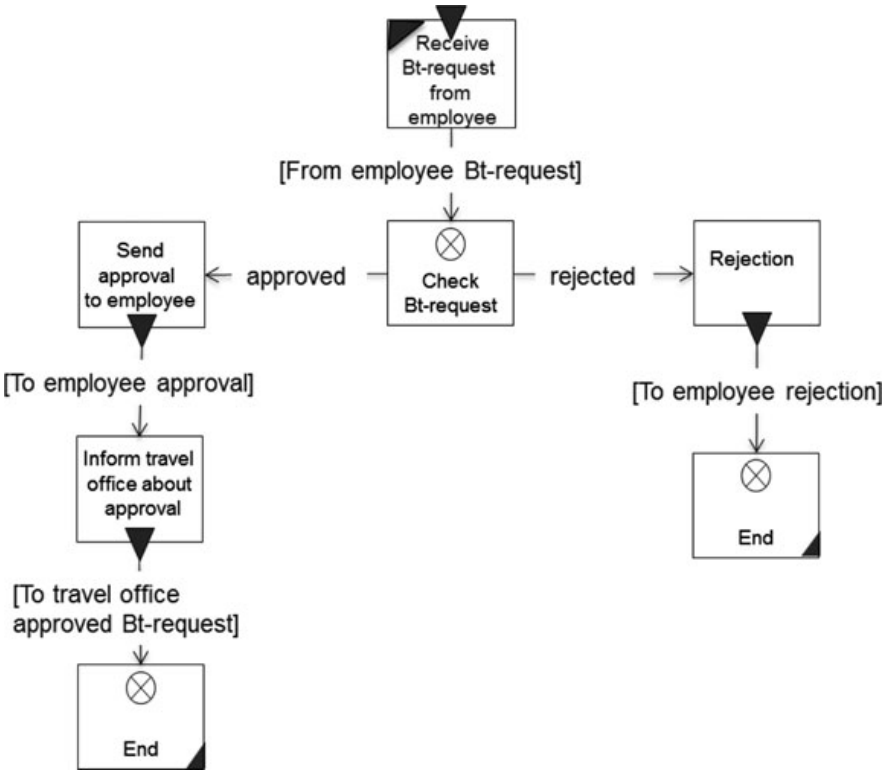

Figure 2.10.: Illustration of the behaviour of the manager in the business trip process [6, p. 20]

3. Process Mining

A business process has to be managed throughout different phases, which is described by the so-called BPM life-cycle that is shown in Figure 3.1. After designing a business process, the resulting model has to be configured for and implemented into a running system. In these two phases the process model plays an important role. During the phase of enactment/monitoring, a running process is monitored and process data is generated. If necessary changes emerge, they may be handled in the adjustment stage by using predefined controls without redesigning the process. In the diagnosis/requirements phase however, the process is evaluated based on the data generated and emerging changes in the environment are monitored. A new iteration of of the BPM life-cycle is started if poor performance or new demands are detected. Until recently most organisations did not support the phase of diagnosis/requirements systematically and continuously. Moreover, a new iteration of the life-cycle was triggered only by severe problems or major changes and the data generated was not actively used in the redesign phase [17, pp. 30-31].

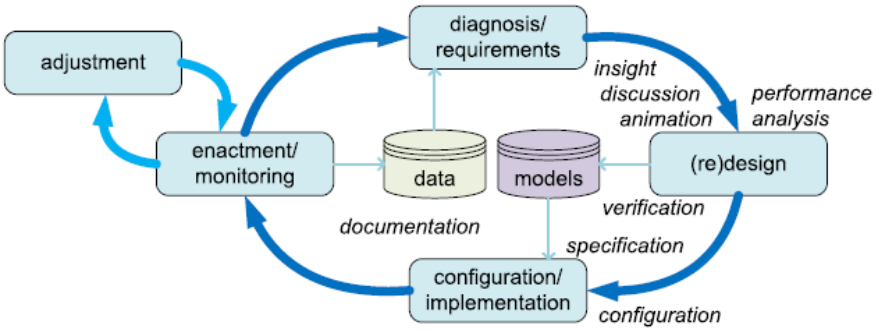

Figure 3.1.: BPM life-cycle [17, p. 31]

Process mining methods use event logs to create connections between actual processes, process models and data. Therefore, information about activities, including their sequence, the corresponding case, the executing resource as well as additional data elements, has to be provided by an event log to initiate process mining *discovery*, *conformance* or *enhancement* techniques. First, by using discovery techniques a process model is generated out of an event log without any additional information provided. Second, conformance methods are used for comparing a process with a corresponding event log, to check the alignment of reality and model. Third, process mining enhancement uses event logs to improve existing process models, by changing or extending them [17, pp. 32-34].

3.1. Using the Data

Today's information system store incredible amounts of event data, however, often in unstructured form, over many tables or over multiple subsystems. Hence, data extraction techniques are necessary for applying Process Mining methods. The minimum information required to be stored in an event log for any Process Mining effort is an identifier for the case as well as for the activity. According to [17, p. 38] the goal of process mining "*is not to represent just the particular set of example traces in the event log. Process mining algorithms need to generalise the behaviour [...]*". Models generated by process mining techniques should neither be too specific ("overfitting"), allowing only the behaviour observed in the event log, nor too general ("underfitting"), allowing any unrelated behaviour [17, pp. 32-38].

Inspired by David Harel, van der Aalst [17, p. 41] uses the terms *Play-In*, *Play-Out* and *Replay* to illustrate the relation between a process model and an event log. Play-Out refers to the execution of a process model while generating behaviour (event log or other source of information). This is for example a workflow-engine or a simulation tool. Play-In, by contrast, takes example behaviour (from an event log or other source of information) to construct a process model. Since traditional data mining techniques are not sufficient for discovering process models, Process Mining methods have been developed. Replay refers to a combination of using an event log and a process model. Replay can be used for conformance checking, extending the model with frequencies and temporal information, for constructing predictive models and for operational support [17, pp. 41-44].

3.2. Event Logs

To answer certain questions about operational processes by using process mining techniques, event logs, as exemplified in Figure 3.2, are needed. However, often the data about processes is distributed over different data sources like databases, flat-files, scanned text, PDF documents and many more. In most cases data extraction efforts are needed to collect data from different sources into one event log [17, pp. 125-128].

Case id	Event id	Properties				
		Timestamp	Activity	Resource	Cost	...
1	35654423	30-12-2010:11.02	register request	Pete	50	...
	35654424	31-12-2010:10.06	examine thoroughly	Sue	400	...
	35654425	05-01-2011:15.12	check ticket	Mike	100	...
	35654426	06-01-2011:11.18	decide	Sara	200	...
	35654427	07-01-2011:14.24	reject request	Pete	200	...
2	35654483	30-12-2010:11.32	register request	Mike	50	...
	35654485	30-12-2010:12.12	check ticket	Mike	100	...
	35654487	30-12-2010:14.16	examine casually	Pete	400	...
	35654488	05-01-2011:11.22	decide	Sara	200	...
	35654489	08-01-2011:12.05	pay compensation	Ellen	200	...
...	...	...	...	...	...	...

Figure 3.2.: Fragment of an event log [17, p. 129] (edited)

In this thesis, operational data, created by an S-BPM execution platform (see Chapter 5), will be extracted from a database. This chapter will describe a suitable structure for event logs and the corresponding extraction mechanism will be implemented into the S-BPM platform, which will be described and illustrated in a subsequent chapter.

3.2.1. General Structure

According to van der Aalst [17, pp. 128-129] an event log contains data that can be related to a single process. Furthermore, he states that a process consists of cases that are composed of events. Within a case, events are ordered and they may have attributes. Standard attributes are activity, timestamp, resource and transaction type. Since activities may take time, events may describe different transactions within an activity, for example assignment, start and completion. Possible transaction types are described by the standard transactional life-cycle model illustrated in Figure 3.3. Events can be identified by their activity name. Since same activities can occur multiple times in a process, it may be useful to track activity instances in an extra attribute to avoid correlation problems. Different process mining techniques may need different event log structures. Therefore, filtering mechanisms can be applied to the data source [17, pp. 128-133].

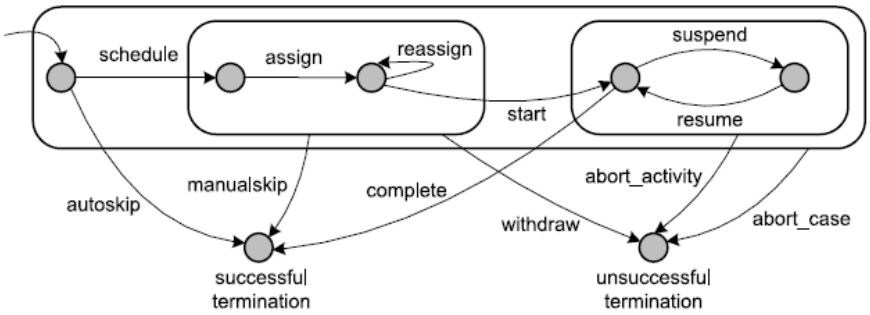

Figure 3.3.: Standard transactional life-cycle model [17, p. 131]

Taken all together, an *event log* is a set of *cases*, consisting of *events*. The finite sequence of events form a *trace*. Events and cases may have an indefinite number of *attributes*. Events should have an *activity* attribute, referring to the name of the activity. Unique identifiers are used to represent events and cases. Different aspects may be analysed based on the attributes given in the log. For example, Petri nets can be discovered by using only activity attributes. To measure the duration of activities, the transactional attribute and a timestamp is needed. Moreover, the flow of activities between persons or roles can be discovered by using the resource attribute [17, pp. 134-136].

3.2.2. Structure for S-BPM

As described in Chapter 2.5, S-BPM process models are composed of subjects, states, transitions and messages. The Subject Interaction Diagram (SID), illustrated in Figure 3.4, shows a vacation request process model. The SID reveals, that two subjects are involved in this process: an employee and a boss. These subjects exchange messages, containing information about the vacation request itself and information about either a positive or a negative response.

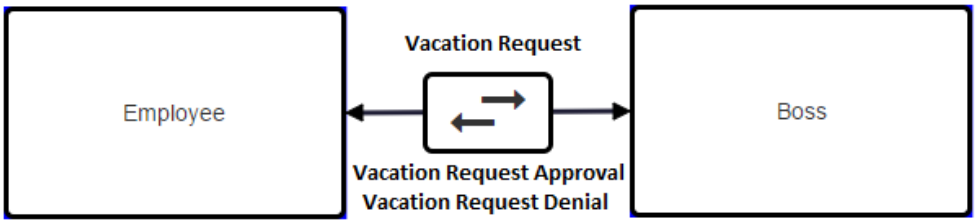

Figure 3.4.: Subject Interaction Diagram of a vacation request process

Figure 3.5 shows the Subject Behaviour Diagram (SBD) of an employee. As it can be seen, the process is started by creating a vacation request, followed by sending it to the boss. Then, a response is received, containing either an approval or a denial. Depending on the response, the employee will go on vacation or not. Afterwards the process ends.

The SBD of a boss is visualised in Figure 3.6. It shows that the process is triggered by receiving a vacation request, followed by answering the request. The process ends after sending either an approval or a denial to the employee.

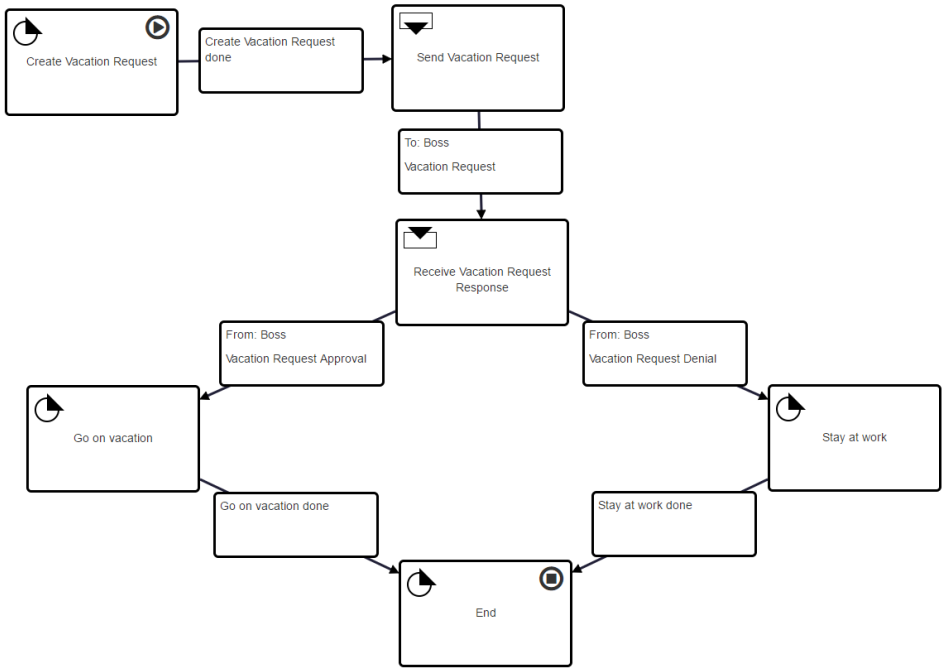

Figure 3.5.: Subject Behaviour Diagram of an employee in a vacation request process

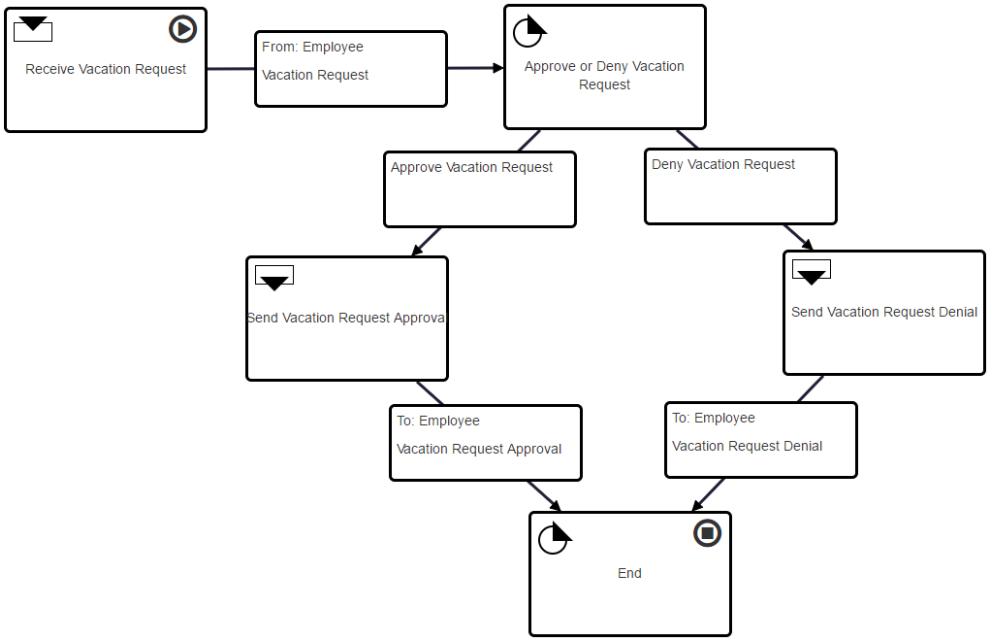

Figure 3.6.: Subject Behaviour Diagram of a boss in a vacation request process

This process may be executed in the S-BPM execution platform (see Chapter 5). Hence, following the principle of *Play-Out*, an event log could be created. The simplest event log that may be generated by the vacation request process is shown in Figure 3.7. Following the examples in Section 3.2.1, this low level event log tracks the case id, timestamps, activity names and resources. Every activity refers to a *state*, the resources refer to the role of the executing *subject*. This log illustrates a valid flow in both SBDs, however, there is too little information provided to reproduce the process model in S-BPM style. For this reason additional attributes are required.

Case Id	Timestamp	Activity	Resource
1	18.05.2017 08:00	Create Vacation Request	Employee
1	18.05.2017 09:00	Send Vacation Request	Employee
1	18.05.2017 09:00	Receive Vacation Request	Boss
1	18.05.2017 10:00	Approve or Deny Vacation Request	Boss
1	18.05.2017 10:05	Send Vacation Request Approval	Boss
1	18.05.2017 10:05	End	Boss
1	18.05.2017 10:05	Receive Vacation Request Response	Employee
1	18.05.2017 11:00	Go on vacation	Employee
1	18.05.2017 11:00	End	Employee

Figure 3.7.: Simple event log of the vacation request process

Besides subjects and activities, an S-BPM process model illustrates the interaction between subjects via messages. As shown in the SID in Figure 3.4, the exchange of information between employee and boss is based on message types called *Vacation Request* (from employee to boss) and either an *Approval* or *Denial* (from boss to employee). The corresponding SBDs (Figures 3.6 and 3.5) show which messages are sent or received throughout the process flow. Therefore information about the messages as well as about the states has to be provided in the event log via attributes. Moreover, it is necessary to determine who is the sender of the corresponding message type in a receive state and who is the receiver of the corresponding message type in a send state, which is especially necessary for process models with more than two subjects. This requirement is illustrated in Figure 2.8 as *Indirect object*. As it can be seen in Figure 3.8, four columns, *State*, *Message Type*, *Recipient* and *Sender*, have been added to the log. The state column reveals if an activity is either associated with a function, receive or send state. Since receive and send states are always linked to a message, the column message type indicates which message is processed in this action. Moreover, it is necessary to determine the recipient or the sender of the message since a message type could be received from or sent to different subjects within a process. Receiving a message of subject A could lead to another process execution path than receiving the same message of subject B. Therefore recipient and sender are stated in the corresponding columns.

Case Id	Timestamp	Activity	Resource	State	Message Type	Recipient	Sender
1	18.05.2017 08:00	Create Vacation Request	Employee	Function			
1	18.05.2017 09:00	Send Vacation Request	Employee	Send	Vacation Request	Boss	Employee
1	18.05.2017 09:00	Receive Vacation Request	Boss	Receive	Vacation Request	Boss	Employee
1	18.05.2017 10:00	Approve or Deny Vacation Request	Boss	Function			
1	18.05.2017 10:05	Send Vacation Request Approval	Boss	Send	Vacation Request Approval	Employee	Boss
1	18.05.2017 10:05	End	Boss	Function			
1	18.05.2017 10:05	Receive Vacation Request Response	Employee	Receive	Vacation Request Approval	Boss	Employee
1	18.05.2017 11:00	Go on vacation	Employee	Function			
1	18.05.2017 11:00	End	Employee	Function			

Figure 3.8.: Advanced event log of the vacation request process

3.3. Discovery and the Alpha Algorithm

Process discovery algorithms, such as the α-algorithm, construct process models based on behaviour captured in event logs. An event log *L* is a set of cases, composed of a sequence of events. *L* is therefore "*a multi-set of traces over some set of activities $\mathscr{A}$, i.e. L ∈ B($\mathscr{A}$*)*" [17, p. 163].

According to [17, p. 163] "*a process discovery algorithm is a function that maps L onto a process model such that the model is "representative" for the behavior seen in the event log*". This vague and general definition does neither specify the target format of the process model nor the input format. A more specific definition includes information about the log and the outcome, for example "*a process discovery algorithm is a function γ that maps a log L ∈ B($\mathscr{A}$*) onto a marked Petri net γ (L) = (N, M)*" [17, p. 164].

There are certain challenges and limitations of process discovery. First, an algorithm cannot be expected to reconstruct the original layout of a process model. The layout does not affect the behaviour of a process and information about it is therefore not included in the event log. Second, discovery techniques focus on certain target languages (for example work-flow nets, BPMN, S-BPM) and any method is therefore restricted by their expressive power, which is called *representational bias*. Third, event logs may contain behaviour which is not representative (*noise*) or logs may contain too few events (*incompleteness*). Moreover, logs may not fulfil other quality dimensions like *fitness, simplicity, precision or generalisation* [17, pp. 178-192].

Alpha Algorithm

The α algorithm is a simple process discovery algorithm that can be seen as a basis for the ideas and challenges of process discovery. However, more advanced algorithms should be used in practical mining scenarios. Given a simple event log the αalgorithm generates a Petri net that replays the log by scanning for particular patterns. For example an activity *a* that is always followed by *b* is illustrated by a sequence pattern *a* →*b*. Other process patterns like *AND-* and *OR-joins* and *-splits* can be found as well [17, pp. 167-169].

The α algorithm is defined in eight steps. First it is checked which activities are in the log that form the transitions of the net. Second, every trace is scanned for the first activity, which together form the set of start activities. Then every trace is scanned for the last activity to form the set of end activities. In step four and five the places are discovered, by identifying the input and output transitions. In step six an initial place and a final place is added to the net. Afterwards, in step seven, the arcs will be added from the initial place to all start activities, from the final place to all end activities and between the other places and the corresponding input and output transitions. Finally, the algorithm returns a Petri net composed of places, transitions and arcs [17, pp. 171-173].

The demonstrations and implementations in this thesis are based on the α algorithm, as it generates a Petri net, which will be used for further manipulation to reconstruct an S-BPM process model as described in Chapter 4. Since there are not any complex processes presented in this thesis, the α-algorithm is sufficient for demonstration purposes.

4. Process Mining Applied

In this chapter process mining approaches are applied to sample logs in order to generate process models automatically. It is shown which methods and procedures can be used to discover process orchestrations as well as process choreographies.

4.1. ProM

ProM[1] is an open-source process mining platform, which was first released in 2004. Over the years a lot of plugins were added to ProM, including support for multiple mining algorithms, process discovery and conformance methods. Since 2004 multiple versions have been released and ProM has become the de-facto standard tool for process mining [17, pp. 332-333].

At the time of preparation of this thesis ProM was in version 6.6. (June 2017). ProM 6 was designated to loosen the tight coupling between the user interface and the analysis techniques of ProM 5. Therefore ProM 6 was implemented completely from scratch [17, p. 333]. As a consequence, multiple plugins of ProM 5 are no longer available in ProM 6, since they would have to be reimplemented[2]. Due to this fact features of both ProM 6.6. and 5.2. will be used.

4.2. First Steps

Using ProM process discovery algorithms, for example the α algorithm, can easily be applied using event logs.

In a first step the α algorithm is applied to the simple event log visualised in Figure 4.1, which shows two cases with different traces. In order to use this algorithm in ProM 6.6., an event log in XES (Extensible Event Stream)[3] format has to be provided.

After starting ProM 6.6. the log is imported in CSV (comma-separated values) format by clicking in the top right corner on *import...* and then the CSV import plugin has to be selected. The imported CSV file is then shown in the list of resources on the *Workspace* page, as shown in Figure 4.2. In the workspace resources can be imported, exported, renamed, deleted as well as viewed and selected for further processing.

[1] `http://www.promtools.org`

[2] e.g. the *Petri Net to oWFN Plugin* is not available in ProM 6 (`http://www.win.tue.nl/promforum/discussion/comment/2327/#Comment_2327` - retrieved 11/07/2017)

[3] `http://www.xes-standard.org/`

Case Id	Timestamp	Activity	Resource
1	18.05.2017 08:00	Create Vacation Request	Employee
1	18.05.2017 09:00	Send Vacation Request	Employee
1	18.05.2017 09:00	Receive Vacation Request	Boss
1	18.05.2017 10:00	Approve or Deny Vacation Request	Boss
1	18.05.2017 10:05	Send Vacation Request Approval	Boss
1	18.05.2017 10:05	End	Boss
1	18.05.2017 10:05	Receive Vacation Request Response	Employee
1	18.05.2017 11:00	Go on vacation	Employee
1	18.05.2017 11:00	End	Employee
2	19.05.2017 08:00	Create Vacation Request	Employee
2	19.05.2017 09:00	Send Vacation Request	Employee
2	19.05.2017 09:00	Receive Vacation Request	Boss
2	19.05.2017 10:00	Approve or Deny Vacation Request	Boss
2	19.05.2017 10:05	Send Vacation Request Denial	Boss
2	19.05.2017 10:05	End	Boss
2	19.05.2017 10:05	Receive Vacation Request Response	Employee
2	19.05.2017 11:00	Stay at work	Employee
2	19.05.2017 11:00	End	Employee

Figure 4.1.: Simple event log of the vacation request process with two different cases

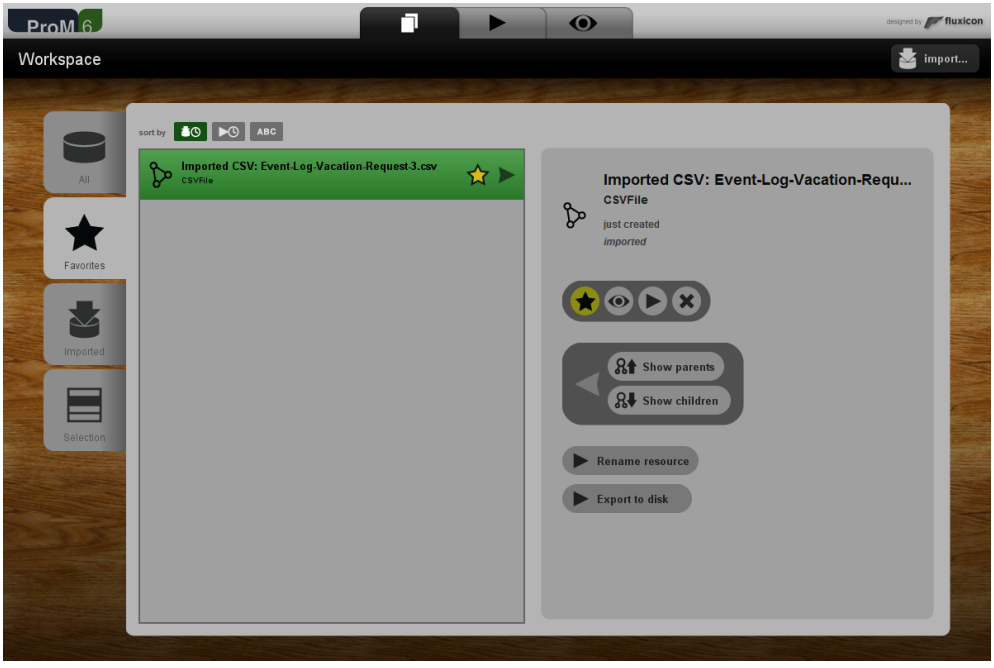

Figure 4.2.: ProM 6.6. Workspace after the CSV file import

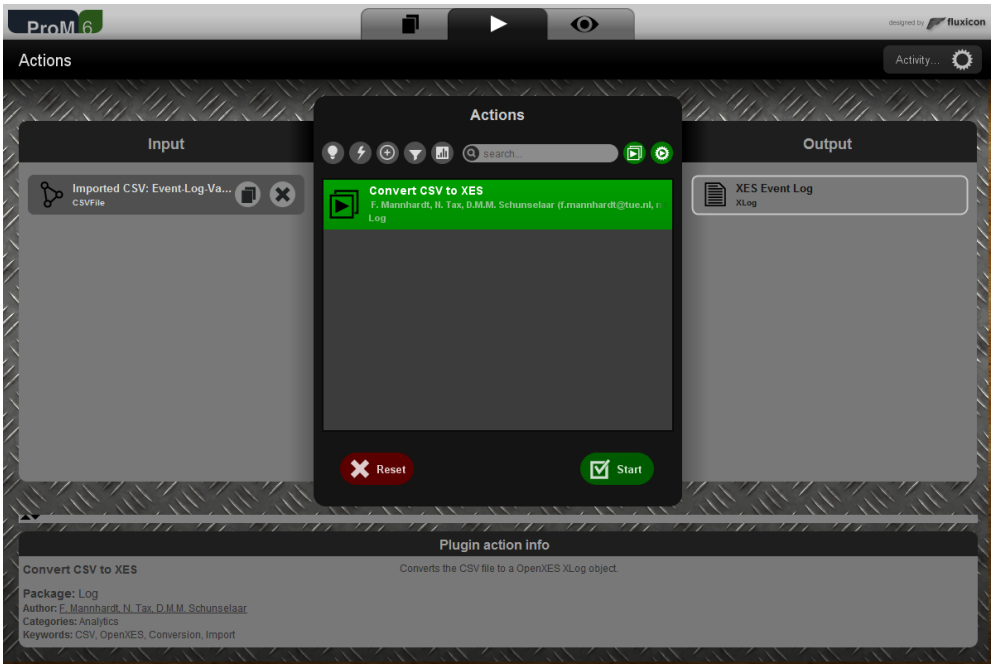

Figure 4.3.: ProM 6.6. Actions view

ProM offers a plugin to generate an XES file from a CSV file by specifying information about the case and the event identifier columns. For this purpose the ”play” button in the resource detail area on the right side of the workspace has to be selected. Afterwards the view changes from the *Workspace* to the *Actions* page which is shown in Figure 4.3. On the actions page plugins can be selected to convert, manipulate and filter data as well as to accomplish process mining tasks. The page shows a column for *Input* data, a column *Actions* and a column *Output* for the output data. Moreover, a short plugin description is shown at the bottom. In the actions column a plugin has to be selected to perform a task on the input data, which results in a specific output shown in the corresponding column. To run the CSV to XES conversion the *Start* button has to be selected.

In the conversion plugin CSV parser settings can be configured. Afterwards the mapping of the case column and the event column to the corresponding XES attributes have to be selected. Moreover a column for start or completion time can be specified. For the first example the column *Case Id* is matched with the case identifier and a combination of the columns *Activity* and *Resource* is used for the event identifier. No additional conversion settings are configured. After the conversion an *XES Event Log* view is shown, illustrating key data and a summary of the information provided in the log. Then the ”play” button in the top right corner is clicked to select this event log as the input for the next action.

In the actions view the *Alpha Miner* plugin is selected, which takes an XES event log as input and generates two output resources: *Petrinet* and *Marking*. After starting the plugin, the event classifier and version of the α algorithm can be specified. In this case *Event Name* and *Alpha* is selected. By applying the α algorithm to the simple event log of the vacation request process, a Petri net that is also a WF-net, is generated. Figure 4.4 shows the generated control flow, which combines the process model of the employee and the process model of the boss into one model.

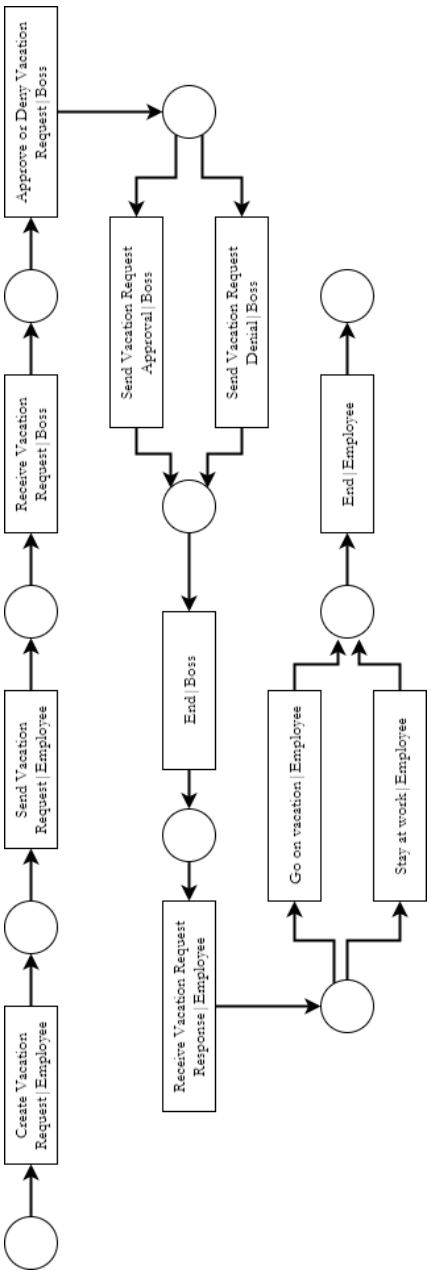

Figure 4.4.: Petri net based on the simple event log with two cases generated by the α algorithm

This process model, however, does neither reveal any information about the subjects and the communication between them, nor does it show distinct behaviours. Only the transition name indicates which subject is involved in which action. In a second attempt some minor adjustments are made when converting the CSV log to the XES log: The case identifier is now composed of the columns *Case Id* and *Resource* and the columns *Activity* and *Resource* form the event identifier. Then, the α algorithm is applied again and the process model shown in Figure 4.5 is generated. At the start of this process model, two ”main” control flows are shown: either a sequence of employee related transitions, or a sequence of boss related transitions could be fired. Both paths show correct orchestrations: the boss can either send an approval or a denial and the employee can either go on vacation or stay at work. However, this process model indicates that either the ”employee path” or the ”boss path” could be taken in a process. Moreover, the choreography between the boss and the employee is missing, since this model does not reveal anything about the communication and interaction between the subjects.

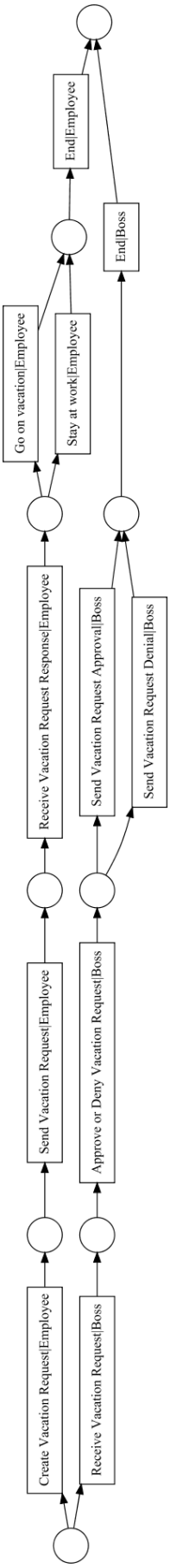

Figure 4.5.: Petri net based on a revised event log with two cases generated by the α algorithm

Conclusion

In a first step the α algorithm has been applied to a simple event log of a vacation request process. The event log was composed of two cases and the data was stored in a CSV file. In ProM 6.6. following steps were performed:

1. Importing the CSV file into ProM.
2. Converting the CSV file to an XES log, specifying the case identifier by using the *Case Id* and the *Resource* and specifying the event identifier by using the *Activity* and *Resource*.
3. Starting the *Alpha Miner* plugin and applying the α-algorithm on the XES log.

The output was the resulting Petri net shown in Figure 4.5, which is also a WF net. It is shown that the α algorithm was able to discover two valid flows of the vacation request. However, the resulting model indicates that this process could either be performed by an employee or by a boss but never by both actors at the same time. Since the α algorithm always adds a single start place and a single end place to the model, the resulting WF net does not show two separate orchestrations. Hence, it was not possible to illustrate a choreography between the two subjects.

4.3. Retrieving Separate Orchestrations

As illustrated and discussed in Section 4.2, the α algorithm is not able to model the orchestrations of interacting subjects separately. However, it was shown that the behaviour of each actor could be visualised in a process model. Nevertheless, a method has to be found that can retrieve a separate process model for each actor from an event log.

The event log in Figure 4.1 offers enough information to retrieve a process model for the employee and a process model for the boss. Therefore the log is filtered by each resource and split into two separate CSV files. Then, each file is loaded into ProM 6.6. Both files have to be converted to XES event logs. For this purpose the column *Case Id* is used for the case identifier property and the column *Activity* is used for the event identifier property. Afterwards, the *Alpha Miner* plugin is applied to both logs, resulting in two separate process models, which are shown in Figure 4.6 and Figure 4.7. Both process models are similar to the subject behaviour diagrams shown in Figure 3.5 and Figure 3.5.

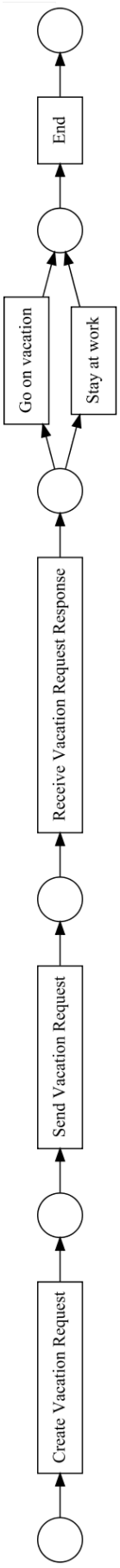

Figure 4.6.: Vacation request process model for the employee mined by the α algorithm

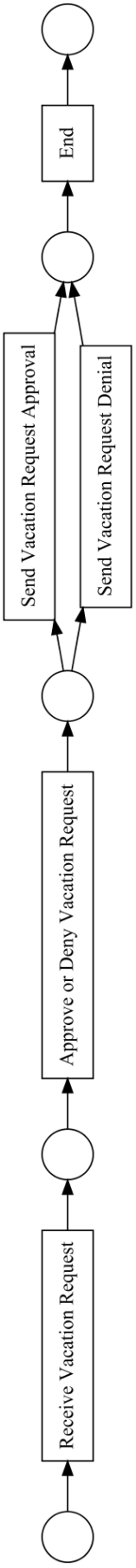

Figure 4.7.: Vacation request process model for the boss mined by the α algorithm

Conclusion

By filtering an event log by the subjects, two separate CSV files were created. Then the same steps as in Section 4.2 were performed for both files in ProM 6.6, except for specifying different columns when converting the CSV file to an XES log. It was shown that a WF net for each subject in the vacation request process was created, which is similar to the corresponding subject behaviour diagram. Nevertheless, information about communication and message exchange is still missing.

4.4. Retrieving Choreographies

As the previous Sections 4.2 and 4.3 show, the process models produced by the α algorithm are always WF nets that are composed of exactly one start place and one end place. However, as discussed in the Sections 2.4.1 and 2.4.2, special WF nets like *Workflow Modules* or *Open Workflow Nets* can be used to model processes using Petri nets. In addition to the internal places, these approaches use communication places as input and output for exchanging messages.

The activities of an event log are mapped to transitions in a Petri net mined by the α algorithm. The places, by contrast, are automatically discovered and are composed of input and output transitions. Only the start and end places, which are automatically added by the algorithm, do not consist of both an input and an output transition. For that reason ProM is not capable of creating an oWFN or a Workflow Module by simply changing the event log manually. As a result, additional manipulation has to be performed to retrieve the communication places of an oWFN or a Workflow Module.

4.4.1. From Petri net to oWFN or Workflow Module

A Petri net created by ProM 6.6. can be saved as a Petri Net Markup Language (PNML)[4] file by selecting *Export to disk* in the *Workspace*. As shown in Listing 4.1, the resulting PNML file contains an XML (Extensible Markup Language) structure. This structure contains a *pnml* tag, which is composed of one *net* tag describing the Petri net itself. The Petri net is of type EPNML (Extended PNML) which is a particular type of PNML [18].

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <pnml>
3    <net id="net1" type="http://www.yasper.org/specs/
        epnml-1.1">
4       <name>
5          <text>Petrinet</text>
6       </name>
7       <place id="n1">
8          <name>
```

[4] http://www.pnml.org/

```
9              <text>([Receive Vacation Request],[
                  Approve or Deny Vacation Request])</
                  text>
10          </name>
11          <toolspecific tool="ProM" version="6.4"
               localNodeID="f577de66-eee4-46c8-a065-58195
               ea9820a"/>
12          <graphics>
13             <position x="11.25" y="11.25"/>
14             <dimension x="12.5" y="12.5"/>
15          </graphics>
16       </place>
17       <!-- More places -->
18       <transition id="n8">
19          <name>
20             <text>Receive Vacation Request</text>
21          </name>
22          <toolspecific tool="ProM" version="6.4"
               activity="Receive Vacation Request"
               localNodeID="d86e3cf9-0e27-43dc-af19-
               de65ed0d8afe"/>
23          <graphics>
24             <position x="17.5" y="15.0"/>
25             <dimension x="25.0" y="20.0"/>
26             <fill color="#FFFFFF"/>
27          </graphics>
28       </transition>
29       <!-- More transitions -->
30       <arc id="arc20" source="n8" target="n1">
31          <name>
32             <text>1</text>
33          </name>
34          <toolspecific tool="ProM" version="6.4"
               localNodeID="fb883a26-e458-45a4-a422-682
               f9b7912e7"/>
35          <arctype>
36             <text>normal</text>
37          </arctype>
38       </arc>
39       <!-- More arcs -->
40       <finalmarkings/>
41    </net>
42 </pnml>
```

Listing 4.1: Skeleton of Petri Net in EPNML

All places of the Petri net are described in *place* tags, which have a name that is composed of a set with the names of the input transitions (for example *Receive Vacation*

Request) and a set with the names of the output transitions (for example *Approve or Deny Vacation Request*). Moreover, there is information about the positioning and the dimension of the place in the *graphics* tag.

Transitions are described in *transition* tags, which are composed of a name and information about the visualisation. Places and transitions are connected through arcs, which are described in *arc* tags. Since all places and transitions can be identified via the *id* tag, arcs are able to reference them in *source* and *target* attributes.

PNML files can be imported and visualised by ProM 6.6. Since a PNML file contains all informations about the elements of a Petri net, it is possible to add additional places, transitions and arcs to it. Therefore, it is possible to add additional places manually to a process model constructed by ProM 6.6. As shown in Listing 4.2 a custom place named *Custom Place* with the ID *n11* is created and added to the PNML file. Moreover, an additional arc, as shown in Listing 4.3, with the ID *arc20* is created and added to the file. In the arc tag the source element *n11*, which is the *Custom Place*, and the target element *n6*, which is the *Approve or Deny Vacation Request* transition of Listing 4.1 are specified.

```
<place id="n11">
   <name>
      <text>Custom Place</text>
   </name>
   <toolspecific tool="ProM" version="6.4"/>
   <graphics>
      <position x="11.25" y="11.25"/>
      <dimension x="12.5" y="12.5"/>
   </graphics>
</place>
```

Listing 4.2: Additional place for a Petri net in EPNML

```
<arc id="arc21" source="n11" target="n6">
   <name>
      <text>1</text>
   </name>
   <toolspecific tool="ProM" version="6.4"/>
   <arctype>
      <text>normal</text>
   </arctype>
</arc>
```

Listing 4.3: Additional arc for a Petri net in EPNML

After adding the elements, the PNML file is imported into ProM 6.6. Then the resulting Petri net can be visualised by ProM, as shown in Figure 4.8. The Petri net has now an additional place, which is an input for the *Approve or Deny Vacation Request* transition. Therefore, the transition in this Petri net can only fire when both input places hold a token. This input place could also be used as a communication place in an oWFN or a workflow module. However, ProM 6.6. does not offer a possibility to display oWFNs

or workflow modules and their communication places correctly. The imported Petri net does not reveal which place is the start place of the WF-net and which place can be used for incoming messages. ProM 5.2., by contrast, supports oWFNs. In ProM 5.2. PNML files can be imported by selecting *File > Open PNML File > Without Log file*. Figure 4.9 shows ProM 5.2. and the resulting Petri net similar to the model in ProM 6.6. However, the names of the transitions are missing.

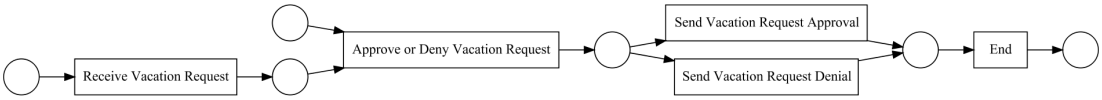

Figure 4.8.: Petri net with manually added place and arc

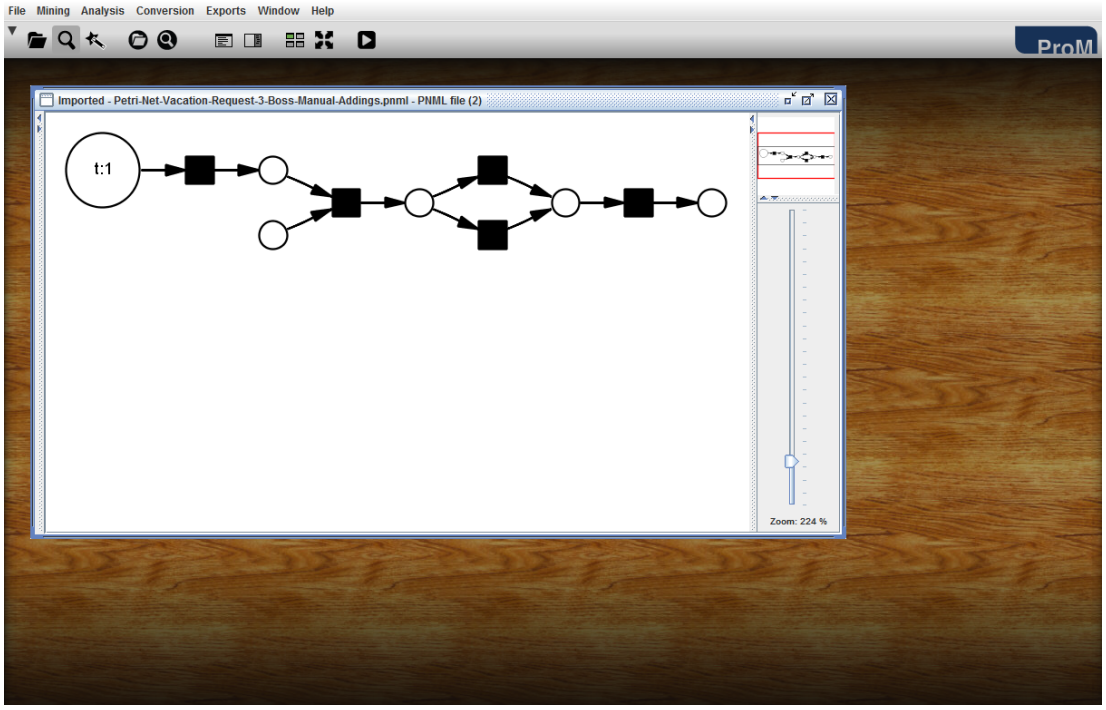

Figure 4.9.: Petri net with manually added place and arc in ProM 5.2.

Unfortunately ProM 5.2. is not able to import EPNML files generated by ProM 6.6. correctly without the corresponding event log. Moreover, it is not possible to import CSV or XES files. For that reason an XES log that was converted from a CSV file in ProM 6.6. has to be exported as an MXML[5] (Mining eXtensible Markup Language) file. Then the MXML log can be imported into ProM 5.2. by selecting *Mining > Open new log...*. After the import, a report of the log is shown, which is similar to the log view in ProM 6.6. Then the PNML file can be imported by selecting *File > Open PNML File > With Raw <filename>*. Afterwards the mappings between events found in the imported model (PNML file) and events in the log (MXML file) have to be configured, as shown in Figure 4.10. If the imported model will be used for further mining efforts the correct events should be selected, otherwise *Make Visible* may be selected. After confirming the mappings, the Petri net with an additional and manually added place is displayed.

[5] `http://www.processmining.org/logs/mxml`

Then the resulting Petri net can be converted to an oWFN by selecting *Conversion > Selected Petri net > Petri net to oWF net*. Afterwards, the *source place* and the *sink place* have to be configured. In this case the *Start* place and the *End* place. This ensures that all additional places of the Petri net will be treated as *communication places*. The result of this conversion with ProM 5.2. is shown in Figure 4.11, which illustrates the behaviour of the boss. Since any conversion is always done with a PNML file and an MXML log of a specific resource, for example a boss or an employee, the files are the only indicator which behaviour is illustrated by the resulting oWFN. It is not possible to provide specific information about the corresponding resource by using ProM and the methods applied.

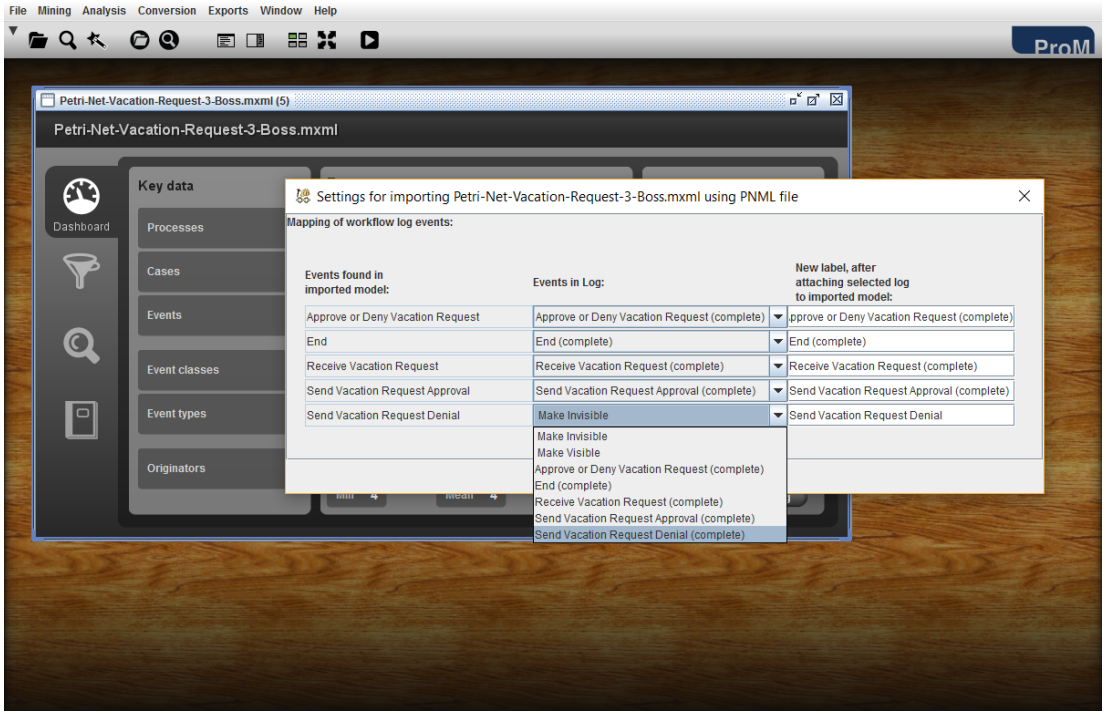

Figure 4.10.: PNML import: Mapping between PNML events and log events in ProM 5.2.

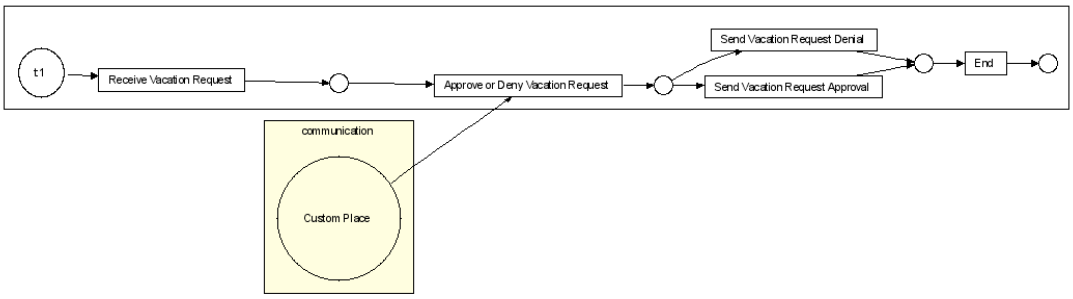

Figure 4.11.: oWFN of the Boss with a custom place used for communication generated by ProM 5.2.

4.4.2. From Event Log to oWFN or Workflow Module

As illustrated in the previous section, an oWFN or Workflow Module may be generated from an event log by applying some additional methods. It was shown that an extra place could be added to a Petri net, which could then be shown as an oWFN with one sink place, one source place and one communication place.

To retrieve two compatible oWFNs, which model the vacation request for the employee and the boss, the event logs have to be adjusted. Starting point is once again the simple event log of the vacation request, as shown in Figure 4.1. This log may be appropriate for mining process orchestrations, however, to model the communication between actors additional information has to be provided in the log. As proposed in Section 3.2.2, information about the *state*, the *message type* and possibly a *recipient* and a *sender* may be used to describe S-BPM processes and the communication between subjects. For this purpose the columns *State*, *Message Type*, *Recipient* and *Sender* are added to the simple log and the log is again split into two logs: one for the employee, as shown in Figure 4.12, and one for the boss, as shown in Figure 4.13.

Case Id	Timestamp	Activity	Resource	State	Message Type	Recipient	Sender
1	18.05.2017 08:00	Create Vacation Request	Employee	Function			
1	18.05.2017 09:00	Send Vacation Request	Employee	Send	Vacation Request	Boss	Employee
1	18.05.2017 10:05	Receive Vacation Request Response	Employee	Receive	Vacation Request Approval	Employee	Boss
1	18.05.2017 11:00	Go on vacation	Employee	Function			
1	18.05.2017 11:00	End	Employee	Function			
2	19.05.2017 08:00	Create Vacation Request	Employee	Function			
2	19.05.2017 09:00	Send Vacation Request	Employee	Send	Vacation Request	Boss	Employee
2	19.05.2017 10:05	Receive Vacation Request Response	Employee	Receive	Vacation Request Denial	Employee	Boss
2	19.05.2017 11:00	Stay at work	Employee	Function			
2	19.05.2017 11:00	End	Employee	Function			

Figure 4.12.: Advanced log of the vacation request process for the employee with two different cases

Case Id	Timestamp	Activity	Resource	State	Message Type	Recipient	Sender
1	18.05.2017 09:00	Receive Vacation Request	Boss	Receive	Vacation Request	Boss	Employee
1	18.05.2017 10:00	Approve or Deny Vacation Request	Boss	Function			
1	18.05.2017 10:05	Send Vacation Request Approval	Boss	Send	Vacation Request Approval	Employee	Boss
1	18.05.2017 10:05	End	Boss	Function			
2	19.05.2017 09:00	Receive Vacation Request	Boss	Receive	Vacation Request	Boss	Employee
2	19.05.2017 10:00	Approve or Deny Vacation Request	Boss	Function			
2	19.05.2017 10:05	Send Vacation Request Denial	Boss	Send	Vacation Request Denial	Employee	Boss
2	19.05.2017 10:05	End	Boss	Function			

Figure 4.13.: Advanced log of the vacation request process for the boss with two different cases

As described in Section 2.5, S-BPM provides three different states for an activity: *send, receive* and *function*. In contrast to a send or a receive state, a function state does neither send nor receive a message. Therefore, as it can be seen in the logs, rows with a function state type do neither have a value for the message type, nor a value in the sender or recipient column.

Rows with a send state type have to provide a value for the handled message type and a value for the recipient and the sender. Receive states type, by contrast, have to provide a value for the expected message type, the recipient and the sender. For an activity of the employee of state type send and a specific message type there is a counterpart of

type receive with the same message type in the log of the boss. Both have specified the employee as sender and the boss as recipient. For example: The activity of the employee with the name *Send Vacation Request*, the message type *Vacation Request*, the recipient *Boss* and the sender *Employee* and its complement activity in the log of the boss with the name *Receive Vacation Request*, the same message type and the same recipient and sender.

The message types will be used as communication places, since oWFNs or Workflow Modules require places with the same name to form compatible models. By using an output place with the name of the message type on one side, and using a corresponding input place on the other side, a compatible oWFN and therefore a process choreography can be generated.

Then following steps are performed in ProM 6.6. for both CSV logs:

1. Import the CSV file into ProM 6.6.
2. Convert the CSV file to an XES log, specifying the case identifier by using the *Case Id* and specifying the event identifier by using the *Activity*.
3. Start the *Alpha Miner* plugin and apply the α-algorithm on the XES log.

This results in the same orchestrations as previously shown in Figure 4.6 and Figure 4.7. Then two final steps have to be done in ProM 6.6.

4. Export the Petri net as EPNML files by selecting *Export to disk* in the *Workspace*.
5. Export the XES log as MXML file by selecting *Export to disk* in the *Workspace*.

The generated EPNML and MXML files of the boss (*Vacation-Request-4-Boss.pnml* and *Vacation-Request-4-Boss.mxml*) and the employee (*Vacation-Request-4-Employee.pnml* and *Vacation-Request-4-Employee.mxml*) can be found on the CD enclosed to this thesis.

Afterwards the EPNML file has to be equipped with additional places, which will form the communication interface. For every *Send* and *Receive* state a communication place has to be added. An outgoing communication place is linked to a transition, which was created from an activity with a send state, since a send state immediately sends a message. For example the transition *Send Vacation Request* has to be linked to the outgoing communication place *Vacation Request To: Boss From: Employee*. Ingoing or receive communication places, however, can not be linked to a transition that derived from a receive activity. Since a subsequent transition of a receive transition may only fire when a specific message is received, the communication place has to be linked to the corresponding subsequent transition. For example the activity *Receive Vacation Request Response* may receive either a *Vacation Request Approval* or a *Vacation Request Denial*, however, the subsequent activity *Go on vacation* can only be executed, when the *Vacation Request Approval* message has been received. Therefore a communication place *Vacation Request Approval From: Boss To: Employee* is linked to the transition *Go on vacation*. The communication place *Vacation Request Denial From: Boss To: Employee*, by contrast, has to be linked to the transition *Stay at work*. Since the communication place has to be linked to the subsequent activity the last state of the process may not be of type receive.

A subject in an S-BPM process may receive or send the same message type from or to different subjects and different paths could be taken in the process based on the sender or the receiver. Therefore it is necessary to state the sender and the recipient in

the name of the communication place, as this information is needed to map the oWFN to an S-BPM process model correctly.

For this purpose, the following algorithm has to be applied for every unique activity - state - message type - recipient - sender quintuplet in the EPNML file (*Petri net*, consisting of a set of transitions $\mathbb{T}$, a set of places $\mathbb{P}$ and a set of arcs $\mathbb{AR}$) by comparing it to its original log (*Log*, consisting of a set of activities $\mathbb{AC}$, each with a state sta and an optional message type mst), which has to be ordered by the case identifier and then by the event identifier.

1. Filter the ordered *Log* for different combinations of quintuplets Q, consisting of an activity act, a state sta, a message type mst, a recipient rcp and a sender sdr.
2. For every filtered combination of a quintuplet Q_i with state sta of type *Send* search for the transition T_i with name equal to the name of the activity act_{Name}. If the transition exists, add a place P_i with name of mst_i, the name of rcp_i and the name of sdr_i to the set of places $\mathbb{P}$. Then add an arc AR_i to the set of arcs $\mathbb{AR}$, specifying T_i as source, P_i as target and the message type mst_i as name.
3. For every filtered combination of a quintuplet Q_i with state sta of type *Receive*, take the subsequent quintuplet Q_s. Search for the corresponding transition T_i with name equal to the name of the subsequent activity act_s. If the transition exists, add a place P_i with the name of mst_i, the name of rcp_i and the name of sdr_i to the set of places $\mathbb{P}$. Add an arc AR_i to the set of arcs $\mathbb{AR}$, specifying P_i as source, T_i as target and the message type mst_i as name.

The positioning of the element must not be taken into account, since ProM 5.2. will rearrange all imported elements. For ProM 5.2. it is important that every custom place and arc gets a unique id.

By applying this algorithm on the event log of the employee (Figure 4.12) and the corresponding EPNML file, the following steps are performed:

1. In the first step, following quintuplets (activity, state, message type, recipient, sender) are identified: [*1: {Create Vacation Request, Function, undefined, undefined, undefined}, 2: {Send Vacation Request, Send, Vacation Request, Boss, Employee}, 3: {Receive Vacation Request Response, Receive, Vacation Request Approval, Employee, Boss}, 4: {Go on vacation, Function, undefined, undefined, undefined}, 5: {End, Function, undefined, undefined, undefined}, 6: {Receive Vacation Request Response, Receive, Vacation Request Denial, Employee, Boss}, 7: {Stay at work, Function, undefined, undefined, undefined}*]
2. For every quintuplet of state *Send* a place is created. For example: for the quintuplet *2: {Send Vacation Request, Send, Vacation Request, Boss, Employee}*, a place with the name *Vacation Request To: Boss From: Employee* is created, which is illustrated in Listing 4.4. Then the transition with the name of the activity *Send Vacation Request* is found in the EPNML file, which can be seen in Listing 4.5. Afterwards an arc has to be created with the id of the transition *n11* as source, the id of the place *cp1* as target and the name *Vacation Request*, as shown in Listing 4.6.
3. For every quintuplet of state *Receive*, take the subsequent quintuplet. For example: for the quintuplet *3: {Receive Vacation Request Response, Receive, Vacation Request Approval, Employee, Boss}*, the subsequent quintuplet is *4: {Go on*

vacation, Function, undefined, undefined, undefined}. Then a place with the name *Vacation Request Approval To: Employee From: Boss* is created, which is illustrated in Listing 4.7. Then the transition with the name of the subsequent activity *Go on vacation* is found in the EPNML file, which can be seen in Listing 4.8. Afterwards an arc has to be created with the id of the transition *cp2* as source, the id of the place *n9* as target and the name *Vacation Request Approval*, as shown in Listing 4.9.

```
<place id="cp1">
   <name>
      <text>Vacation Request To: Boss From: Employee<
         /text>
   </name>
   <toolspecific tool="ProM" version="6.4"/>
   <graphics>
      <position x="0" y="0"/>
      <dimension x="12.5" y="12.5"/>
   </graphics>
</place>
```

Listing 4.4: Place created for the message type ”Vacation Request”

```
<transition id="n11">
   <name>
      <text>Send Vacation Request</text>
   </name>
   <toolspecific tool="ProM" version="6.4" activity="
      Send Vacation Request" localNodeID="0720bc8f-022
      f-4cc3-857a-42e7845addd7"/>
   <graphics>
      <position x="137.5" y="70.5"/>
      <dimension x="25.0" y="20.0"/>
      <fill color="#FFFFFF"/>
   </graphics>
</transition>
```

Listing 4.5: Corresponding transition for the for the activity named ”Send Vacation Request”

```
<arc id="arc25" source="n11" target="cp1">
   <name>
      <text>Vacation Request</text>
   </name>
   <toolspecific tool="ProM" version="6.4"/>
   <arctype>
      <text>normal</text>
   </arctype>
</arc>
```

Listing 4.6: Arc between the transition ”Send Vacation Request” and the place ”Vacation Request To: Boss From: Employee”

```
<place id="cp2">
   <name>
      <text>Vacation Request Approval To: Employee
         From: Boss</text>
   </name>
   <toolspecific tool="ProM" version="6.4"/>
   <graphics>
      <position x="0" y="0"/>
      <dimension x="12.5" y="12.5"/>
   </graphics>
</place>
```

Listing 4.7: Place created for the message type ”Vacation Request Approval”

```
<transition id="n9">
   <name>
      <text>Go on vacation</text>
   </name>
   <toolspecific tool="ProM" version="6.4" activity="
      Go on vacation" localNodeID="c98a3ecf-c901-4744-
      bcbe-e3ea6aa72b1c"/>
   <graphics>
      <position x="312.5" y="88.0"/>
      <dimension x="25.0" y="20.0"/>
      <fill color="#FFFFFF"/>
   </graphics>
</transition>
```

Listing 4.8: Corresponding transition for the for the activity named ”Go on vacation”

```
<arc id="arc26" source="cp2" target="n9">
   <name>
      <text>Vacation Request Approval</text>
   </name>
   <toolspecific tool="ProM" version="6.4"/>
   <arctype>
      <text>normal</text>
   </arctype>
</arc>
```

Listing 4.9: Arc between the place ”Vacation Request Approval To: Employee From: Boss” and the transition ”Go on vacation”

By comparing the Petri nets of the *Boss* and the *Employee* with their corresponding Logs and by applying the algorithm, two adjusted EPNML files are generated, which are available on the CD enclosed to this thesis (*Vacation-Request-4-Boss-With-Communication.pnml* and *Vacation-Request-4-Employee-With-Communication.pnml*).

To visualise both process models following steps are performed in ProM 5.2.:

1. Import the MXML log by selecting *Mining > Open new log...*.
2. Import the EPNML file by selecting *File > Open PNML file > With: Raw <filename>*.
3. In the settings for the import, select *Make Visible* for every event in the log and click *Yes* in the warning pop-up.

The resulting Petri nets with communication places are shown in Figure 4.14 and Figure 4.15.

For converting the Petri nets to oWFNs following steps have to be performed:

4. Select *Conversion > Selected Petri net > Petri net to oWF net*.
5. In the options select *Start* as source and *End* as sink place.

The resulting oWFN of the employee is illustrated in Figure 4.16, the oWFN of the boss is shown in Figure 4.17.

Conclusion

By using the CSV to XES transformation plugin in ProM 6.6, an event log can be exported in MXML format, which can be imported in ProM 5.2. Moreover, ProM 6.6 can export discovered Petri nets to EPNML files. EPNML is used to structure Petri nets in an XML format. EPNML can be manipulated easily, which allows for adding custom places, transitions and arcs. An algorithm has been illustrated that adjusts the EPNML file so that communication places are added to the Petri net. For this purpose the algorithm uses the EPNML file and the original event log. It was stated that the algorithm imposes certain requirements on the log: First, the log has to be ordered by the case ID and the event ID. Second, the last activity of a subjects behaviour not to be of type *Receive*. Third, since the algorithm uses the name of the activity to find the corresponding transition in the EPNML file, unique activity names are essential. After importing the

manipulated EPNML file into ProM 5.2., the Petri net can be converted to an oWF net by specifying which places are the source and sink places.

Figure 4.18 shows a compact and manually created illustration of the oWFNs for the boss and the employee. As they are *syntactical compatible*, a *composed oWFN* can be produced manually, which is shown in Figure 4.19.

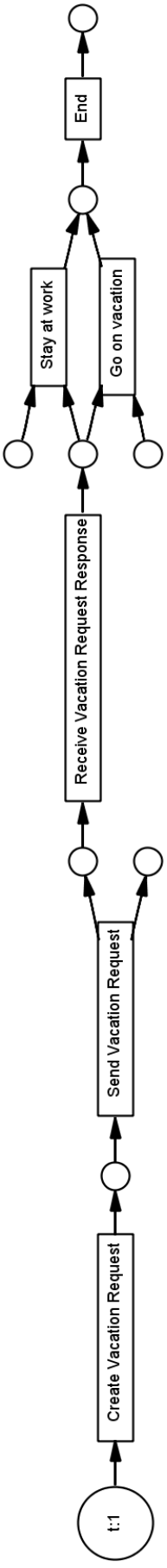

Figure 4.14.: Petri net with communication places for the vacation request process of the employee

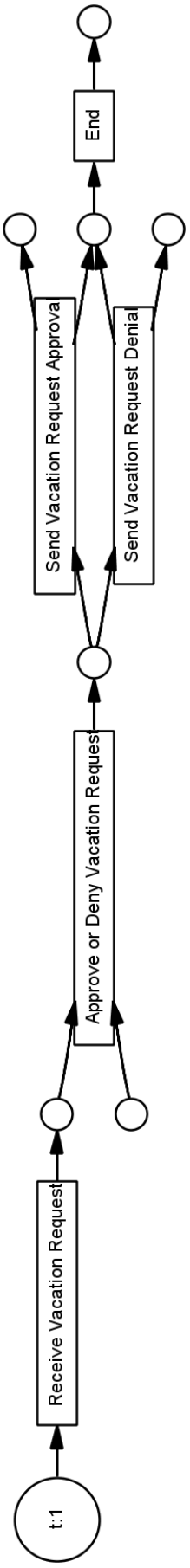

Figure 4.15.: Petri net with communication places for the vacation request process of the boss

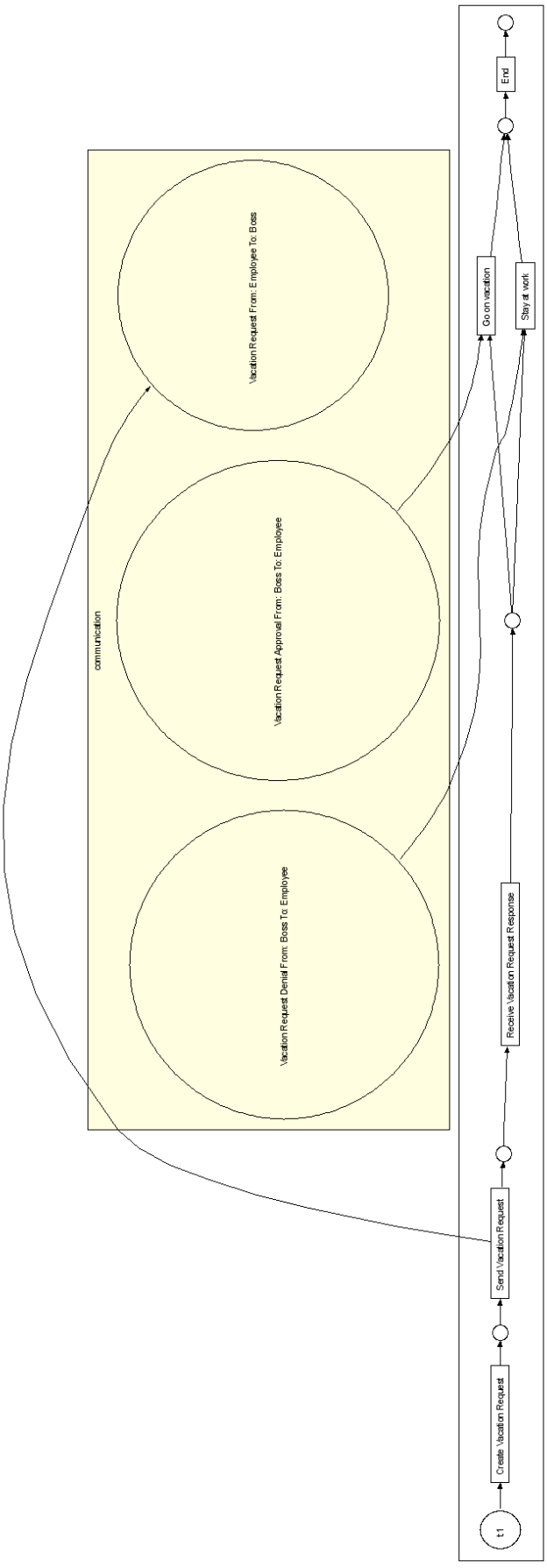

Figure 4.16.: oWFN for the Vacation Request Process of the Employee

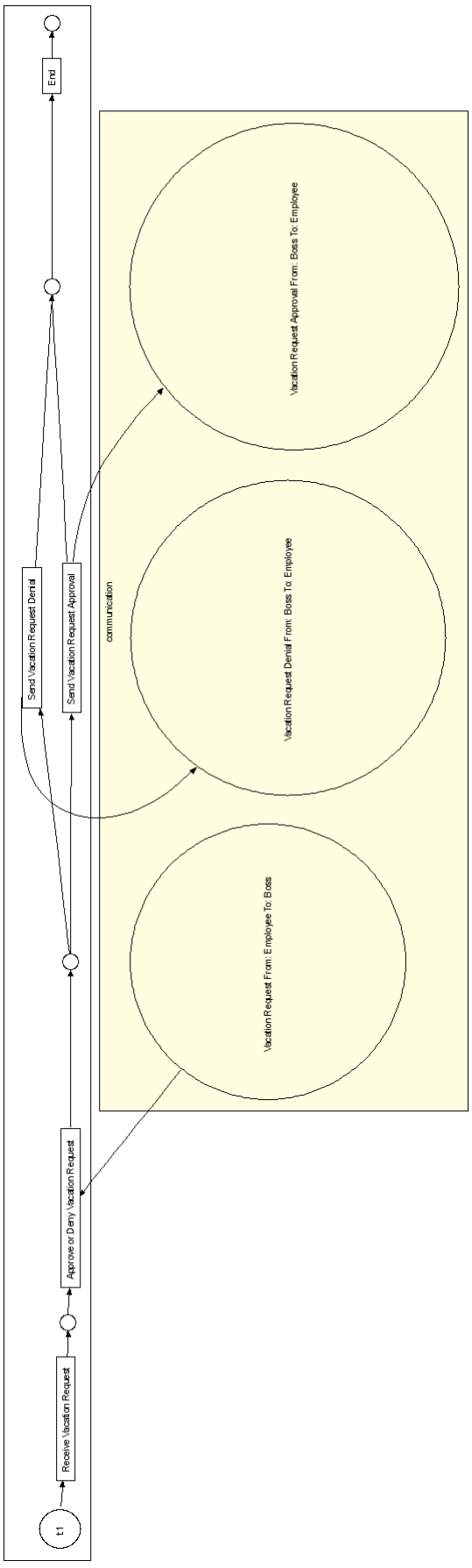

Figure 4.17.: oWFN for the Vacation Request Process of the Boss

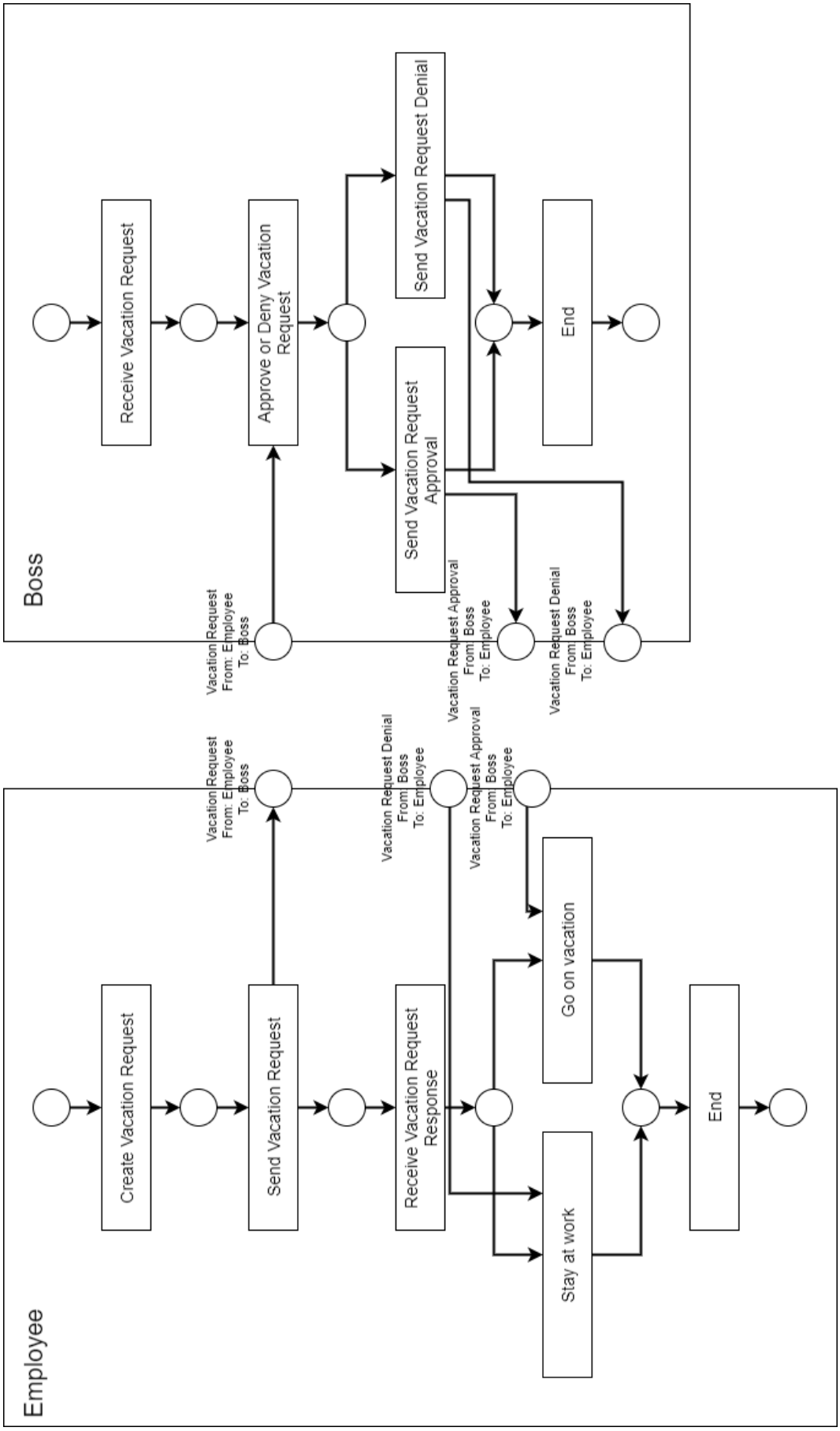

Figure 4.18.: Compact illustration of the oWFNs for the vacation request process

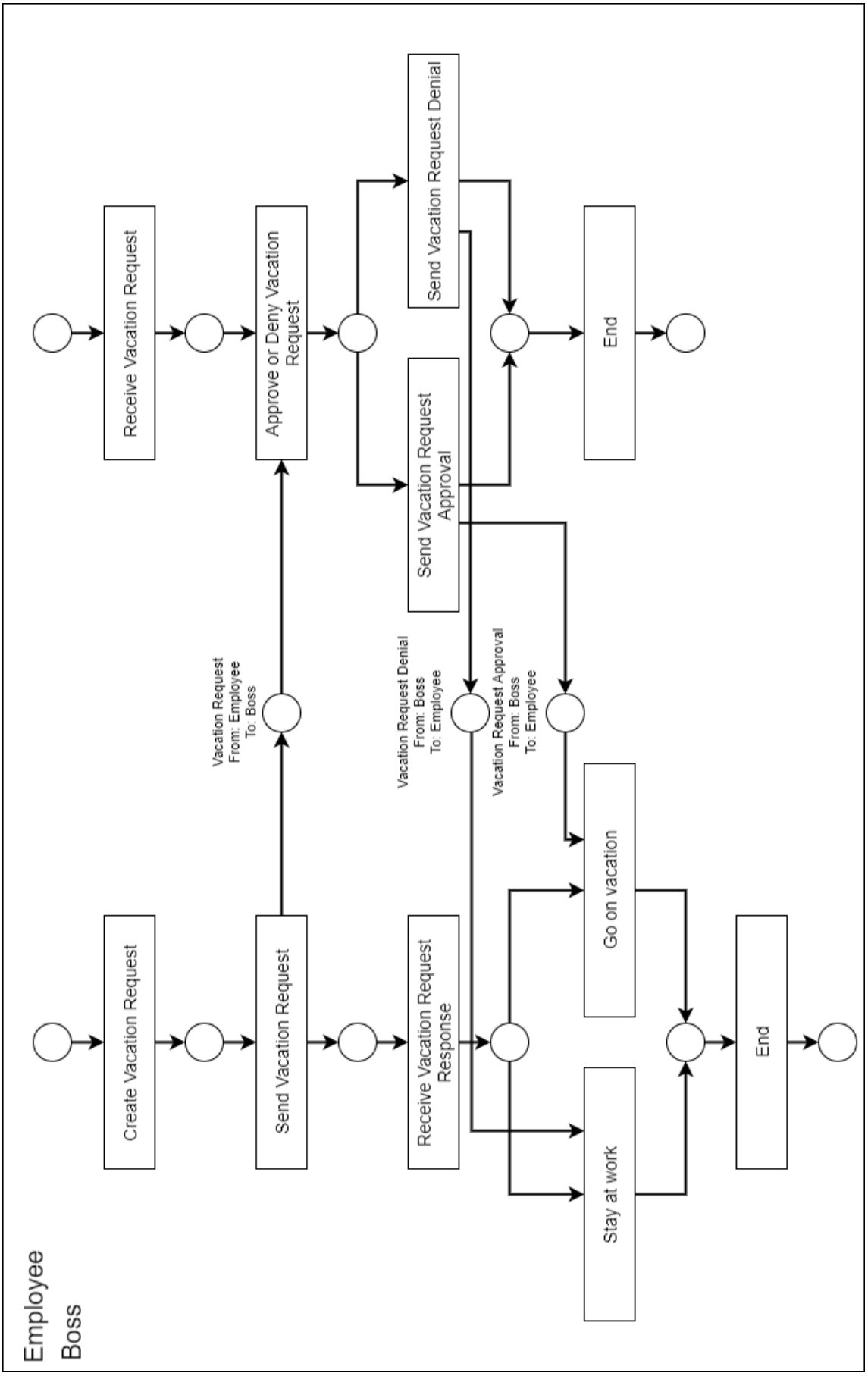

Figure 4.19.: Composed oWFN for the vacation request process

5. S-BPM Modelling and Execution Platform

In [8] the S-BPM Modelling and Execution Platform is presented as a *"reference architecture for the modeling and execution of business processes"*. This prototypical implementation is based on a microservice architecture and separates the modelling of a process of its execution. Figure 5.1 shows a top-level view of the project, which includes three major applications: a user interface for the process definition and modelling, a user interface for the management of processes and the workflow engine, where processes are executed [8].

The S-BPM Modelling and Execution Platform is an open-source project licensed under MIT license and available on Github[1][8].

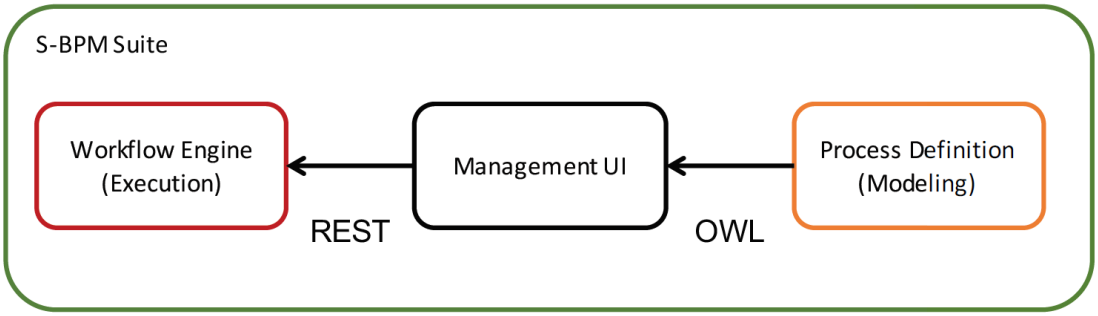

Figure 5.1.: Applications of the S-BPM Modelling and Execution Platform [8]

5.1. Technologies, Architecture and Services

The major technology used for this platform is Spring Boot[2], which enables the deployment of a microservice architecture based on Spring. The communication between the services is based on REST (Representational State Transfer) interfaces, which exchange data via JSON (JavaScript Object Notation). This allows for independent communication between internal and even external services. Process model data, process instance data and user data is stored in a MySQL database [8].

The complete architecture of the execution platform is visualised in Figure 5.2. The following list includes a short explanation of each service:

1. *Gateway*: The gateway forwards requests and responses to the correct recipient. Moreover, it is responsible for the user authentication [8].
2. *ServiceDiscovery*: This service is responsible for detecting devices and services automatically [7].

[1] `https://github.com/stefanstaniAIM/IPPR2016`
[2] `https://projects.spring.io/spring-boot/`

3. *ConfigurationService*: Configuration files of all components are stored centrally in the configuration service [7].
4. *ProcessModelStorage*: The process model storage stores all information about the process models. Process models specified in OWL (Web Ontology Language) files can be uploaded to this service, which will transform them to executable processes (see Section 5.2) [8].
5. *ProcessEngine*: The process engine is responsible for handling the workflow of a process from its start to its end. It is based on an Actor Model framework called Akka[3]. Akka provides a system of actors that act and interact via messages, which is suitable for the execution of S-BPM [8].
6. *GUI*: The graphical user interface is based on Angular2[4] and offers a management interface to this platform to import process models and to execute processes [8].
7. *ExternalCommunicator*: This service offers the possibility to use external projects in combination with the platform [7].
8. *Persistence*: The persistence service is responsible for the database mapping [7].

The modelling platform is a standalone application based on AngularJS and JointJS, which provides a functionality to export modelled processes as OWL files [8].

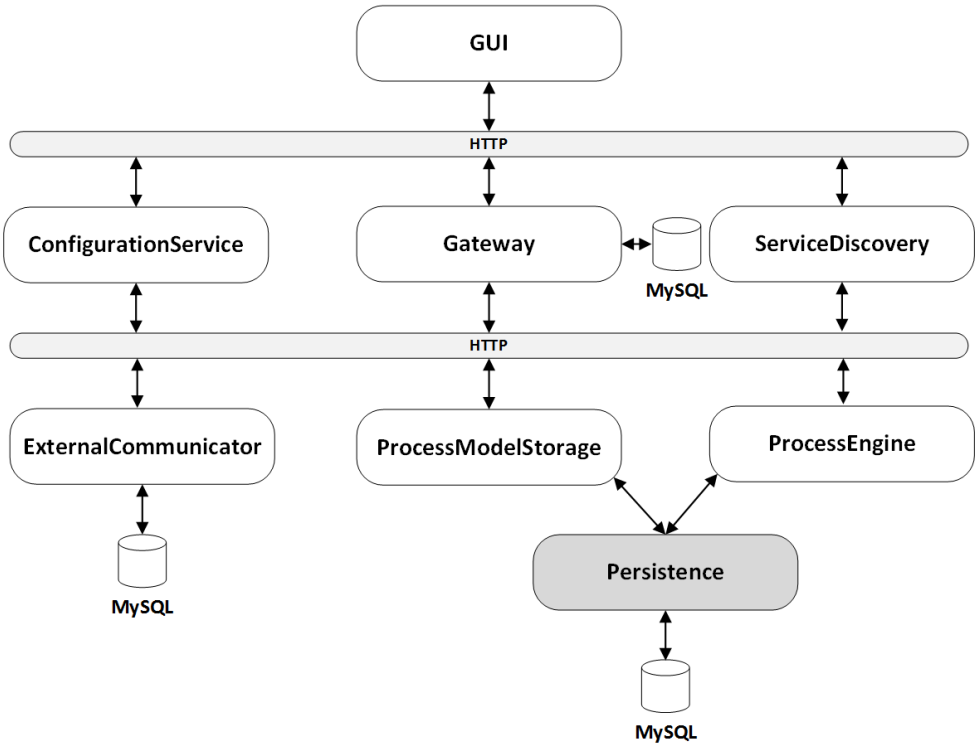

Figure 5.2.: Complete architecture of the S-BPM Execution Platform [16]

[3] http://akka.io/
[4] https://angular.io/

5.2. S-BPM Ontology Model

As mentioned before, the S-BPM Modelling and Execution Platform uses OWL files for the import and export of process models. The semantics of the S-BPM modelling notation are defined as an ontology by the Institute of Innovative Process Management (I2PM)[5] [8]. This definition is based on a working draft OWL ontology developed by Matthes Elstermann [5]. Moreover, the ontology is based on the ideas and the *S-BPM-ont* of Höver and Mühlhäuser [10], who proposed to use ontologies for exchanging process models between different process modelling and execution tools. The refined ontology of Elstermann is called *standard-pass-ont*, which specifies the description of a process based on the Parallel Activity Specification Schema (PASS), which is a subject-oriented process description language [3] [5].

The *ProcessModelStorage* service is capable of parsing OWL files via Apache Jena[6] and transforming them into executable process models. After uploading an OWL file via the *Management UI*, the process model, its subjects and their behaviour as well as the messages and the message exchange between the subjects is parsed. Then the result is serialised into JSON objects and sent back to the Management UI in order to finalise the import procedure. The user has to specify certain information, which cannot be designed in the modelling phase (e.g. the linking between subject models and actual user groups). Finally, the additional information, along with the JSON serialised process model, is uploaded to the process model storage, which stores a new process model in the database [8].

5.3. Event Logger Service

The S-BPM Execution Platform, in combination with the Management UI, offers the possibility to run processes e.g. the vacation request process. However, the platform lacks an event logging mechanism, which is necessary to perform any Process Mining tasks. For this purpose, a service, dedicated to log any event that occurs during the execution of process, has to be implemented. As shown in Figure 5.3, a new service has to be added to the system, to fit into the existing microservice architecture of the platform. Apart from logging (see Subesction 5.3.1), the EventLogger service will be responsible for generating log files in CSV format (see Subsection 5.3.2), for manipulating PNML files (see Subsection 5.3.3) and for generating OWL files (see Subsection 5.3.4).

5.3.1. Event Logging

As described in Section 3.2.2, an event-log suitable for mining S-BPM process models may contain information about the process instance, the activities, the subjects, the state types as well as about the message exchange. This results in following required columns: *Case Id, Timestamp, Activity, Resource, State, Message Type, Recipient, Sender* (see Figure 3.8). Moreover, the process model ID will be logged and since the log will

[5] `http://I2PM.net`
[6] `https://jena.apache.org/`

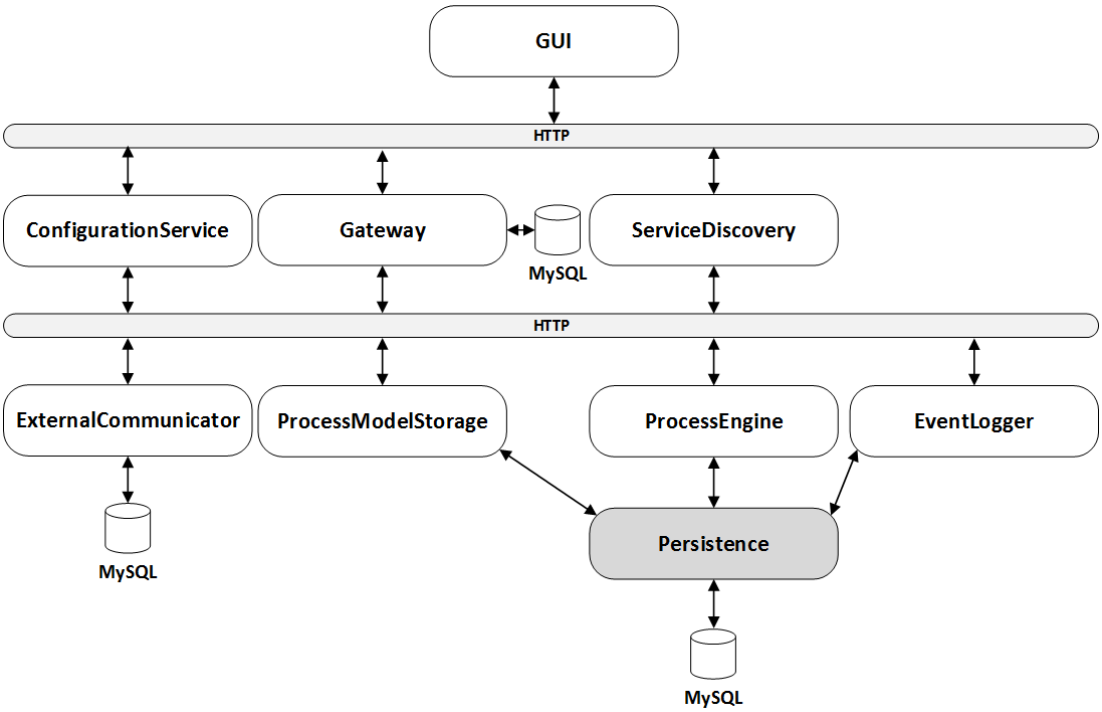

Figure 5.3.: Complete architecture of the S-BPM Execution Platform with added EventLogger service [16] (edited)

be stored in a relational database, a unique identifier column *Event Id* is required as well. For this purpose a new Java class *EventLogEntry*, containing all the properties mentioned above, is created in the *EventLogger* service. This entity will be automatically mapped to a database table with the corresponding structure by the *Persistence* service.

Five types of events will be logged:

1. The start of a process instance.
2. The end of a process instance.
3. Finishing an activity of state type *Function*.
4. Sending a message in an activity of state type *Send*.
5. Receiving a message in an activity of state type *Receive*.

Since all process steps are executed in the *ProcessEngine* service, the logging mechanism has to be triggered by the engine. As described in Section 5.1, all services communicate via REST interfaces. For this purpose a new interface has to be added to the *EventLoggerController* class of the *EventLogger*. This interface is illustrated in Listing 5.1, which handles post requests on the *newevent* path. Every post request has to provide a JSON object describing the event, which is serialised in the *EventLoggerDTO* object. After parsing all necessary properties, a new *EventLogEntry* is created and saved to the database.

```
@RequestMapping(value = "newevent", method =
    RequestMethod.POST)
public String newEvent(@RequestBody final
    EventLoggerDTO eventLoggerDTO) {
    final Long caseId = eventLoggerDTO.getCaseId();
    final Long processModelId = eventLoggerDTO.
       getProcessModelId();
    final String timestamp = eventLoggerDTO.
       getTimestamp();
    final String activity = eventLoggerDTO.getActivity
       ();
    final String resource = eventLoggerDTO.getResource
       ();
    final String state = eventLoggerDTO.getState();
    final String messageType = eventLoggerDTO.
       getMessageType();
    final String recipient = eventLoggerDTO.
       getRecipient();
    final String sender = eventLoggerDTO.getSender();

    final EventLogEntry eventLogEntry = new
       EventLogEntry(caseId, processModelId, timestamp,
        activity, resource, state, messageType,
       recipient, sender);
    eventLogRepository.save(eventLogEntry);
    return eventLogEntry.toString();
}
```

Listing 5.1: Implemented REST interface for logging new events

A new process instance is started in the *ProcessStartTask* class of the *ProcessEngine* service. After creating a new process instance and after assigning all information required, Akka notifies the sender about the successful start of the process. At the same time a new event log entry, which records the start of a new process instance, has to be created. The implementation is illustrated in Listing 5.2, which shows that a new *EventLoggerDTO* event object is created. This event contains information about the case, which is the ID of the process instance, the process model ID, the name ”Process Start” and the time of creation. As this is a special log entry about the process initialisation, there is no information about a state or a message exchange available. After the initialisation, the start subject of the process is set to its initial state in the *UserActorInitializeTask* class.

```
final ProcessInstance processInstance =
    processInstanceRepository.save((ProcessInstanceImpl
    ) processBuilder.build());
callback(processInstance.getPiId());

final ActorRef sender = getSender();
TransactionSynchronizationManager.
    registerSynchronization(new
    TransactionSynchronizationAdapter() {
    @Override
    public void afterCommit() {
      // start process
      sender.tell(new ProcessStartMessage.Response(
         processInstance.getPiId()), getSelf());
      final long caseId = processInstance.getPiId();
      final long processModelId = processInstance.
         getProcessModel().getPmId();
      final String activity = "Process Start";
      final String timestamp = DateTime.now().toString
         ("dd.MM.yyyy HH:mm");
      final EventLoggerDTO event =
          new EventLoggerDTO(caseId, processModelId,
             timestamp, activity, "", "", "", "", "");
      eventLoggerSender.send(event);
    }
});
```

Listing 5.2: Implemented logging functionality for new process instances

```
process.setState(ProcessInstanceState.FINISHED);
processInstanceRepository.save((ProcessInstanceImpl)
    process);

final long caseId = process.getPiId();
final long processModelId = process.getProcessModel()
    .getPmId();
final String activity = "Process End";
final String timestamp = DateTime.now().toString("dd.
    MM.yyyy HH:mm");
final EventLoggerDTO event = new EventLoggerDTO(
    caseId, processModelId, timestamp, activity, "", ""
    , "", "", "");
eventLoggerSender.send(event);
});
```

Listing 5.3: Implemented logging functionality for finishing process instances

When a process is completed, its state is set to finished in the *ProcessStateChangeTask* class of the *ProcessEngine*. Similar to the start event entry, an entry for the process end is added to the log, as shown in Listing 5.3.

The state change of a subject is logged in the *changeToNextState* method of the *StateObjectChangeTask* class. However, as shown in Listing 5.4, only the change to a state of type *Function* can be logged there, since there is no information about the message exchange available.

If a subject receives a message, its state changes and the *MessageReceivedTask* of the engine is executed. A receive event can be logged by retrieving information about the message flow between two subjects, which includes the message type as well as the sender of the message. This information can be extracted from the message flow, which is queried by defining the message flow ID, since it contains the name of the sender as well as the name of the corresponding message.

When sending a message, the method *TriggerSendInternal* of the *StateObjectChangeTask* class is called. As shown in Listing 5.6, the message type as well as the recipient can easily be retrieved from message flow of the current state.

```
1  if (subjectState.getCurrentState().getFunctionType()
       == StateFunctionType.FUNCTION) {
2      final long caseId = subjectState.
           getProcessInstance().getPiId();
3      final long processModelId = subjectState.
           getProcessInstance().getProcessModel().getPmId()
           ;
4      final String activity = subjectState.
           getCurrentState().getName();
5      final String timestamp = DateTime.now().toString("
           dd.MM.yyyy HH:mm");
6      final String resource = subjectState.getSubject().
           getSubjectModel().getName();
7      final String state = StateFunctionType.FUNCTION.
           name();
8      final String messageType = "";
9      final String recipient = "";
10     final String sender = "";
11
12     final EventLoggerDTO event = new EventLoggerDTO(
           caseId, processModelId, timestamp, activity,
           resource, state, messageType, recipient, sender)
           ;
13     eventLoggerSender.send(event);
14 }
```

Listing 5.4: Implemented logging functionality for state changes to type function

```
final long caseId = subjectState.getProcessInstance()
    .getPiId();
final long processModelId = subjectState.
    getProcessInstance().getProcessModel().getPmId();
final String activity = subjectState.getCurrentState
    ().getName();
final String state = StateFunctionType.RECEIVE.name()
    ;
final String resource = subjectState.getSubject().
    getSubjectModel().getName();
final String timestamp = DateTime.now().toString("dd.
    MM.yyyy HH:mm");

final MessageFlow messageFlow = messageFlowRepository
    .findOne(request.getMfId());
final String messageType = messageFlow.
    getBusinessObjectModels().get(0).getName();
final String recipient = resource;
final String msgSender = messageFlow.getSender().
    getName();

final EventLoggerDTO event = new EventLoggerDTO(
    caseId, processModelId, timestamp, activity,
    resource, state, messageType, recipient, msgSender)
    ;
eventLoggerSender.send(event);
```

Listing 5.5: Implemented logging functionality for receiving messages

```
final long caseId = subjectState.getProcessInstance()
    .getPiId();
final long processModelId = subjectState.
    getProcessInstance().getProcessModel().getPmId();
final String activity = subjectState.getCurrentState
    ().getName();
final String state = StateFunctionType.SEND.name();
final String resource = subjectState.getSubject().
    getSubjectModel().getName();
final String timestamp = DateTime.now().toString("dd.
    MM.yyyy HH:mm");
final MessageFlow messageFlow = messageFlowRepository
    .findOne(subjectState.getCurrentState().
    getMessageFlow().get(0).getMfId());
final String messageType = messageFlow.
    getBusinessObjectModels().get(0).getName();
final String recipient = subjectState.getCurrentState
    ().getMessageFlow().get(0).getReceiver().getName();
final String msgSender = resource;

final EventLoggerDTO event = new EventLoggerDTO(
    caseId, processModelId, timestamp, activity,
    resource, state, messageType, recipient, msgSender)
    ;
eventLoggerSender.send(event);
```

Listing 5.6: Implemented logging functionality for sending messages

Conclusion

For the logging of process events a new REST interface, used for communication, was added to the *EventLogger* service. The *ProcessMining* service uses this interface in order to log events about the start, the end, the message exchange and state changes of a process instance. The logging mechanism has to be triggered in the methods of the corresponding Akka tasks of the engine.

Every process instance has a unique identifier, which is used for the *Case ID* in the log. The name of the subject state is used as activity name and the state function type (function, send or receive) is mapped to the state column. The name of the subject involved is used for the resource column. If a state is of type send or receive, information about the message type, the sender and the receiver is logged as well.

5.3.2. Generating Log Files

Logging process events results in MySQL database entries, however, for applying Process Mining methods, event log files are necessary. As described in Chapter 4, ProM 6.6. offers the possibility to import event logs in CSV format. For this purpose a CSV export mechanism was implemented in the *EventLogger* service. As discussed in Section 4.3, the event log has to be split into separate logs for each subject, in order to retrieve a single orchestration for each subject. For this purpose another REST interface had to be implemented, as shown in Listing 5.7. This interface takes a process model ID and a subject name as input for generating an CSV event log file. The *EventLogService* calls the *EventLogRepository* to query the database for the corresponding entries, as shown in Listing 5.8. The query selects all entries with the given process model ID and the given subject name. However, only log entries of finished process instances are included, which is ensured by the *EXISTS* clauses of the query. This is necessary due to the fact, that the algorithm proposed in Section 4.4.2 requires a specific end entry in the log in order to work properly. Furthermore, the log will be sorted by the case ID and event ID as this is required by the algorithm. Finally, the result of the query will be transformed into a CSV formatted file by using the Super CSV[7] package in the *downloadCSV* method of the *EventLoggerController* class.

```
1  @RequestMapping(value = "eventlogCSV/{processModelId
       }/{subject}", method = RequestMethod.GET)
2  public @ResponseBody void getEventLogCSV(final
       HttpServletRequest request, @PathVariable("
       processModelId") final int processModelId,
       @PathVariable("subject") final String subject,
       final HttpServletResponse response)
3      throws IOException {
4          List<EventLoggerDTO> events;
5
6      try {
7          events = eventLogService.
               getEventLogForProcessModelAndSubject(
               processModelId, subject).get();
8          this.downloadCSV(response, events,
               processModelId, subject);
9      } catch (final Exception e) {
10         e.printStackTrace();
11     }
12 }
```

Listing 5.7: Implemented functionality for retrieving a CSV log

[7] https://super-csv.github.io/super-csv/

```
@Repository
public interface EventLogRepository extends
    CrudRepository<EventLogEntry, Long> {
    @Query(value = "SELECT * FROM event_log AS log
       WHERE log.process_model_id = :processModelId AND
        log.resource = :subject AND EXISTS((SELECT 1
       from event_log AS x " +
       "WHERE x.case_id = log.case_id AND x.activity =
           'Process Start' AND x.message_type = '' AND
         x.resource = '' AND x.state = '')) " +
       "AND EXISTS((SELECT 1 from event_log AS y WHERE
           y.case_id = log.case_id AND y.activity = '
         Process End' AND y.message_type = '' AND y.
         resource = '' AND y.state = ''))" +
       "ORDER BY log.case_id, log.event_id",
         nativeQuery = true)
    public List<EventLogEntry>
       getEventLogForProcessModelAndSubject(@Param("
       processModelId") int processModelId, @Param("
       subject") String subject);
}
```

Listing 5.8: Implemented functionality for querying the database for the log of a process model and subject

Conclusion

Another REST interface was implemented in the *EventLogger* service, to provide a possibility to download a log file in CSV format for a specific process model and a specific subject. The generated CSV log file contains a list of completed process instances only, ordered by the case ID and then by the event ID, which is required for further manipulation. After downloading a CSV file, it can be used for Process Mining e.g. by using ProM.

5.3.3. Adding the Communication Perspective

As discussed in Section 4.4.2, ProM can generate EPNML files based on CSV logs. However, to retrieve process orchestrations based on the exchange of messages, communication places have to be added in order to generate an oWF net or a workflow module. For this purpose an algorithm was presented. This algorithm was also implemented into the *EventLogger* service, which provides the possibility to upload EPNML files and to equip them with additional communication places based on the original log. Therefore, another REST interface was added to the *EventLoggerController*, which takes the PNML file content and the content of the corresponding CSV log as input, as shown in Listing 5.9. The controller calls the *manipulatePNML* method of the manipulation service, which is listed in Appendix A.1.

```
@RequestMapping(value = "manipulatePNML", method =
    RequestMethod.POST)
public @ResponseBody void manipulatePNML(@RequestBody
     final Map<String, String> fileContents,
    final HttpServletRequest request, final
       HttpServletResponse response) throws IOException
        {
    final String pnmlContent = fileContents.get("
       pnmlContent");
    final String csvLog = fileContents.get("csvLog");
    try {
       final StreamResult result =
          manipulatePNMLService.manipulatePNML(
          pnmlContent, csvLog);
       downloadXML(response, result);
    } catch (final Exception e) {
       response.sendError(400, e.getMessage());
    }
}
```

Listing 5.9: Implemented REST interface for manipulating an EPNML file

The manipulation service implements the proposed algorithm of Section 4.4.2. First, the *parseCSV* method parses the uploaded CSV log for the unique quintuplets consisting of the activity name, the state type, the message type, the recipient and the sender. Then, for every entry with state type send or receive of the quintuplet set, a message place is added to the uploaded EPNML file. Moreover, an arc, which links the message place to the corresponding transition, is added. This is done by using the Java API for XML Processing[8]. It is important to note that every communication place is equipped with a special *toolspecific* element, which can be used to add additional information that is not used by other tools and that does not influence the Petri net [11]. The toolspecific element specifies the message name, the state type (send or receive), the recipient and the sender of the message. This additional information is necessary for a possible OWL generation, which will be discussed in Section 5.3.4. After adding all the necessary places, arcs and information to EPNML file, the resulting file is sent back to the user who initiated the request.

Conclusion

Based on the proposed algorithm, a PNML file is manipulated and equipped with communication places based on the underlying event log. Therefore, another REST interface was implemented into the *EventLogger* service.

[8] https://docs.oracle.com/javase/8/docs/technotes/guides/xml/jaxp/index.html

5.3.4. Reconstructing the Process Model

In Section 3.1 the terms *Play-Out*, *Play-In* and *Replay* were explained. Since an event logging mechanism was introduced, the S-BPM Platform is now able to ”play-out” process model behaviour in the form of an event log. Moreover, as the generated event log can be used as input for discovering a process model e.g. by using ProM, it is possible to ”play-in” a log to construct a Petri net. However, the platform is not able to reconstruct an *executable* process model by default. To overcome this deficiency, a mechanism was developed, which transforms Petri nets into an OWL process model, following the S-BPM modelling notation. The Petri nets may be discovered by ProM but have to be manipulated and equipped with communication places by the platform.

For this purpose another REST interface was added to the *EventLogger* service. This interface, as shown in Listing 5.10, takes the process model name and the EPNML file of each associated subject as input. Then the *generateOWL* method of the *generateOWLService* is called, which is shown in Appendix A.2.

```
1 @RequestMapping(value = "generateOWL", method =
    RequestMethod.POST)
2 public @ResponseBody void generateOWL(@RequestBody
    final GenerateOWLPostBodyHelper requestBody, final
    HttpServletRequest request, final
    HttpServletResponse response) throws IOException {
3    try {
4    final StreamResult result = generateOWLService.
       generateOWL(requestBody.getProcessModelName(),
       requestBody.getPnmlFiles());
5       downloadXML(response, result);
6    } catch (final Exception e) {
7       response.sendError(400, e.getMessage());
8    }
9 }
```

Listing 5.10: Implemented REST interface for generating an OWL file

This service creates a new XML document, which will be the resulting OWL file. At the beginning, the OWL skeleton, which specifies and imports the underlying ontology structures, is added to the document. Then, the process model element *PASSProcessModel* is added, which will contain the main information about the model e.g. the name, the subjects and the messages. Moreover, the *ModelLayer* element is appended, which is the equivalent to the Subject Interaction Diagram (SID) of S-BPM. Afterwards, the subjects are created by using their name, which have to be provided along with the uploaded Petri net files. Then, for every uploaded PNML file, the messages, which are defined by the communication places, will be added to the OWL file. The *toolspecific* information of the communication place, which has been discussed in Section 5.3.3, is used to retrieve the state type (receive or send), the message name, the recipient and the sender. Besides defining the messages, the corresponding message exchange elements are created as well. Afterwards, the states are added to the OWL structure, based on the transitions of the PNML file. Since transitions in a Petri net are connected

through an intermediate place, they are not directly linked. S-BPM states, however, are directly connected. Therefore, the direct links between two transitions have to be discovered by identifying and removing the intermediate places. The type of the state (receive, send or function) is discovered by checking whether the connecting arc has a communication place defined as target (which identifies a send state) or as source (which identifies a receive state). Moreover, it is determined if the state is the initial or end state, by checking if the corresponding transition is not the target of any other transition (which identifies the initial state) or if the the transition is not the source of any other transition (which identifies the end state). This indicates, that the first state and last state of a subject behaviour may not be revisited in a process instance. After creating the states, the OWL transitions can be created by using the direct connections, based on the PNML arcs, created before. Finally, the resulting OWL document is sent back to the user.

Conclusion

Manipulated PNML files, which are equipped with communication places, can be uploaded to the *EventLogger* service that transforms them into a process model defined in an OWL file. This generated OWL file can be imported (”play-in”) into the S-BPM Execution Platform. By executing and testing the imported process model, which was discovered by applying Process Mining methods, a type of *conformance checking* and therefore a ”replay” can be performed.

5.3.5. Other Adaptations

As shown in Section 5.1, the S-BPM Platform offers a *Management UI* service that provides a graphical interface for all functionalities of the platform. For the implementations described in the previous sections, corresponding views and controllers have been added to the GUI (developed in the GUI-Dev package, see [7]):

1. *Event logs*: The GUI offers a possibility to view event logs for a process model and a subject directly in the GUI. Moreover, a CSV file can be generated. For this purpose the *eventLogger* component was added to the GUI.
2. *PNML manipulation*: To equip PNML files with communication places, a page was implemented to upload a PNML file together with the corresponding CSV log. The manipulated PNML can be downloaded. Therefore, the *manipulatePNML* package was added to the GUI.
3. *OWL generation*: An OWL file can be generated by specifying the process model name, the name of the subjects and their PNML files in a separate view. This was implemented in the *generateOWL* component of the GUI.

Moreover, the *ProcessModelStorage* service, which offers the possibility to import process models via OWL files, was updated to support the OWL standard in version 0.7.5. For this purpose a version (0.7.2 or 0.7.5) has to be provided when uploading an OWL file. Based on the version, the *OWLParser* uses different identifiers for certain elements that have been changed in the S-BPM OWL specification.

Finally, the Git repository [7] was updated.

6. Modelling, Execution & Process Mining in the Platform

The goal of this chapter is to illustrate the capabilities of the S-BPM Modelling and Execution Platform based on the implementations presented in Chapter 5. First, the modelling and import of a process model will be outlined. Then, the process will be executed and the resulting event log will be shown. Third, ProM will be used to discover process models as Petri nets based on the generated event log. Afterwards, the resulting Petri nets will be manipulated and equipped with communication places in order to transform them into Open Workflow Nets. The manipulated PNML files will then be transformed into an OWL file defining a Process Model via the *standard-pass-ont*. Finally, the ontology will be imported into the platform and the discovered process model will be evaluated.

6.1. Process Modelling and Import

The process modelling service provides a user interface to model S-BPM processes, which can then be downloaded as an OWL file, specifying the process model in the *standard-pass-ont* in version 0.7.2 (see 5.2). The process, that will be used in this chapter, is a travel request process between a customer and the travel agency. Figure 6.1 shows the SID view of the travel request process in the modelling platform. As illustrated, the process is composed of two interacting subjects, the customer and the travel agency. Both send and receive messages, which can be seen in the properties column on the right side.

The subject behaviour of the customer is visualised in Figure 6.2. The process starts by creating a travel request, which is afterwards sent to the travel agency. After receiving a proposal, the customer can decide whether to accept or to decline the offer. If the customer accepts the proposal, he or she sends the selected offer to the travel agency. Afterwards, the customer has to provide the credit card information to the agency. If the credit card information is valid, the booking will be confirmed by the agency. If the credit card could not be charged, the customer can decide whether to provide new credit card information or to cancel the booking process.

The subject behaviour diagram of the travel agency, as illustrated in Figure 6.3, shows the opposite view of the process. The process of the agency starts by receiving a travel request, which is answered with a proposal. If the offer is declined, a confirmation is sent back and the process ends. If the offer is accepted, the credit card information of the customer is requested. Afterwards, the trip is booked. If the booking was successful, a confirmation is sent to the customer and the process ends. If the credit card was rejected, the customer can either provide new information or cancel the request.

If the process model is finished, the model can be downloaded. The resulting OWL file (*TravelRequest.owl*) is provided on the CD enclosed to this thesis. This file can now

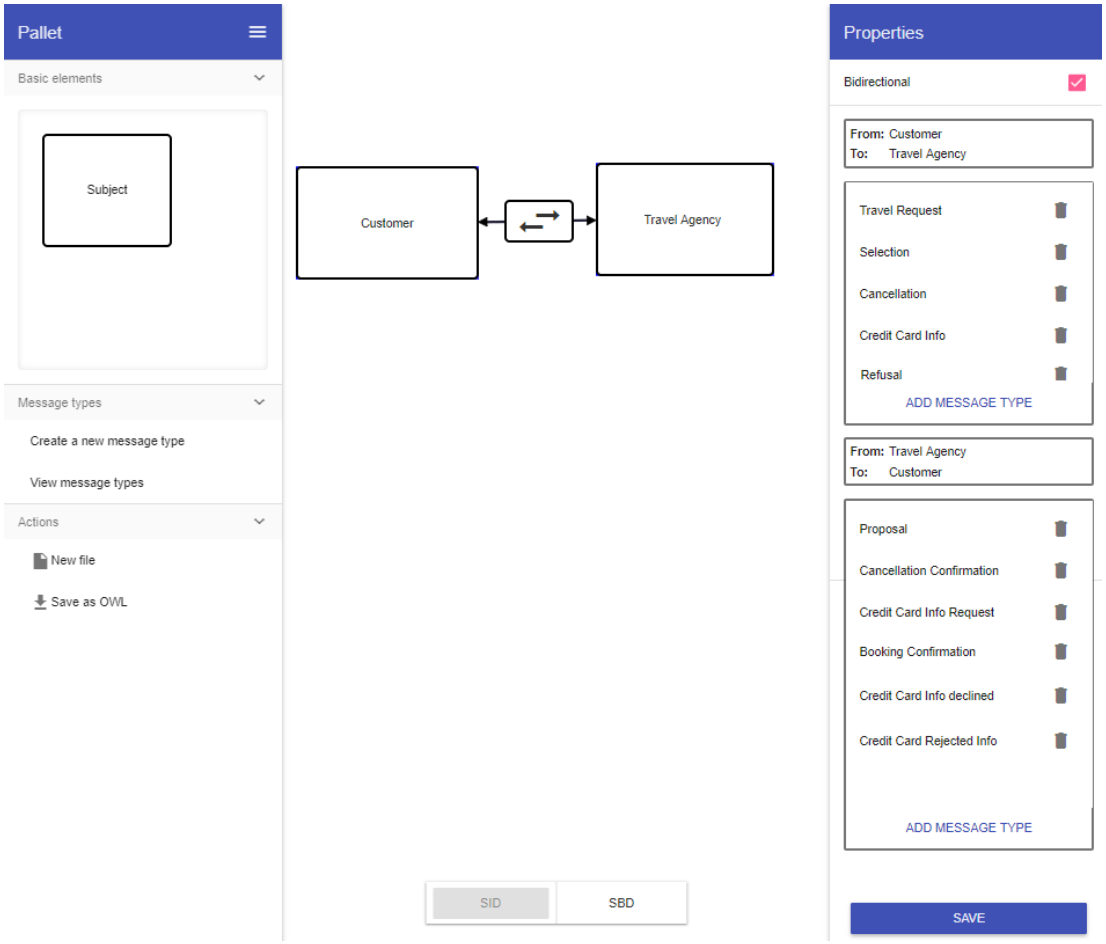

Figure 6.1.: SID view of the travel request process in the modelling platform

be uploaded to the process model storage by using the management UI. As described in Section 5.2, the OWL does not provide enough information for executing it in the platform. Therefore, the user has to specify certain information, as shown in the Figures 6.4, 6.5 and 6.6. The first picture shows, that the name, the description and a version number can be specified. Then, the corresponding rule group has to be selected for every subject, in order to determine which users of the platform can execute the process. Moreover, the starting subject has to be selected. The second picture shows the business objects, which were specified as messages in the modelling platform. The user is able to add fields of different types to the business object, which can be used for data exchange. For example, when sending the travel request, the customer should specify the date and the destination of the journey. Picture three shows that the access rights have to be specified for each field and for each state in which the business object is used. As illustrated, the customer must specify the destination in the *Send Travel Request to Travel Agency* state. The travel agency is only allowed to read the specified data. Finally, the process model can be uploaded and as a result, the model can be executed in the platform.

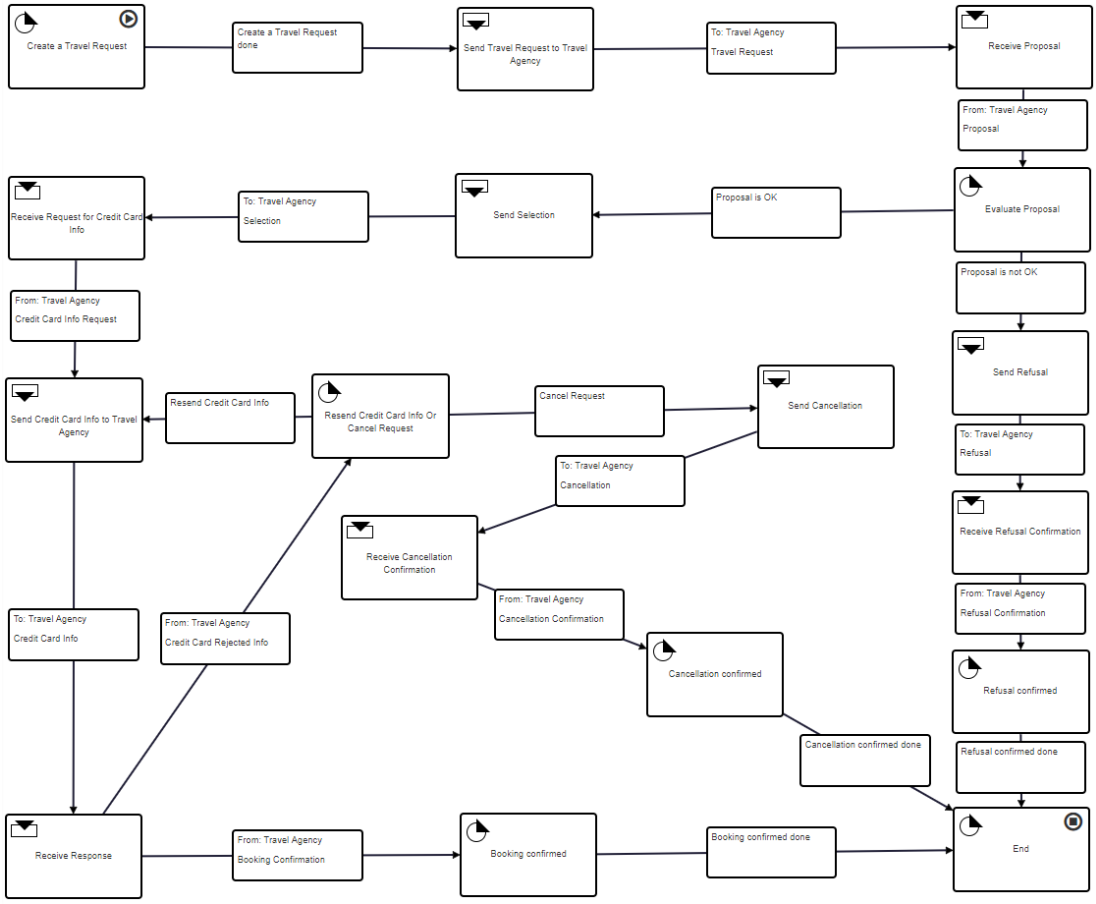

Figure 6.2.: SBD of the customer in the travel request process

6.2. Process Execution and Logging

The imported process is now available for execution in the platform. The management UI provides a guided interface through the process, by providing the fields that were specified in the import step. Moreover, if multiple execution paths are possible, the user is able to decide which path should be taken. Figure 6.7 shows a view of the *Send Proposal* state of the travel agency. As shown, the travel agency is able to offer two proposals. By selecting the next state *Receive Answer*, the message is sent to the customer and the travel agency has now to wait for the answer. The customer receives the proposal, as shown in Figure 6.8. Then the customer selects the next state *Evaluate Proposal*, in order to send an answer back to the travel agency.

The subject behaviour diagrams of the customer and the travel agency (Figures 6.2 and 6.3) reveal, that there are multiple process paths possible:

1. The customer declines the offer.
2. The customer accepts the offer and the booking is successful at the first attempt.
3. The customer accepts the offer, the booking is not successful, the customer updates the credit card information and then the booking is successful.
4. The customer accepts the offer, the booking is not successful and the customer decides to cancel the request.

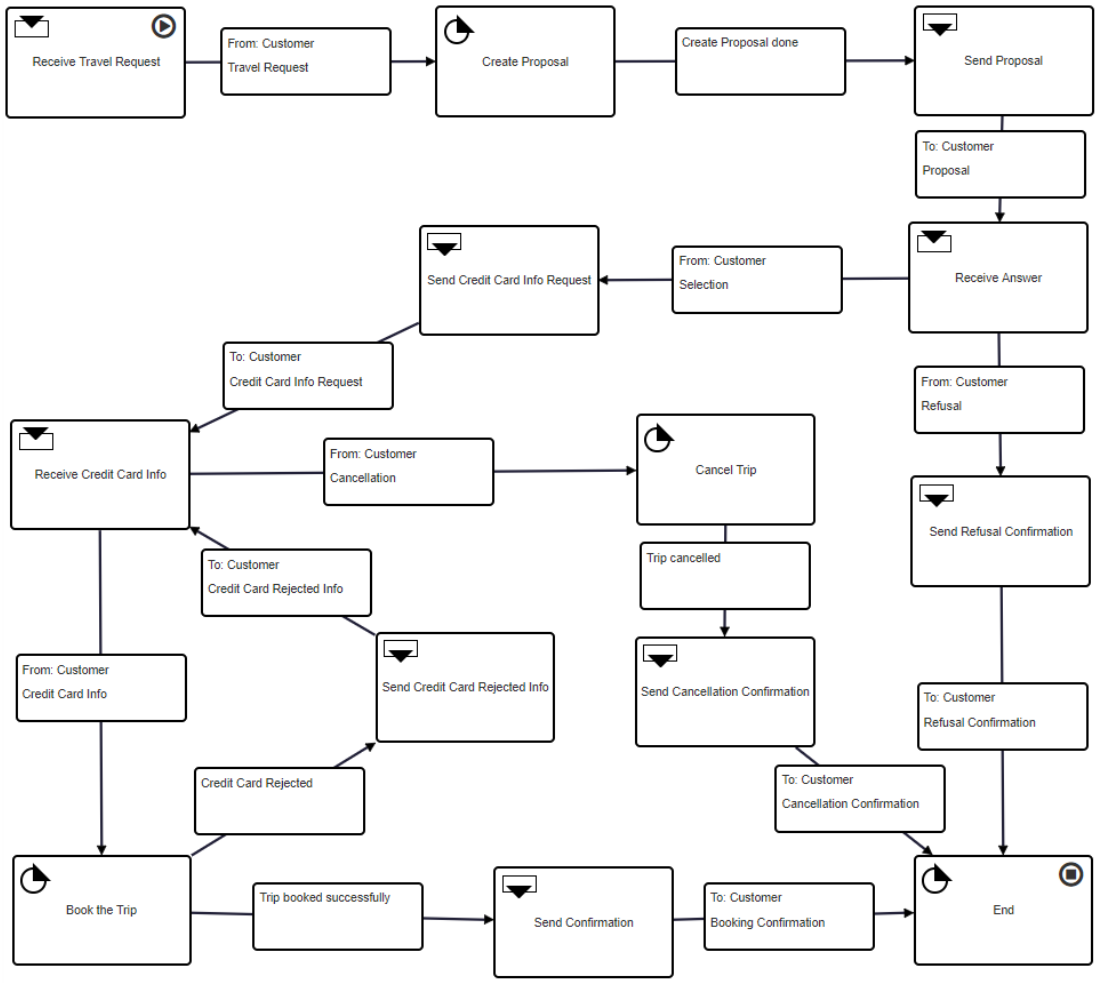

Figure 6.3.: SBD of the travel agency in the travel request process

In fact, there are indefinite process paths, since the credit card info could be rejected, the customer resends the credit card info which is then again rejected and so on. However, the goal of this demonstration is to show that the S-BPM Platform is able to reconstruct the whole process model based on Process Mining techniques. Therefore, the process is executed four times. Each time another path of the four paths listed above is taken. This results in an event log, as partially shown in Figure 6.9. In order to retrieve an event log, the user has to specify the desired process model and the desired subject. The log can be viewed directly in the platform, but due to the fact that the log is needed to perform Process Mining methods, a CSV file can be downloaded as well. The generated event logs in CSV format of the customer (*Eventlog_1_Customer_20082017-1031.csv*) and the travel agency (*Eventlog_1_Travel Agency_20082017-1031.csv*) are available on the CD enclosed to this thesis.

6.3. Process Mining and oWFN Transformation

As described in Chapter 4, ProM can be used for Process Mining. The logs of the customer and the travel agency generated in 6.2 are used as the starting point the

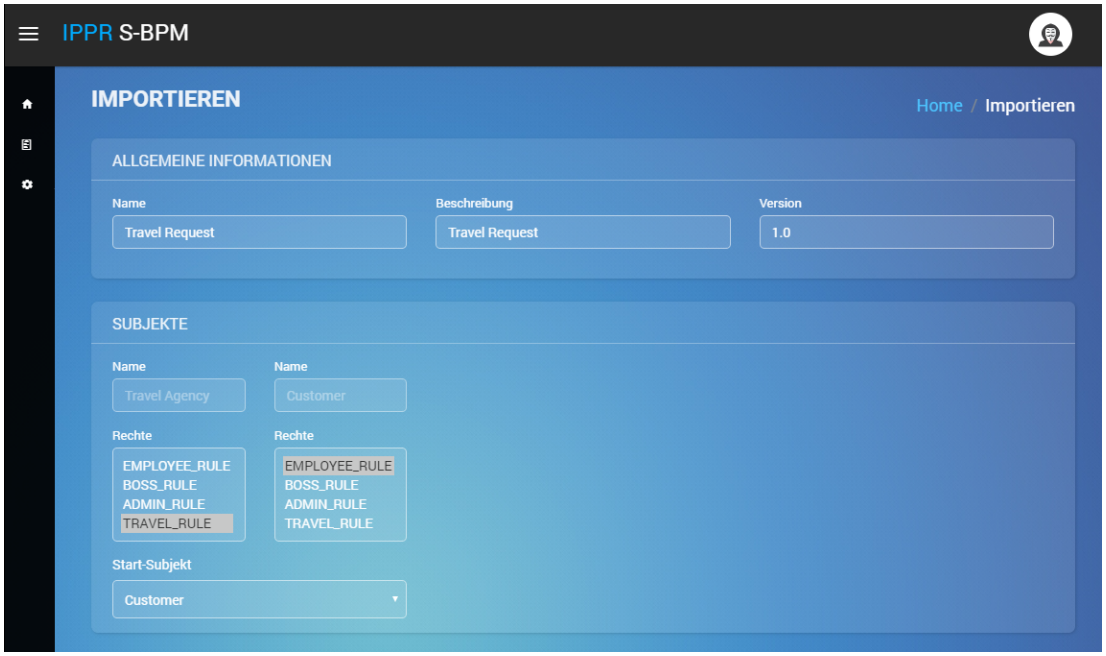

Figure 6.4.: Import of the travel request process: specifying process and subject information

mining efforts. First, ProM 6.6. is used to transform a CSV log to an MXML log, which is necessary for further usage (see Section 4.2). Then, the α algorithm is applied to the MXML log to generate a Petri net representing the process model. The generated Petri nets are visualised in Figures 6.10 and 6.11, which are manually drawn based on the graphical outcome of the α algorithm of ProM 6.6. Moreover, the resulting PNML files for the customer (*Eventlog_1_Customer_20082017-1031.pnml*) and the travel agency (*Eventlog_1_Travel Agency_20082017-1031.pnml*) can be found on the CD enclosed to this thesis.

The Petri nets visualise the behaviour of the subjects correctly, however, the communication perspective, which is the central part of S-BPM processes, is missing. Therefore, the PNML files have to be manipulated and equipped with communication places, as described in Sections 4.4.1 and 4.4.2. For this purpose, the management UI offers an interface for uploading a Petri net along with its original CSV log. After uploading the files, two manipulated Petri nets are retrieved, which are available on the CD enclosed (*Eventlog_1_Customer_20082017-1031-manipulated.pnml* and *Eventlog_1_Travel Agency_20082017-1031-manipulated.pnml*). Each manipulated PNML file, along with the MXML log, could now be imported to ProM 5.2. and transformed to an oWFN, as described in Section 4.4.2. The result is shown in Figure 6.12. It shows a composed oWFN of both subjects (manually created).

6.4. OWL Transformation and Import

After creating an oWFN for each subject, the manipulated PNML files can be uploaded in the management UI to generate a single OWL file representing the process model via the *standard-pass-ont* in version 0.7.5. As illustrated in Figure 6.13, the process

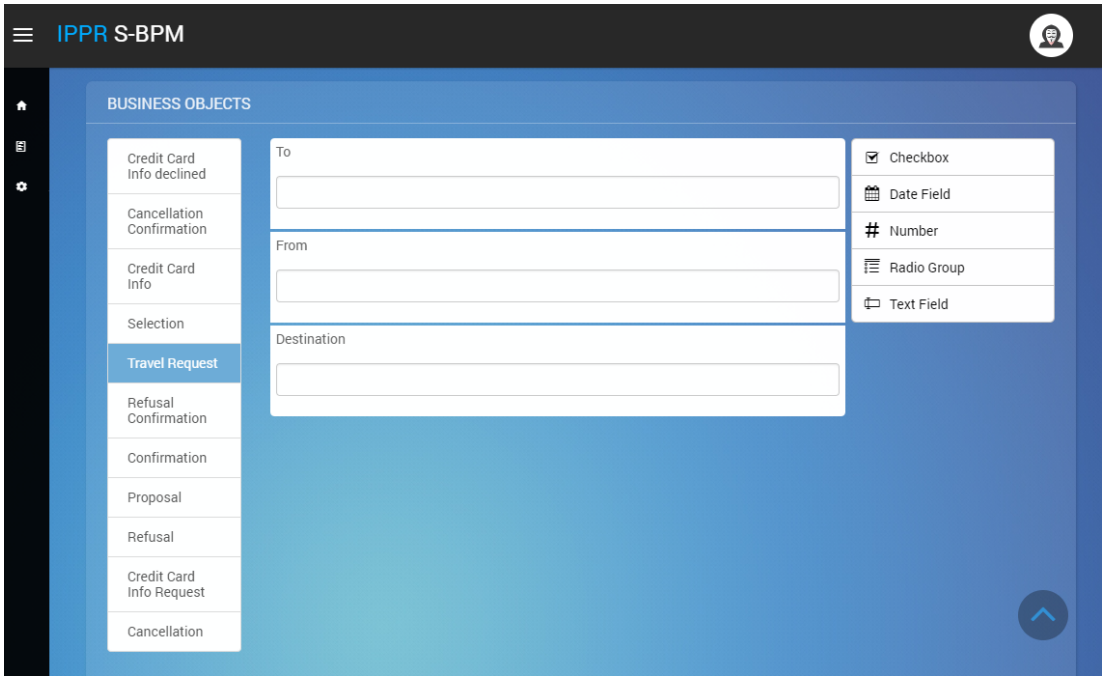

Figure 6.5.: Import of the travel request process: specifying business object fields

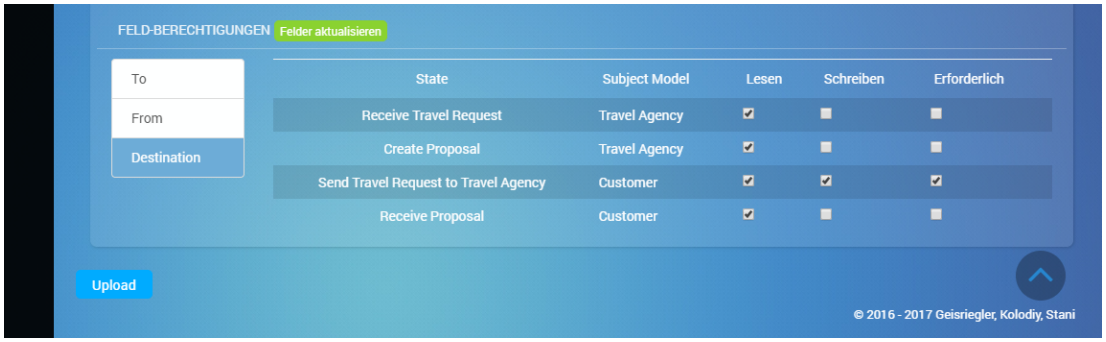

Figure 6.6.: Import of the travel request process: specifying business object field access rights

names and subjects names have to be specified, since this information is not available in the manipulated Petri net files, which have to be uploaded. The uploaded oWFNs will be transformed into one OWL file, as described in Section 5.3.4, which is available on the CD enclosed to this thesis (*Travel Request-generated.owl*).

The resulting OWL ontology can now be uploaded via the management UI, as described in Section 6.1. Afterwards, a new S-BPM process model is created, which can be executed in the platform. Since this demonstration is based on a log containing exactly those events, which are needed to reconstruct the original process model, the generated model is identical to the original model.

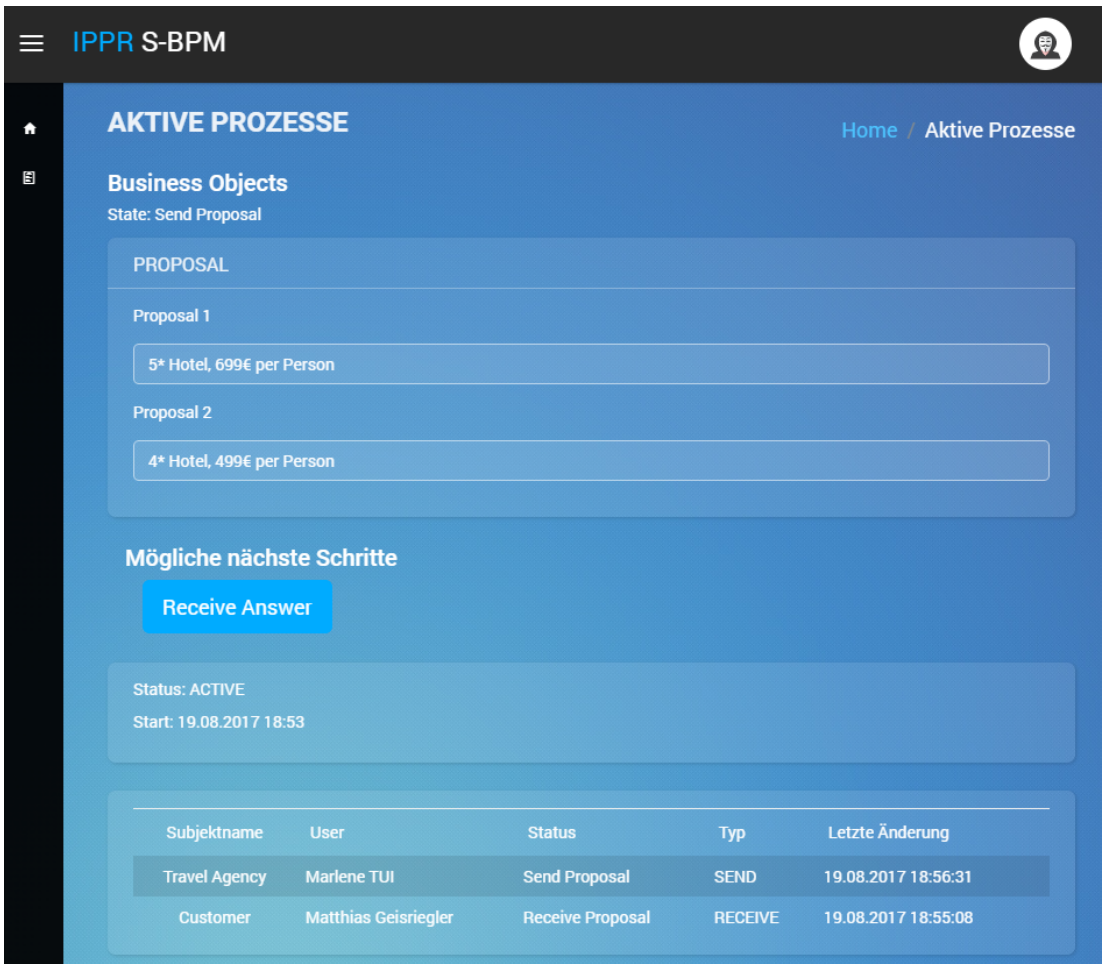

Figure 6.7.: Execution of the travel request process: specifying and sending two proposals

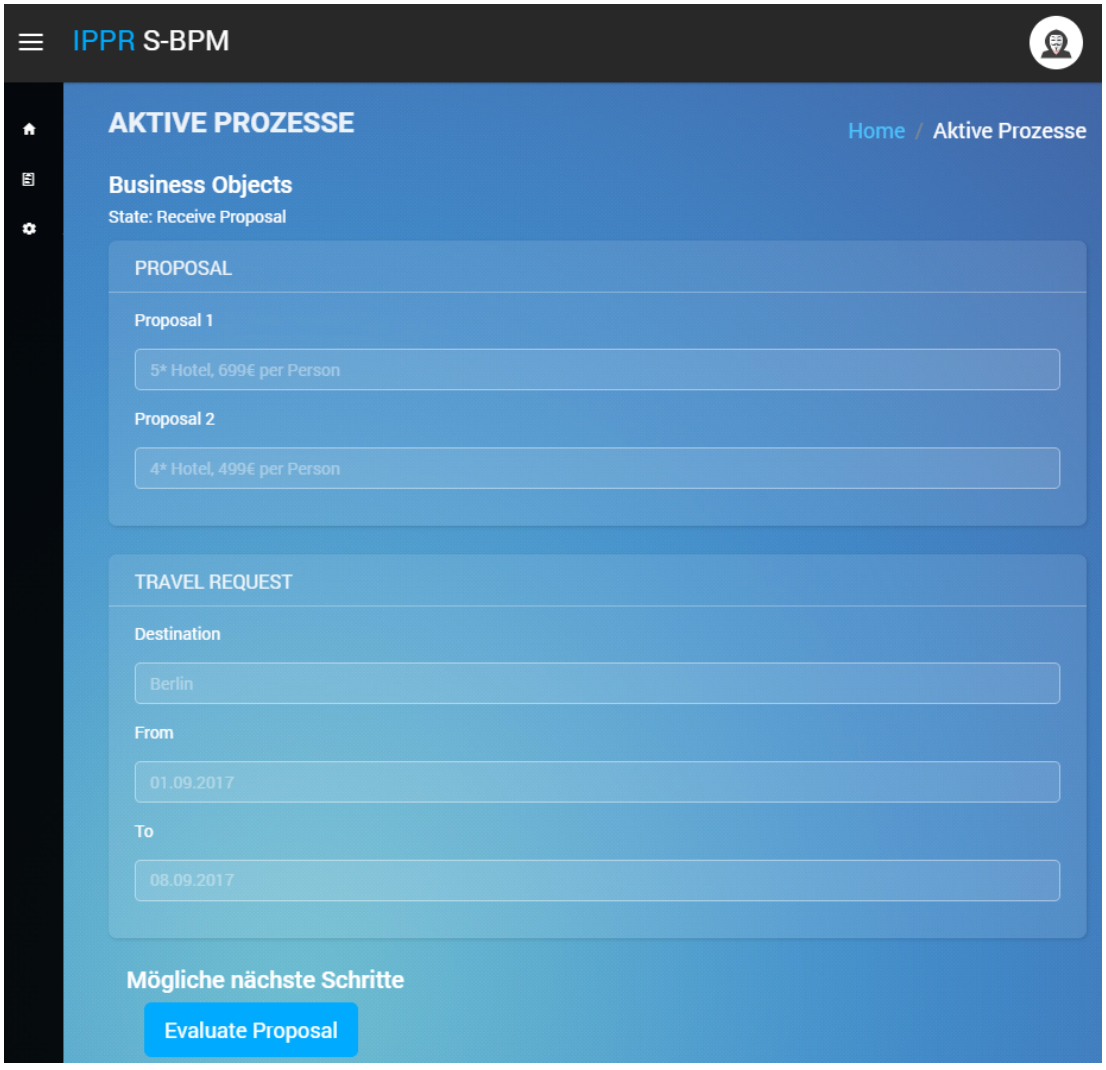

Figure 6.8.: Execution of the travel request process: receiving two proposals

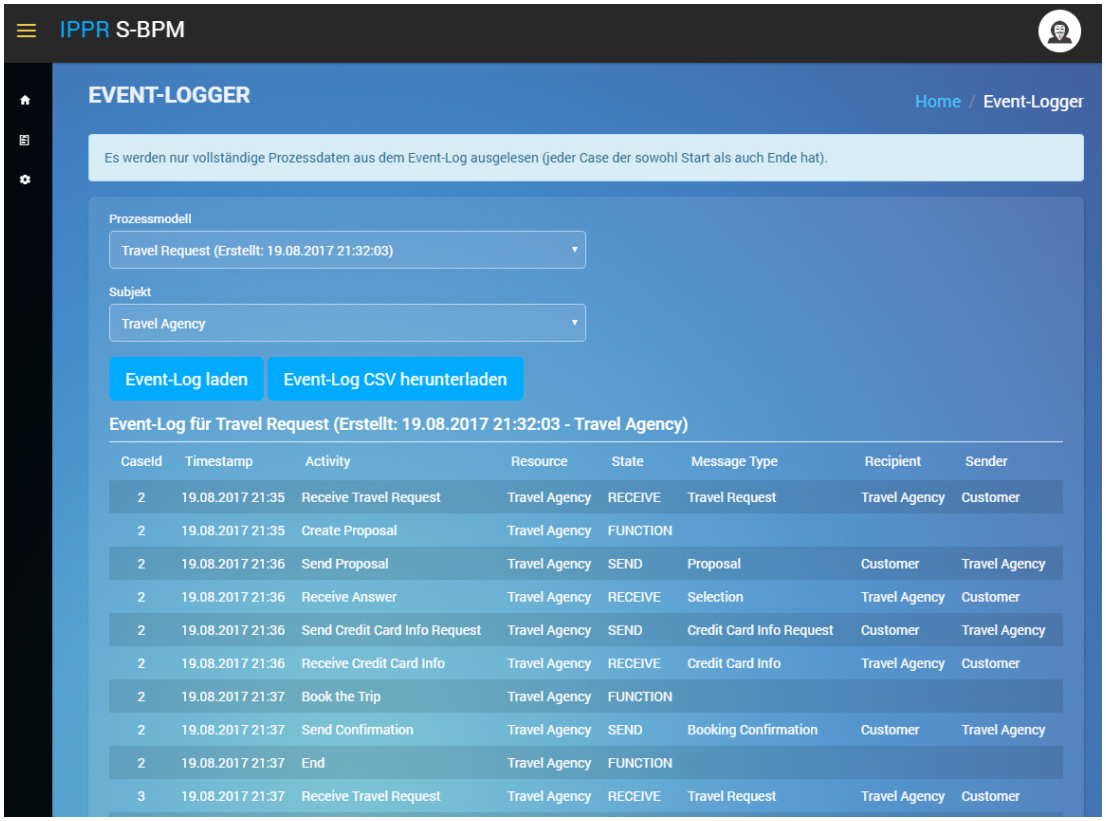

Figure 6.9.: Event log of the travel agency of the travel request process

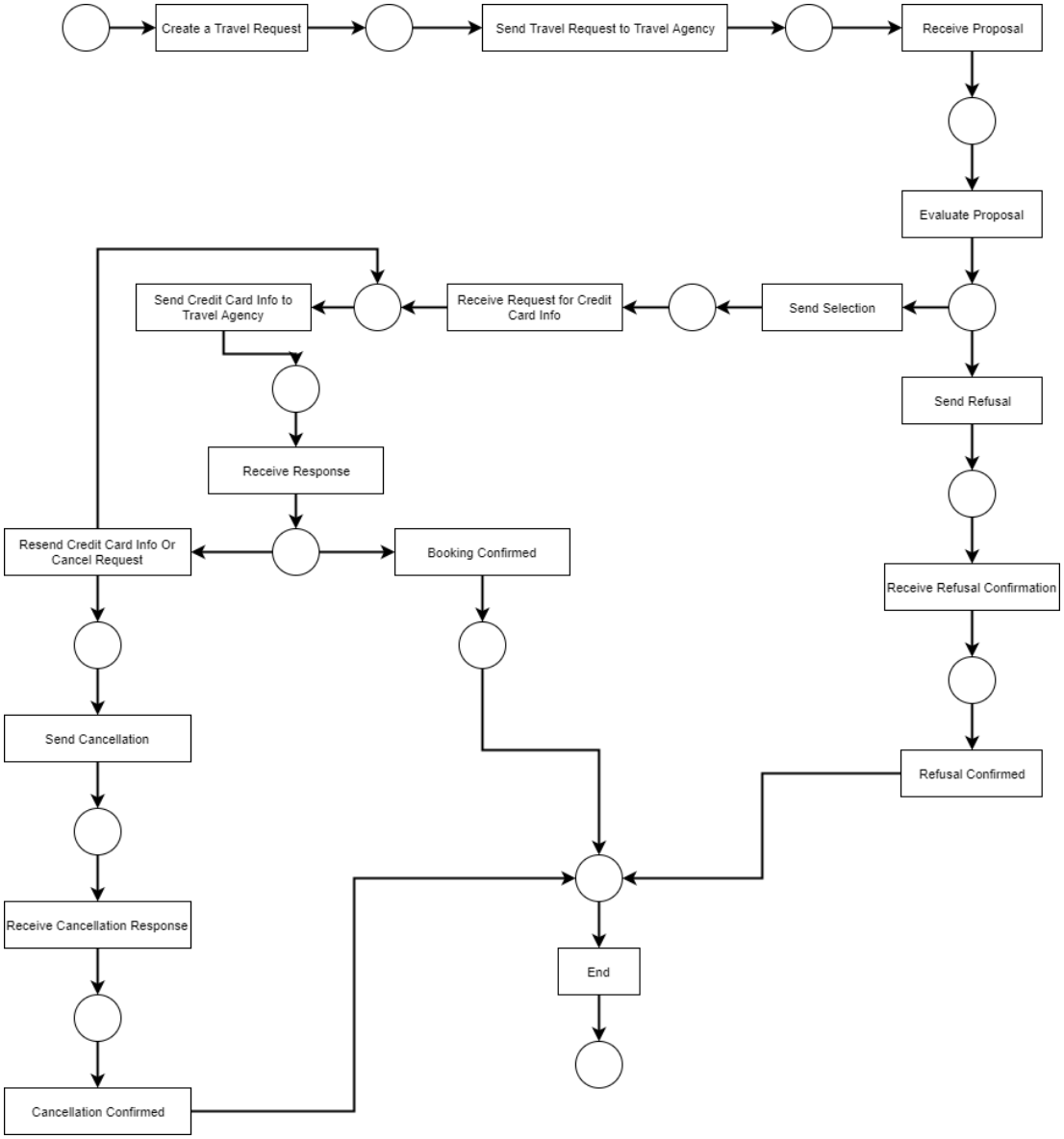

Figure 6.10.: Petri net of the travel request process of the customer

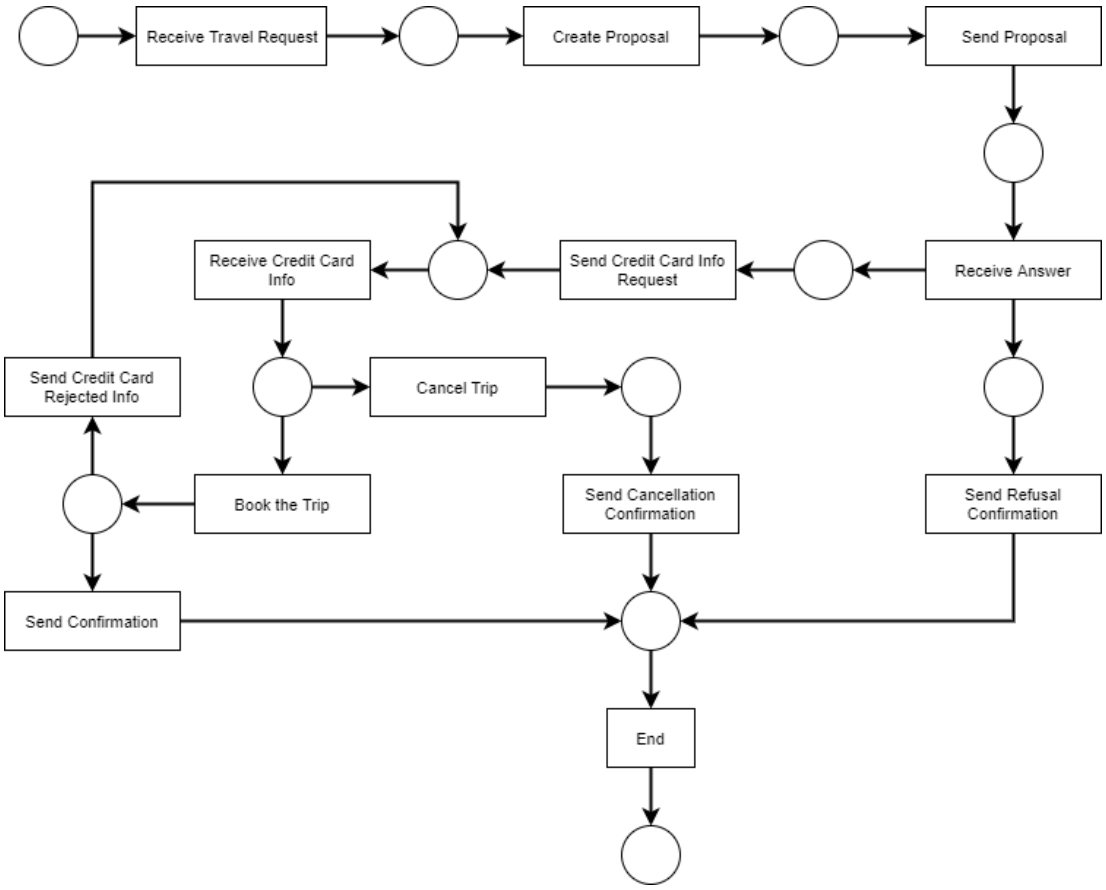

Figure 6.11.: Petri net of the travel request process of the travel agency

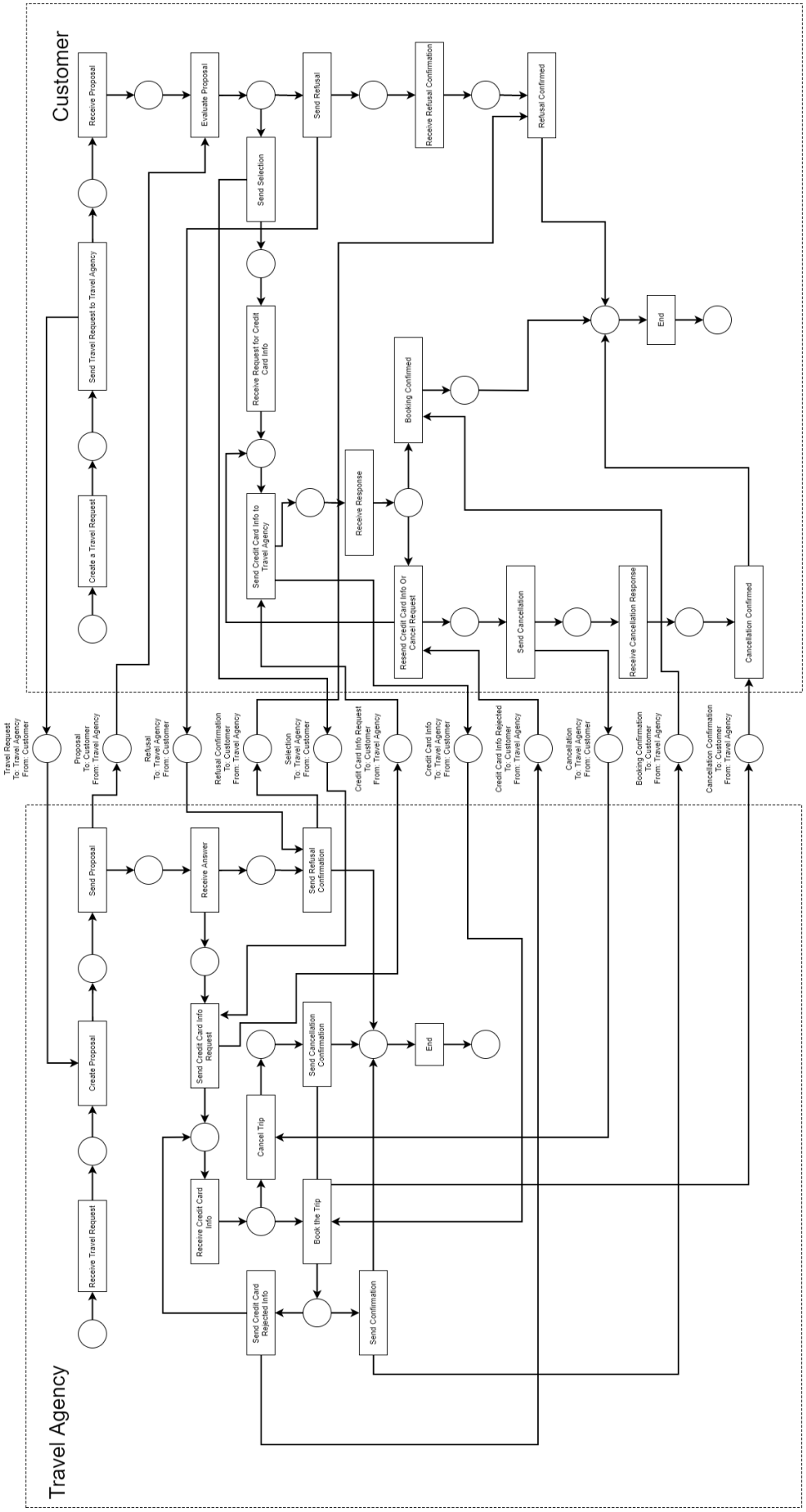

Figure 6.12.: Composed oWFN of the travel request process

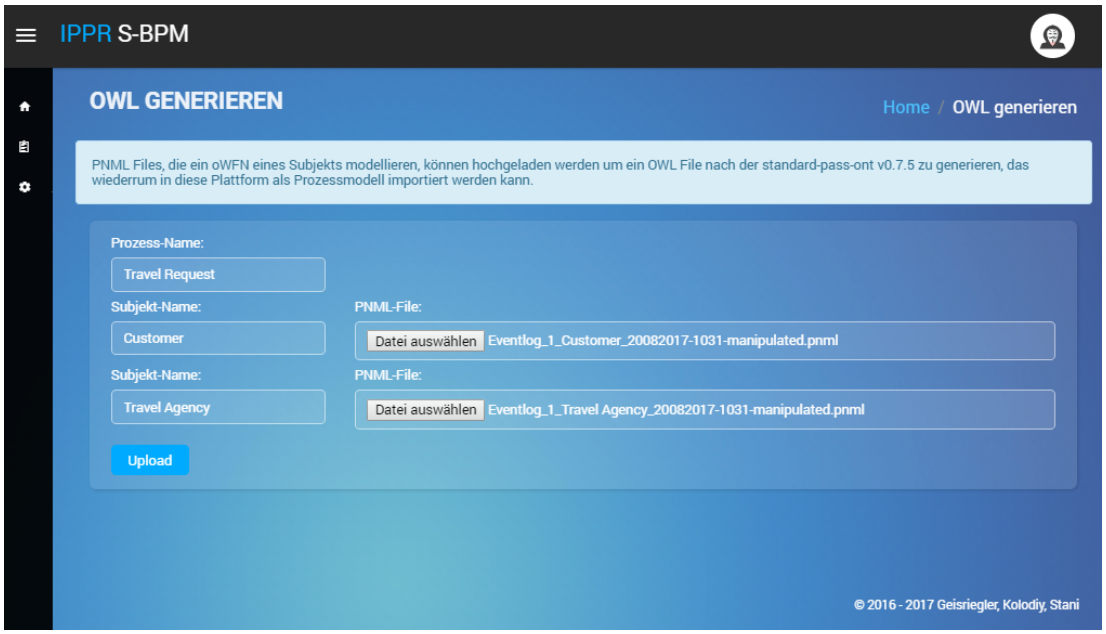

Figure 6.13.: Dialog for Petri net to OWL transformation

7. Discussion

This thesis showed, that Process Mining supports the analysis of performance, the diagnosis and the redesign of processes in the BPM life-cycle. Based on actual process data described in the event log, process discovery, conformance checking and process enhancement can be accomplished. Therefore, event logs, which may combine data from different sources, are used as output, input and general source of information in terms of Process Mining.

Event logs contain data that relates to a single process. Every process event-log is composed of cases, each consisting of a number of events. Typically, each event consists of a unique identifier, a timestamp, a resource and an activity. Since different approaches and goals of Process Mining may required different data, event logs can specify their own attributes. The target of this thesis is to detect Process Mining solutions for S-BPM. Therefore, a suitable event log format was presented. Since S-BPM processes rely heavily on the exchange of messages, it was pointed out, that an event log must contain information about the communication. For this purpose, an event log structure was developed, consisting of:

1. a case identifier, which identifies the corresponding process instance.
2. a timestamp.
3. the activity name.
4. the resource name, which identifies the role of the executing subject.
5. the state type, either function, send or receive.
6. the message type, which describes the name of the message exchanged.
7. the recipient, which identifies the role of the receiving subject.
8. the sender, which identifies the role of the sending subject.

In contrast to the other attributes, the message type, the recipient and the sender are optional, as this information is only available for activities of type send or receive.

This thesis focussed on process discovery to identify representative behaviour provided in the event log. Different Process Mining algorithms generate process models in different modelling notations. The α algorithm, which is a simple algorithm for discovery, maps an event log onto a Petri net that illustrates the orchestration of a process. However, S-BPM models do not only visualise the orchestration of each subject, but also the process choreography between the actors. Workflow Modules or Open Workflow Nets (oWF nets) specify an extension to Petri nets by introducing interfaces for communication. This provides a possibility to use Petri nets also for modelling process choreographies. As a result, the α algorithm in combination with oWF-nets is used for S-BPM process discovery.

The first attempt in ProM 6.6. showed, that by providing a simple event log, without information about the message exchange, one long control flow, visualised in a Petri net, was discovered (see Figure 4.4). Despite the fact, that this Petri net shows a valid

flow of a process, the outcome is not applicable for the S-BPM method. In a second try, by rearranging the event log, two possible control flows were discovered and visualised in a Petri net (see Figure 4.5). This result showed a process that offers a path for each subject. However, this Petri net indicates that only the path of one subject could be taken. Moreover, the discovered process model does not reveal anything about the choreography between the actors, which is vital for S-BPM processes. In the following attempt, the event log was split into multiple logs, each containing just the information for one subject. Based on the subject specific logs and by using the α algorithm in ProM, separate orchestrations could be retrieved (see Figures 4.6 and 4.7). These orchestrations are modelled in Petri nets that are also a WF nets. Nevertheless, the WF nets do not include information about the communication between the subjects. Afterwards, it was shown that the Petri nets, generated by ProM and the α algorithm, can easily be equipped with additional places, which form the communication interface.

Based on the findings concerning the splitting of the event logs and the PNML manipulation, an algorithm was developed to transform discovered Petri nets into oWF nets. This algorithm takes the event log of a subject and adds communication places to the PNML file. These communication places are connected with send or receive states and describe the message to be received or sent, the sender and the recipient. As ProM 5.2. offers a Petri net to oWFN conversion module, the results were directly illustrated in the Figures 4.16 and 4.17. As a result, a method was found to discover Petri nets based on an event log of an S-BPM process, which could be transformed into a composed oWFN (see Figure 4.19). This composed oWFN already reminds of an S-BPM process model, showing the behaviours of each subject in a merged view. Moreover, the oWFN already shows the choreography between two or more actors of a process. However, the illustration of the choreography gets very confusing if there is a high number of messages exchanged in the process. By contrast, an S-BPM model provides a clearer visualisation for process choreographies.

After finding a solution for the generation of oWFNs for the subjects, a method had to be developed, which is capable of transforming oWFNS into an executable S-BPM model. The S-BPM Modelling and Execution Platform uses the *standard-pass-ont*, an ontology for S-BPM processes, which is designed for the exchange of process models between different process modelling and execution tools. For that reason, a feature was implemented that transforms oWFNs to an S-BPM model, specified in an OWL file. This feature maps the places, transitions and arcs of the oWFN and maps them to the corresponding ontology structures. The communication places provide the information about the message exchange, which is needed to reconstruct the choreography for S-BPM.

7.1. Implementation

A goal of this thesis was to extend the capabilities of the S-BPM Modelling and Execution Platform in terms of supporting Process Mining methods. The existing architecture was supplemented with a service, dedicated to log events and to transform Petri nets to oWFNs and oWFNs to OWL files. These implementations rely on the theoretical findings and proposals presented in this thesis and are designed to be used in addition to ProM 6.6. and ProM 5.4.

Since event logs provide the basis for all Process Mining efforts, the existing execution mechanisms, based on Akka, had to be equipped with a logging functionality, suitable for the event log structure presented in the theoretical part. Moreover, an interface was created to export the event logs in CSV format, which is supported by ProM 6.6.

To transform Petri nets that are serialised in PNML files, the PNML structure had to be acquired. Moreover, a method was created to combine the capabilities of PNML and the data given in the original log, in order to create an oWFN describing the message exchange interfaces for a subject.

Finally, a procedure for the transformation from the manipulated PNML files to an importable OWL file had to be developed. For this purpose, a mapping from PNML structures to the corresponding structures of the *standard-pass-ont* in version 0.7.5 was implemented.

7.2. Scope

The main goal of this thesis was to find, illustrate and implement an algorithm that allows for the discovery of process choreographies based on event log data. The solution presented is tightly coupled with the principles of S-BPM, since the implementation is part of the S-BPM Execution Platform. The algorithm and methods presented were developed based on the vacation process with two subjects. The vacation request is a clear and simple process model, which is sufficient for demonstration purposes. Moreover, the travel request process was used to present the life-cycle of a business process in the platform starting with the creation, via the execution through to the discovery and re-import of the model. The travel request was used to present that the recommendations and implementation of this thesis work out with a more complex process model as well. The methods presented in this thesis prove to be applicable for process choreographies modelled via the S-BPM notation, representing the orchestrations of two subjects.

The algorithm developed is designed to work with the event log format presented in this thesis. Therefore, event logs from external sources may not be compatible without any additional data transformation efforts. Moreover, the event log structure has a strong focus on the subjects and the messages exchanged throughout the execution of a process. Sample event logs available on the internet (e.g. the event logs used in [17], which are available on the process mining website[1] or sample event logs of the four technical universities of the Netherlands[2]) may be tested with the implemented features. However, they do not provide enough information for reconstructing process choreographies, as they are dedicated to illustrate process orchestrations or as they do not reveal any or insufficient information about the message exchange in the process.

The Process Mining methods used for retrieving S-BPM models were based on the α algorithm, since it is a simple and comprehensible algorithm for demonstration purposes. However, any algorithm may be used that generates a Petri net, as the communication dimension is added after applying the discovery methods. Moreover, this thesis aimed to show, that the original process can be reconstructed by using Process

[1] http://www.processmining.org/event_logs_and_models_used_in_book
[2] http://data.4tu.nl/repository/collection:event_logs

Mining techniques and the algorithm developed. For that reason, the algorithm may be adjusted to be applicable for other purposes. Nevertheless, the methods presented in this thesis provide a basis for discovering and constructing process choreographies by transforming Petri nets into oWFNs and oWFNs into OWL files.

Given these facts, common Process Mining method as well as sample event logs available on the Internet seem to focus on process orchestrations. This leads to the assumption, that the majority of the process models designed and executed in companies also focus rather on choreographies than on orchestrations. Moreover, it may also be possible that event logs, generated by process engines or other facilities that support the process execution, most commonly provide information about a process orchestration or they do not emphasize process choreographies sufficiently.

8. Conclusion

There are several processes modelling notations available, however, they differ strongly in their support of process choreographies. Petri nets, for example, are not designed to model the interaction between multiple actors of a process. The subject oriented modelling approach S-BPM, by contrast, provides a methodology to model process orchestrations as well as process choreographies. As common Process Mining methods emphasise orchestrations, a solution had to be found to support generation of choreographies. Moreover, several Process Mining techniques generate Petri nets, which lack the interaction and communication perspective of a process. For this purpose, Open Workflow Nets and Workflow Modules were presented. They extend the Petri net notation by offering communication places, which form input and output interfaces for the interaction between multiple orchestrations. As a result, oWFNs or Workflow Modules provide a possible solution for combining Process Mining methods and process choreographies.

In this thesis, the principles of Process Mining were discussed, tested and illustrated based on the open source Process Mining tool *ProM*. A typical event log structure was extended for the purpose of developing a structure that is suitable for S-BPM. With the aid of the α algorithm, a solution was presented to retrieve separate orchestrations based on filtering the original event log by the individual subjects. Subsequently, an algorithm was developed and illustrated, which provides a possibility to transform discovered Petri nets to oWFNs. For this purpose, the PNML files generated by ProM are parsed and equipped with communication places, by comparing the result of the discovery algorithm with the event log of the corresponding subjects.

The S-BPM Modelling and Execution Platform is a prototypical web application, providing a toolset for S-BPM processes. As illustrated, the platform was extended by a new service dedicated to support all procedures concerning the event log and the Process Mining efforts. This service enables the logging of all events during the execution of a process, including all information necessary to generate an event log according to the structure proposed in this thesis. Moreover, the presented algorithm, which was developed for transforming Petri nets to oWFNs based on the event log, was implemented. The S-BPM ontology, a semantic notation in OWL for describing S-BPM process models, was briefly explained and a feature for converting the generated oWFNs to an OWL file was implemented. For this purpose, the existing import functionality of the platform was extended to support the newest version (0.7.5) of the *standard-pass-ont*. Finally, all implemented features were demonstrated and an S-BPM process model was successfully reconstructed.

As illustrated in this thesis, existing Process Mining methods had to be extended to support the discovery of process choreographies. It was shown that an algorithm, which discovers Petri nets from a given log, is not sufficient for reconstructing choreographies. However, due to the fact that oWFNs or Workflow Modules, which extend the expressive power of Petri nets, provide suitable mechanisms for describing process

choreographies. As a result, not only the α algorithm but any discovery algorithm generating a Petri net may be used in combination with the algorithm proposed in this thesis, to retrieve process choreographies.

Moreover, this thesis showed that the implemented features enable the S-BPM Platform to *Play-Out*, *Play-In* and *Replay* an event log by using Process Mining techniques. As a consequence, the platform covers all phases of the BPM life-cycle (see Figure 3.1):

1. *Design*: a process can be modelled using the S-BPM notation.
2. *Configuration / Implementation*: a process can be imported into and executed via the execution platform.
3. *Enactment / Monitoring*: process events are stored in the event log.
4. *Diagnosis / Requirements*: the data of the event log is used as input for Process Mining methods to reconstruct and discover process models.
5. *Redesign*: after applying Process Mining techniques, the results can be transformed into OWL files and the discovered and / or revised process model can be imported and executed.

All things considered, a solution was developed that combines Process Mining methods and process choreography modelling. In conclusion, this thesis showed that it is possible to generate process choreographies by means of Process Mining.

A. Appendix

A.1. Source Code of the EPNML Manipulation Service

```
1  package at.fhjoanneum.ippr.eventlogger.services;
2
3  import at.fhjoanneum.ippr.eventlogger.helper.LogEntry
       ;
4  import at.fhjoanneum.ippr.eventlogger.helper.LogKey;
5  import at.fhjoanneum.ippr.eventlogger.helper.
       XMLParserCommons;
6  import at.fhjoanneum.ippr.persistence.objects.model.
       enums.StateFunctionType;
7  import org.apache.commons.lang.StringUtils;
8  import org.slf4j.Logger;
9  import org.slf4j.LoggerFactory;
10 import org.springframework.stereotype.Service;
11 import org.springframework.transaction.annotation.
       Isolation;
12 import org.springframework.transaction.annotation.
       Transactional;
13 import org.supercsv.cellprocessor.Optional;
14 import org.supercsv.cellprocessor.ParseLong;
15 import org.supercsv.cellprocessor.constraint.NotNull;
16 import org.supercsv.cellprocessor.ift.CellProcessor;
17 import org.supercsv.io.CsvBeanReader;
18 import org.supercsv.io.ICsvBeanReader;
19 import org.supercsv.prefs.CsvPreference;
20 import org.w3c.dom.Document;
21 import org.w3c.dom.Element;
22 import org.w3c.dom.Node;
23 import org.w3c.dom.NodeList;
24 import org.xml.sax.InputSource;
25
26 import javax.xml.parsers.DocumentBuilderFactory;
27 import javax.xml.transform.Transformer;
28 import javax.xml.transform.TransformerFactory;
29 import javax.xml.transform.dom.DOMSource;
30 import javax.xml.transform.stream.StreamResult;
31 import java.io.StringReader;
32 import java.io.StringWriter;
33 import java.util.*;
```

```
34
35
36 @Transactional(isolation = Isolation.READ_COMMITTED)
37 @Service
38 public class ManipulatePNMLServiceImpl implements
      ManipulatePNMLService {
39
40   private static final Logger LOG = LoggerFactory.
       getLogger(ManipulatePNMLServiceImpl.class);
41
42   @Override
43   public StreamResult manipulatePNML(final String
       pnmlContent, final String csvLog)
44       throws Exception {
45     StreamResult result = new StreamResult();
46
47     try {
48       final LinkedHashSet<LogEntry> logEntries =
           parseCSV(csvLog);
49       final LinkedHashMap<LogKey, LogEntry>
           logQuintuplets = getQuintuplets(logEntries);
50       final DocumentBuilderFactory
           documentBuilderFactory =
           DocumentBuilderFactory.newInstance();
51       final InputSource input = new InputSource(new
           StringReader(pnmlContent));
52       final Document document =
           documentBuilderFactory.newDocumentBuilder().
           parse(input);
53       final Element root = document.
           getDocumentElement();
54       final NodeList nets = root.getElementsByTagName
           ("net");
55
56       for (int temp = 0; temp < nets.getLength();
           temp++) {
57         final Node netNode = nets.item(temp);
58         if (netNode.getNodeType() == Node.
             ELEMENT_NODE) {
59           final Element net = (Element) netNode;
60           final HashMap<String, String> transitions =
                XMLParserCommons.getTransitionNameIdMap(
               net);
61
62           int numOfCustomPlaces = 1;
63           int numOfCustomArcs = 1;
```

```
64            for(Map.Entry<LogKey, LogEntry> entry :
                 logQuintuplets.entrySet()){
65              final LogEntry eventLogEntry = entry.
                   getValue();
66
67              final String state = eventLogEntry.
                   getState();
68              final String activity = eventLogEntry.
                   getActivity();
69              final String messageType = eventLogEntry.
                   getMessageType();
70              final String recipient = eventLogEntry.
                   getRecipient();
71              final String sender = eventLogEntry.
                   getSender();
72
73              if (state.equals(StateFunctionType.SEND.
                   name()) && transitions.containsKey(
                   activity)) {
74                final String placeId = addPlace(
                     document, net, numOfCustomPlaces++,
                     messageType, recipient, sender, "send
                     ", "");
75                addArc(document, net, numOfCustomArcs
                     ++, transitions.get(activity),
                     placeId, messageType);
76              } else if (state.equals(StateFunctionType
                   .RECEIVE.name())) {
77                if (eventLogEntry.getNextLogEntryKey()
                     == null) {
78                  throw (new Exception("Last Log-Entry
                       may not be of type Receive!"));
79                }
80                final LogEntry nextEventLogEntry =
                     logQuintuplets.get(eventLogEntry.
                     getNextLogEntryKey());
81
82                final String nextActivity =
                     nextEventLogEntry.getActivity();
83
84                if(transitions.containsKey(nextActivity
                     )) {
85                  final String placeId = addPlace(
                       document, net, numOfCustomPlaces++,
                        messageType, recipient, sender, "
                       receive", transitions.get(activity)
                       );
```

```
86                  addArc(document, net, numOfCustomArcs
                       ++, placeId, transitions.get(
                       nextActivity), messageType);
87                }
88              }
89            }
90          }
91
92          final DOMSource source = new DOMSource(
               document);
93
94          result = new StreamResult(new StringWriter())
               ;
95          final TransformerFactory transformerFactory =
                TransformerFactory.newInstance();
96          final Transformer transformer =
               transformerFactory.newTransformer();
97          transformer.transform(source, result);
98        }
99      } catch (final Exception e) {
100       LOG.error(e.getMessage());
101       LOG.error("Exception while manipulating PNML");
102       throw (e);
103     }
104
105     return result;
106   }
107
108   private LinkedHashSet<LogEntry> parseCSV(final
         String csvLog) throws Exception {
109
110     final LinkedHashSet<LogEntry> result = new
           LinkedHashSet<>();
111     ICsvBeanReader beanReader = null;
112     try {
113       beanReader = new CsvBeanReader(new StringReader
             (csvLog), CsvPreference.
             EXCEL_NORTH_EUROPE_PREFERENCE);
114
115       // the header elements are used to map the
             values to the bean (names must match)
116       final String[] header = beanReader.getHeader(
             true);
117       final String[] uncapitalizedHeader = new String
             [header.length];
118       for (int i = 0; i < header.length; i++) {
```

```
119         uncapitalizedHeader[i] = StringUtils.
               uncapitalize(header[i]);
120       }
121
122       final CellProcessor[] processors =
             getProcessors();
123
124       LogEntry eventLogEntry;
125       while ((eventLogEntry = beanReader.read(
             LogEntry.class, uncapitalizedHeader,
             processors)) != null) {
126         result.add(eventLogEntry);
127       }
128     } finally {
129       if (beanReader != null) {
130         beanReader.close();
131       }
132
133       return result;
134     }
135   }
136
137   private LinkedHashMap<LogKey, LogEntry>
        getQuintuplets(final HashSet<LogEntry> logEntries
        ) throws Exception {
138
139     final LinkedHashMap<LogKey, LogEntry> result =
           new LinkedHashMap<>();
140
141     LogKey lastKey = null;
142     for(LogEntry logEntry : logEntries){
143       final LogKey key = new LogKey(logEntry.
             getActivity(), logEntry.getState(), logEntry.
             getMessageType(), logEntry.getRecipient(),
             logEntry.getSender());
144       if (!result.containsKey(key)) {
145         result.put(key, logEntry);
146       }
147
148       if (lastKey != null){
149         result.get(lastKey).setNextLogEntryKey(key);
150       }
151
152       lastKey = key;
153     }
154
155     return result;
```

```
156   }
157
158   private static CellProcessor[] getProcessors() {
159     final CellProcessor[] processors = new
          CellProcessor[] {new NotNull(new ParseLong()),
          // EventId
160         new NotNull(new ParseLong()), // CaseId
161         new NotNull(), // Timestamp
162         new NotNull(), // Activity
163         new NotNull(), // Resource
164         new NotNull(), // State
165         new Optional(), // MessageType
166         new Optional(), // Recipient
167         new Optional() // Sender
168     };
169
170     return processors;
171   }
172
173   private String addArc(final Document document,
        final Element net, final int id,
174       final String sourceId, final String targetId,
            final String name) {
175     // Custom Arc
176     final Element newArc = document.createElement("
          arc");
177     net.appendChild(newArc);
178
179     // Id
180     final String arcId = "ca" + id;
181     newArc.setAttribute("id", arcId);
182
183     // Source Id
184     newArc.setAttribute("source", sourceId);
185
186     // Target Id
187     newArc.setAttribute("target", targetId);
188
189     // Name
190     final Element nameElement = document.
          createElement("name");
191     newArc.appendChild(nameElement);
192
193     // Name Text
194     final Element nameText = document.createElement("
          text");
```

```
195     nameText.appendChild(document.createTextNode(name
            ));
196     nameElement.appendChild(nameText);
197
198     // Arctype
199     final Element arcTypeElement = document.
            createElement("arcType");
200     newArc.appendChild(arcTypeElement);
201
202     // Arctype Text
203     final Element arcTypeText = document.
            createElement("text");
204     arcTypeText.appendChild(document.createTextNode("
            normal"));
205     arcTypeElement.appendChild(arcTypeText);
206
207     return arcId;
208   }
209
210   private String addPlace(final Document document,
          final Element net, final int id,
211         final String name, final String recipient,
              final String sender, final String type, final
                String actualTargetId) {
212     // Custom Place
213     final Element newPlace = document.createElement("
            place");
214     net.appendChild(newPlace);
215
216     // Id
217     final String placeId = "cp" + id;
218     newPlace.setAttribute("id", placeId);
219
220     // Name
221     final Element nameElement = document.
            createElement("name");
222     newPlace.appendChild(nameElement);
223
224     // Name Text
225     String nameText = name;
226     if(name != null){
227       nameText = name +  " To: " + recipient + " From
              : " + sender;
228     }
229     final Element text = document.createElement("text
            ");
```

```
230     text.appendChild(document.createTextNode(nameText
            ));
231     nameElement.appendChild(text);
232
233     //Toolspecific infos used for generating OWL
            files afterwards
234     final Element toolspecificElement = document.
            createElement("toolspecific");
235     toolspecificElement.setAttribute("tool", "SBPM");
236     toolspecificElement.setAttribute("version", "1.0"
            );
237     toolspecificElement.setAttribute("message", name)
            ;
238     toolspecificElement.setAttribute("type", type);
239     toolspecificElement.setAttribute("recipient",
            recipient);
240     toolspecificElement.setAttribute("sender", sender
            );
241
242     if(!actualTargetId.isEmpty()){
243       toolspecificElement.setAttribute("
              actualTargetId", actualTargetId);
244     }
245
246     newPlace.appendChild(toolspecificElement);
247
248     // Graphics
249     final Element graphics = document.createElement("
            graphics");
250     newPlace.appendChild(graphics);
251
252     // Graphics Position
253     final Element position = document.createElement("
            position");
254     position.setAttribute("x", "0");
255     position.setAttribute("y", "0");
256     graphics.appendChild(position);
257
258     // Graphics Dimension
259     final Element dimension = document.createElement(
            "dimension");
260     dimension.setAttribute("x", "12.5");
261     dimension.setAttribute("y", "12.5");
262     graphics.appendChild(position);
263
264     return placeId;
265   }
```

```
266 }
```

Listing A.1: Source code of the EPNML manipulation service

A.2. Source Code of the Petri net to OWL Transformation Service

```
1  package at.fhjoanneum.ippr.eventlogger.services;
2
3  import at.fhjoanneum.ippr.eventlogger.helper.Arc;
4  import at.fhjoanneum.ippr.eventlogger.helper.Message;
5  import at.fhjoanneum.ippr.eventlogger.helper.
       XMLParserCommons;
6  import org.joda.time.DateTime;
7  import org.slf4j.Logger;
8  import org.slf4j.LoggerFactory;
9  import org.springframework.stereotype.Service;
10 import org.springframework.transaction.annotation.
       Isolation;
11 import org.springframework.transaction.annotation.
       Transactional;
12 import org.w3c.dom.*;
13 import org.xml.sax.InputSource;
14 import org.xml.sax.SAXException;
15
16 import javax.xml.parsers.DocumentBuilder;
17 import javax.xml.parsers.DocumentBuilderFactory;
18 import javax.xml.transform.Transformer;
19 import javax.xml.transform.TransformerFactory;
20 import javax.xml.transform.dom.DOMSource;
21 import javax.xml.transform.stream.StreamResult;
22 import java.io.*;
23 import java.util.*;
24 import java.util.function.Predicate;
25 import java.util.stream.Collectors;
26
27
28 @Transactional(isolation = Isolation.READ_COMMITTED)
29 @Service
30 public class GenerateOWLServiceImpl implements
       GenerateOWLService {
31
32   private static final Logger LOG = LoggerFactory.
       getLogger(GenerateOWLServiceImpl.class);
33   private static final String standardPassOntUri = "&
       standard-pass-ont;";
```

```
34  private String ontologyUri;
35
36  @Override
37  public StreamResult generateOWL(String
       processModelName, Map<String, String> petriNets)
       throws Exception {
38    final DocumentBuilderFactory
         resultDocumentBuilderFactory =
         DocumentBuilderFactory.newInstance();
39    final DocumentBuilder resultDocumentBuilder =
         resultDocumentBuilderFactory.newDocumentBuilder
         ();
40    final Document resultDocument =
         resultDocumentBuilder.newDocument();
41    final String date = DateTime.now().toString("
         ddMMyyyy-HHmm");
42    ontologyUri = "http://fh-joanneum.at/aim/s-bpm/
         processmodels/"+date+"/";
43    HashMap<String, Node> subjectNameToSubjectNodeMap
        = new HashMap<>();
44    HashMap<String, Node>
         subjectNameToMessageExchangeListNodeMap = new
         HashMap<>();
45    HashMap<String, Node>
         subjectNameToBehaviorNodeMap = new HashMap<>();
46    HashMap<String, Node> messageNameToMessageNodeMap
        = new HashMap<>();
47    HashMap<String, Node>
         messageNameToMessageExchangeNodeMap = new
         HashMap<>();
48
49    Node rdfNode = getOWLSkeleton(resultDocument,
         resultDocumentBuilder);
50    Node processModelNode = createNamedIndividual(
         resultDocument, processModelName, "
         PASSProcessModel", ontologyUri,
         processModelName);
51    rdfNode.appendChild(processModelNode);
52
53    Node sidNode = createNamedIndividual(
         resultDocument, "SID_1", "ModelLayer", "SID_1",
          "SID_1");
54    rdfNode.appendChild(sidNode);
55    addPriorityNumberElement(resultDocument, sidNode)
         ;
56
```

```
57      addContainsElement(resultDocument,
           processModelNode, sidNode);
58
59      //First create nodes based on name that will be
           referenced when looping over the PNML content
60      petriNets.keySet().forEach(name -> {
61        String subjectIdentifier = "
             SID_1_FullySpecifiedSingleSubject_"+name;
62        Node subjectNode = createNamedIndividual(
             resultDocument, subjectIdentifier, "
             FullySpecifiedSingleSubject",
             subjectIdentifier, name);
63        rdfNode.appendChild(subjectNode);
64        addContainsElement(resultDocument, Arrays.
             asList(processModelNode, sidNode),
             subjectNode);
65        addMaximumSubjectInstanceRestrictionElement(
             resultDocument, subjectNode);
66
67        subjectNameToSubjectNodeMap.put(name,
             subjectNode);
68
69        Node behaviorNode = createNamedIndividual(
             resultDocument, "SBD_"+subjectIdentifier, "
             SubjectBehavior", "SBD_"+subjectIdentifier, "
             SBD: "+name);
70        rdfNode.appendChild(behaviorNode);
71        addContainsElement(resultDocument, Arrays.
             asList(processModelNode, sidNode),
             behaviorNode);
72        addResourceElement(resultDocument, subjectNode,
              behaviorNode, "containsBehavior");
73        addResourceElement(resultDocument, subjectNode,
              behaviorNode, "containsBaseBehavior");
74        subjectNameToBehaviorNodeMap.put(name,
             behaviorNode);
75
76  String messageExchangeListIdentifier = "
      MessageExchangeList_on_
      [1]SID_1_StandardMessageConnector_"+name;
77        Node messageExchangeListNode =
             createNamedIndividual(resultDocument,
             messageExchangeListIdentifier, "
             MessageExchangeList",
             messageExchangeListIdentifier, "
```

[1]Added line break because variable name exceeds listing width

```
            SID_1_StandardMessageConnector_"+name);
78        rdfNode.appendChild(messageExchangeListNode);
79        addContainsElement(resultDocument,
            processModelNode, messageExchangeListNode);
80
81        subjectNameToMessageExchangeListNodeMap.put(
            name, messageExchangeListNode);
82      });
83
84      //Then create nodes based on content
85      int messageId = 1;
86      int stateId = 1;
87      int transitionId = 1;
88      int messageConnectorId = 1;
89
90      for (Map.Entry<String, String> entry : petriNets.
          entrySet()) {
91        String name = entry.getKey();
92        String pnmlContent = entry.getValue();
93
94        try {
95          final DocumentBuilderFactory
              documentBuilderFactory =
              DocumentBuilderFactory.newInstance();
96          final InputSource input = new InputSource(new
               StringReader(pnmlContent));
97          final Document document =
              documentBuilderFactory.newDocumentBuilder()
              .parse(input);
98          final Element root = document.
              getDocumentElement();
99          final NodeList nets = root.
              getElementsByTagName("net");
100
101         for (int temp = 0; temp < nets.getLength();
              temp++) {
102           final Node netNode = nets.item(temp);
103           if (netNode.getNodeType() == Node.
                ELEMENT_NODE) {
104             final Element net = (Element) netNode;
105
106             HashSet<String> allPlaces =
                  getAllPlaceIds(net);
107             HashMap<String, List<Message>>
                  placeIdToMessagesMap =
                  getPlaceIdToMessagesMap(net);
108
```

```
109             for (Map.Entry<String, List<Message>>
                    placeToMessageEntry :
                    placeIdToMessagesMap.entrySet()) {
110               List<Message> messages =
                     placeToMessageEntry.getValue();
111               for(Message message : messages) {
112                 String messageName = "Message: "+
                       message.getName() + " From: "+
                       message.getSender() + " To: "+
                       message.getRecipient();
113                 if(!messageNameToMessageNodeMap.
                       containsKey(messageName)){
114                   Node messageNode =
                         createNamedIndividual(
                         resultDocument, "message_"+
                         messageId, "MessageSpecification"
                         , "message_"+messageId, message.
                         getName());
115                   rdfNode.appendChild(messageNode);
116                   addContainsElement(resultDocument,
                         Arrays.asList(processModelNode,
                         sidNode), messageNode);
117
118                   Node payloadDescriptionNode =
                         createNamedIndividual(
                         resultDocument, "
                         payload_description_of_message_"+
                         messageId, "PayloadDescription",
                         "payload_description_of_message_"
                         +messageId, "
                         payload_description_of_message_"+
                         messageId);
119                   rdfNode.appendChild(
                         payloadDescriptionNode);
120                   addResourceElement(resultDocument,
                         messageNode,
                         payloadDescriptionNode, "
                         containsPayloadDescription");
121
122                   Node messageExchangeNode =
                         createNamedIndividual(
                         resultDocument, "
                         SID_1_StandardMessageConnector_"+
                         messageConnectorId+"_message_"+
                         messageId, "MessageExchange", "
                         SID_1_StandardMessageConnector_"+
                         messageConnectorId+"_message_"+
```

```
                        messageId, messageName);
123                   rdfNode.appendChild(
                        messageExchangeNode);
124                   addResourceElement(resultDocument,
                        messageExchangeNode,
                        subjectNameToSubjectNodeMap.get(
                        message.getSender()), "hasSender"
                        );
125                   addResourceElement(resultDocument,
                        messageExchangeNode,
                        subjectNameToSubjectNodeMap.get(
                        message.getRecipient()), "
                        hasReceiver");
126                   addResourceElement(resultDocument,
                        messageExchangeNode, messageNode,
                         "hasMessageType");
127                   addContainsElement(resultDocument,
                        Arrays.asList(processModelNode,
                        sidNode), messageExchangeNode);
128
129                   messageNameToMessageExchangeNodeMap
                        .put(messageName,
                        messageExchangeNode);
130                   messageNameToMessageNodeMap.put(
                        messageName, messageNode);
131                   messageId++;
132                   messageConnectorId++;
133                 }
134
135                 if(message.getSender().equals(name)){
136 addContainsElement(resultDocument,
    subjectNameToMessageExchangeListNodeMap.get(name),
    messageNameToMessageExchangeNodeMap.get(messageName
    ));
137                 }
138               }
139             }
140
141             final HashMap<String, Node>
                  transitionIdToStateNodeMap = new
                  HashMap<>();
142             final HashMap<String, Node>
                  transitionIdToActionNodeMap = new
                  HashMap<>();
143             final HashMap<String, String>
                  doStatesIdToNameMap = new HashMap<>();
```

```
144             final HashMap<String, String>
                   sendStatesIdToNameMap = new HashMap<>()
                   ;
145             final HashMap<String, Arc> pnmlArcIdMap =
                    getArcIdMap(net);
146             final List<Arc> pnmlArcs = new ArrayList
                   <>(pnmlArcIdMap.values());
147
148             HashMap<String, Message>
                   messagePlaceIdToMessageMap = new
                   HashMap<>();
149             for (Map.Entry<String, List<Message>>
                   placeEntry : placeIdToMessagesMap.
                   entrySet()) {
150               placeEntry.getValue().forEach(pe ->
                     messagePlaceIdToMessageMap.put(pe.
                     getMessagePlaceId(), pe));
151             }
152             HashSet<Arc>
                   directLinksBetweenTransitions =
                   getDirectLinksBetweenTransitions(
                   allPlaces, messagePlaceIdToMessageMap.
                   keySet(), pnmlArcs);
153
154             //First create all OWL states based on
                   the PNML transitions, since the states
                   will be referenced afterwards
155             final HashMap<String, Node>
                   stateNamesToStateNodeMap = new HashMap
                   <>();
156             final HashMap<String, Node>
                   stateNamesToActionNodeMap = new HashMap
                   <>();
157             final HashMap<String, String> transitions
                    = XMLParserCommons.
                   getTransitionNameIdMap(net);
158             for (Map.Entry<String, String>
                   transitionEntry : transitions.entrySet
                   ()) {
159               String transitionName = transitionEntry
                     .getKey();
160               String transitionIdentifier =
                     transitionEntry.getValue();
161
162               String stateType = getStateType(
                     pnmlArcs, transitionIdentifier,
                     placeIdToMessagesMap);
```

```
163                 if(stateType.equals("DoState")){
164                   doStatesIdToNameMap.put(
                        transitionIdentifier,
                        transitionName);
165                 } else if (stateType.equals("SendState"
                     )){
166                   sendStatesIdToNameMap.put(
                        transitionIdentifier,
                        transitionName);
167                 }
168
169                 Node stateNode;
170                 Node actionNode;
171
172                 if(stateNamesToStateNodeMap.containsKey
                     (transitionName)){
173                   stateNode = stateNamesToStateNodeMap.
                        get(transitionName);
174                   actionNode =
                        stateNamesToActionNodeMap.get(
                        transitionName);
175                 } else {
176                   stateNode = createNamedIndividual(
                        resultDocument, "SBD_"+name+"_"+
                        stateType+"_"+stateId, stateType, "
                        SBD_"+name+"_"+stateType+"_"+
                        stateId, transitionName);
177                   rdfNode.appendChild(stateNode);
178                   addHasFunctionSpecificationElement(
                        resultDocument, stateNode,
                        stateType);
179                   addContainsElement(resultDocument,
                        subjectNameToBehaviorNodeMap.get(
                        name), stateNode);
180
181                   String actionIdentifier = "
                        action_of_SBD_"+name+"_"+stateType+
                        "_"+stateId;
182                   actionNode = createNamedIndividual(
                        resultDocument, actionIdentifier, "
                        Action", actionIdentifier,
                        actionIdentifier);
183                   rdfNode.appendChild(actionNode);
184                   addContainsElement(resultDocument,
                        actionNode, stateNode);
185                   addContainsElement(resultDocument,
                        subjectNameToBehaviorNodeMap.get(
```

```
                    name), actionNode);
186
187
188                 stateNamesToStateNodeMap.put(
                       transitionName, stateNode);
189                 stateNamesToActionNodeMap.put(
                       transitionName, actionNode);
190                 stateId++;
191               }
192
193               if(isInitialState(
                     directLinksBetweenTransitions,
                     transitionIdentifier)){
194                 addInitialStateElement(resultDocument
                       , stateNode);
195                 addResourceElement(resultDocument,
                       subjectNameToBehaviorNodeMap.get(
                       name), stateNode, "hasInitialState"
                       );
196               } else if(isEndState(
                     directLinksBetweenTransitions,
                     transitionIdentifier)){
197                 addEndStateElement(resultDocument,
                       stateNode);
198                 addResourceElement(resultDocument,
                       subjectNameToBehaviorNodeMap.get(
                       name), stateNode, "hasEndState");
199               }
200
201               transitionIdToStateNodeMap.put(
                     transitionIdentifier, stateNode);
202               transitionIdToActionNodeMap.put(
                     transitionIdentifier, actionNode);
203             }
204
205             //Then create OWL transitions, that
                  reference the states created before
206             for(Arc arc :
                  directLinksBetweenTransitions){
207               HashMap<String, List<Node>>
                     transitionNodes = new HashMap<>();
208               if(doStatesIdToNameMap.containsKey(arc.
                     getSource())){
209                 String transitionName =
                       doStatesIdToNameMap.get(arc.
                       getSource());
```

```
210                     Node transitionNode =
                           createNamedIndividual(
                           resultDocument, "SBD_"+name+"
                           _StandardTransition_"+transitionId,
                            "StandardTransition", "SBD_"+name+
                           "_StandardTransition_"+transitionId
                           , transitionName+" done");
211                     transitionNodes.put(arc.getSource(),
                           Arrays.asList(transitionNode));
212                     transitionId++;
213                   } else if(sendStatesIdToNameMap.
                       containsKey(arc.getSource())){
214                     List<Message> messages = new
                           ArrayList<>();
215                     arc.getRefersToMessagePlaceIds().
                           forEach(mpId -> messages.add(
                           messagePlaceIdToMessageMap.get(mpId
                           )));
216                     Node transitionNode;
217
218                     for(Message message : messages){
219                       if (!message.getRecipient().equals(
                             name)){
220                         transitionNode =
                              createNamedIndividual(
                              resultDocument, "SBD_"+name+"
                              _SendTransition_"+transitionId,
                               "SendTransition", "SBD_"+name+
                              "_SendTransition_"+transitionId
                              , "To: "+message.getRecipient()
                              +" Msg: "+message.getName());
221                         String transitionConditionLabel =
                               "sendTransitionCondition_"+"
                              SBD_"+name+"_SendTransition_"+
                              transitionId;
222                         Node transitionConditionNode =
                              createNamedIndividual(
                              resultDocument,
                              transitionConditionLabel, "
                              SendTransitionCondition",
                              transitionConditionLabel,
                              transitionConditionLabel);
223                         rdfNode.appendChild(
                              transitionConditionNode);
224                         addResourceElement(resultDocument
                              , transitionNode,
                              transitionConditionNode, "
```

```
                        hasTransitionCondition");
225
226                     String messageIdentifier = "
                        Message: "+message.getName() +
                        " From: "+message.getSender() +
                         " To: "+message.getRecipient()
                        ;
227 addResourceElement(resultDocument, transitionNode,
    messageNameToMessageExchangeNodeMap.get(
    messageIdentifier), "refersTo");
228
229                     addToMapList(transitionNodes, arc
                        .getSource(), transitionNode);
230                     transitionId++;
231                   }
232                 }
233
234               } else {
235                 List<Message> messages = new
                    ArrayList<>();
236                 arc.getRefersToMessagePlaceIds().
                    forEach(mpId -> messages.add(
                    messagePlaceIdToMessageMap.get(mpId
                    )));
237                 Node transitionNode;
238
239                 for(Message message : messages){
240                   if (!message.getSender().equals(
                       name)){
241                     transitionNode =
                        createNamedIndividual(
                        resultDocument, "SBD_"+name+"
                        _ReceiveTransition_"+
                        transitionId, "
                        ReceiveTransition", "SBD_"+name
                        +"_ReceiveTransition_"+
                        transitionId, "From: "+message.
                        getSender()+" Msg: "+message.
                        getName());
242
243                     String messageIdentifier = "
                        Message: "+message.getName() +
                        " From: "+message.getSender() +
                         " To: "+message.getRecipient()
                        ;
244 addResourceElement(resultDocument, transitionNode,
    messageNameToMessageExchangeNodeMap.get(
```

```
    messageIdentifier), "refersTo");
245
246                     addToMapList(transitionNodes, arc
                            .getSource(), transitionNode);
247                     transitionId++;
248                   }
249                 }
250               }
251
252               transitionNodes.forEach((id, nodes) ->
                     {
253                 nodes.forEach(transitionNode -> {
254                   rdfNode.appendChild(transitionNode)
                          ;
255                   addPriorityNumberElement(
                          resultDocument, transitionNode);
256                   Node source =
                          transitionIdToStateNodeMap.get(
                          arc.getSource());
257                   Node target =
                          transitionIdToStateNodeMap.get(
                          arc.getTarget());
258
259                   addResourceElement(resultDocument,
                          transitionNode, source, "
                          hasSourceState");
260                   addResourceElement(resultDocument,
                          transitionNode, target, "
                          hasTargetState");
261                   addContainsElement(resultDocument,
                          Arrays.asList(
                          subjectNameToBehaviorNodeMap.get(
                          name),
                          transitionIdToActionNodeMap.get(
                          id)), transitionNode);
262                 });
263               });
264             }
265           }
266         }
267       } catch (final Exception e) {
268         LOG.error(e.getMessage());
269         LOG.error("Exception while generating OWL");
270       }
271     }
272
273     StreamResult documentResult;
```

```
274     final DOMSource source = new DOMSource(
            resultDocument);
275
276     Writer writer = new StringWriter();
277     documentResult = new StreamResult(writer);
278     final TransformerFactory transformerFactory =
            TransformerFactory.newInstance();
279     final Transformer transformer =
            transformerFactory.newTransformer();
280     transformer.transform(source, documentResult);
281
282
283     //This is necessary since inline doctype is
            erased by the transformer Source: https://
            stackoverflow.com/questions/3509860/how-can-i-
            retain-doctype-info-in-xml-documnet-during-read
            -write-process
284     String docType =  "<!DOCTYPE rdf:RDF [ " +
285             "    <!ENTITY owl \"http://www.w3.org
                    /2002/07/owl#\" >" +
286             "    <!ENTITY xsd \"http://www.w3.org
                    /2001/XMLSchema#\" >" +
287             "    <!ENTITY rdfs \"http://www.w3.org
                    /2000/01/rdf-schema#\" >" +
288             "    <!ENTITY abstract-pass-ont \"http://
                    www.imi.kit.edu/abstract-pass-ont#\" >"
                     +
289             "    <!ENTITY standard-pass-ont \"http://
                    www.i2pm.net/standard-pass-ont#\" >" +
290             "    <!ENTITY rdf \"http://www.w3.org
                    /1999/02/22-rdf-syntax-ns#\" >" +
291             "]>";
292
293     String documentString = writer.toString();
294     documentString = documentString.replaceFirst("<
            rdf:RDF", docType+"<rdf:RDF");
295     documentString = documentString.replaceAll("&amp;
            ", "&");
296
297     Writer resultWriter = new StringWriter();
298     resultWriter.write(documentString);
299     StreamResult result = new StreamResult(
            resultWriter);
300     return result;
301
302 }
303
```

```
304   private Node getOWLSkeleton(Document doc,
          DocumentBuilder builder) throws IOException,
          SAXException {
305     String skeleton = "<?xml version=\"1.0\"?>" +
306             "" +
307             "<rdf:RDF xmlns:abstract-pass-ont=\"http
                  ://www.imi.kit.edu/abstract-pass-ont#\"
                   xmlns:standard-pass-ont=\"http://www.
                  i2pm.net/standard-pass-ont#\" xmlns:rdf
                  =\"http://www.w3.org/1999/02/22-rdf-
                  syntax-ns#\" xmlns:owl=\"http://www.w3.
                  org/2002/07/owl#\" xmlns:xsd=\"http://
                  www.w3.org/2001/XMLSchema#\" xmlns:rdfs
                  =\"http://www.w3.org/2000/01/rdf-schema
                  #\" xmlns=\"" + ontologyUri + "\">" +
308             " <owl:Ontology rdf:about=\""+ontologyUri
                  +"\">" +
309             "       <owl:versionIRI rdf:resource=\""+
                  ontologyUri+"\"></owl:versionIRI>" +
310             "   <owl:imports rdf:resource=\"http://
                  www.imi.kit.edu/abstract-pass-ont\"></
                  owl:imports>" +
311             "   <owl:imports rdf:resource=\"http://
                  www.i2pm.net/standard-pass-ont\"></owl:
                  imports>" +
312             " </owl:Ontology>" +
313             "</rdf:RDF>";
314
315     Document doc2 = builder.parse(new
          ByteArrayInputStream(skeleton.getBytes()));
316     Node node = doc.importNode(doc2.
          getDocumentElement(), true);
317     doc.appendChild(node);
318     return node;
319   }
320
321   private Node createNamedIndividual(Document doc,
          String aboutName, String type, String id, String
          label) {
322     final Element namedIndividual = doc.createElement
          ("owl:NamedIndividual");
323     namedIndividual.setAttribute("rdf:about",
          ontologyUri+"#"+aboutName);
324
325     final Element rdfType = doc.createElement("rdf:
          type");
```

```
326     rdfType.setAttribute("rdf:resource",
           standardPassOntUri+type);
327     namedIndividual.appendChild(rdfType);
328
329     final Element componentId = doc.createElement("
           standard-pass-ont:hasModelComponentID");
330     componentId.setAttribute("rdf:datatype", "&xsd;
           string");
331     componentId.appendChild(doc.createTextNode(id));
332     namedIndividual.appendChild(componentId);
333
334     final Element componentLabel = doc.createElement(
           "standard-pass-ont:hasModelComponentLabel");
335     componentLabel.setAttribute("xml:lang", "en");
336     componentLabel.appendChild(doc.createTextNode(
           label));
337     namedIndividual.appendChild(componentLabel);
338
339     return namedIndividual;
340   }
341
342   private void addContainsElement(Document doc, Node
        addToNode, Node resourceNode){
343     addResourceElement(doc, addToNode, resourceNode,
           "contains");
344   }
345
346   private void addContainsElement(Document doc, List<
        Node> addToNodes, Node resourceNode){
347     addToNodes.forEach(node -> addContainsElement(doc
           , node, resourceNode));
348   }
349
350   private void addPriorityNumberElement(Document doc,
        Node addToNode) {
351     final Element priorityNumber = doc.createElement(
           "standard-pass-ont:hasPriorityNumber");
352     priorityNumber.setAttribute("rdf:datatype", "http
           ://www.w3.org/2001/XMLSchema#positiveInteger");
353     priorityNumber.appendChild(doc.createTextNode("1"
           ));
354     addToNode.appendChild(priorityNumber);
355   }
356
357   private void
        addMaximumSubjectInstanceRestrictionElement(
        Document doc, Node addToNode) {
```

```
358     final Element restriction = doc.createElement("
           standard-pass-ont:
           hasMaximumSubjectInstanceRestriction");
359     restriction.setAttribute("rdf:datatype", "http://
           www.w3.org/2001/XMLSchema#integer");
360     restriction.appendChild(doc.createTextNode("1"));
361     addToNode.appendChild(restriction);
362   }
363
364   private void addHasFunctionSpecificationElement(
        Document doc, Node addToNode, String stateType) {
365     final Element specification = doc.createElement("
           standard-pass-ont:hasFunctionSpecification");
366
367     String type = "Do1_EnvoironmentChoice";
368     if (stateType.equals("SendState")) {
369       type = "Send";
370     } else if (stateType.equals("ReceiveState")) {
371       type = "Receive";
372     }
373
374     specification.setAttribute("rdf:resource", "http
           ://www.i2pm.net/standard-pass-ont#
           DefaultFunction"+type);
375     addToNode.appendChild(specification);
376   }
377
378   private void addResourceElement(Document doc, Node
        addToNode, Node resourceNode, String tag){
379     final Element resourceElement = doc.createElement
           ("standard-pass-ont:"+tag);
380     resourceElement.setAttribute("rdf:resource",
           resourceNode.getAttributes().getNamedItem("rdf:
           about").getNodeValue());
381     addToNode.appendChild(resourceElement);
382   }
383
384   private void addInitialStateElement(Document doc,
        Node addToNode){
385     final Element element = doc.createElement("rdf:
           type");
386     element.setAttribute("rdf:resource",
           standardPassOntUri+"InitialState");
387     addToNode.appendChild(element);
388   }
389
```

```
390   private void addEndStateElement(Document doc, Node
         addToNode){
391     final Element element = doc.createElement("rdf:
           type");
392     element.setAttribute("rdf:resource",
           standardPassOntUri+"EndState");
393     addToNode.appendChild(element);
394   }
395
396   private HashMap<String, List<Message>>
         getPlaceIdToMessagesMap(final Element net) {
397     HashMap<String, List<Message>>
           placeIdToMessagesMap = new HashMap<>();
398     final NodeList placeNodes = net.
           getElementsByTagName("place");
399     for (int i = 0; i < placeNodes.getLength(); i++)
           {
400       final Element placeNode = (Element) placeNodes.
             item(i);
401       final NodeList toolspecificNodes = placeNode.
             getElementsByTagName("toolspecific");
402       String id = placeNode.getAttributes().
             getNamedItem("id").getNodeValue();
403
404       if(toolspecificNodes.getLength() > 0){
405         NamedNodeMap attributes = toolspecificNodes.
              item(0).getAttributes();
406         String tool = attributes.getNamedItem("tool")
              .getNodeValue();
407
408         if(tool.equals("SBPM")){
409           String type = attributes.getNamedItem("type
                ").getNodeValue();
410           String message = attributes.getNamedItem("
                message").getNodeValue();
411           String recipient = attributes.getNamedItem(
                "recipient").getNodeValue();
412           String sender = attributes.getNamedItem("
                sender").getNodeValue();
413
414           Message messageObj = new Message(message,
                recipient, sender, id);
415
416           if(type.equals("receive")){
417             id = attributes.getNamedItem("
                  actualTargetId").getNodeValue();
418           }
```

```
419
420             if(placeIdToMessagesMap.containsKey(id)){
421               placeIdToMessagesMap.get(id).add(
                     messageObj);
422             } else {
423               List<Message> newList = new ArrayList<>()
                     ;
424               newList.add(messageObj);
425               placeIdToMessagesMap.put(id, newList);
426             }
427           }
428         }
429       }
430       return placeIdToMessagesMap;
431     }
432
433     private HashSet<String> getAllPlaceIds(final
           Element net) {
434       HashSet<String> allPlaceIds = new HashSet<>();
435       final NodeList placeNodes = net.
             getElementsByTagName("place");
436       for (int i = 0; i < placeNodes.getLength(); i++)
             {
437         final Element placeNode = (Element) placeNodes.
               item(i);
438         String id = placeNode.getAttributes().
               getNamedItem("id").getNodeValue();
439         allPlaceIds.add(id);
440       }
441       return allPlaceIds;
442     }
443
444     private HashSet<Arc>
           getDirectLinksBetweenTransitions(HashSet<String>
           allPlaceIds, Set<String> allMessagePlaceIds, List
           <Arc> pnmlArcs){
445       HashMap<String, List<String>>
             sourcePlaceIdToTargetTransitionIds = new
             HashMap<>();
446       HashMap<String, List<String>>
             targetPlaceIdToSourceTransitionIds = new
             HashMap<>();
447       HashMap<String, List<String>>
             transitionIdRefersToMessageIds = new HashMap
             <>();
448       HashSet<Arc> directLinksBetweenTransitions = new
             HashSet<>();
```

```
449     int arcId = 1;
450     for(Arc arc : pnmlArcs){
451       if(allMessagePlaceIds.contains(arc.getTarget())
            ){
452         //arc is for sending
453         addToMapList(transitionIdRefersToMessageIds,
              arc.getSource(), arc.getTarget());
454       }
455       else if(allMessagePlaceIds.contains(arc.
            getTarget()) || allMessagePlaceIds.contains(
            arc.getSource())){
456         //arc is for receiving
457         addToMapList(transitionIdRefersToMessageIds,
              arc.getTarget(), arc.getSource());
458       } else if(allPlaceIds.contains(arc.getSource())
            ){
459         //arc is from place (source) to transition (
              target)
460         if(sourcePlaceIdToTargetTransitionIds.
              containsKey(arc.getSource())){
461           sourcePlaceIdToTargetTransitionIds.get(arc.
                getSource()).add(arc.getTarget());
462         } else {
463           List<String> newList = new ArrayList<>();
464           newList.add(arc.getTarget());
465           sourcePlaceIdToTargetTransitionIds.put(arc.
                getSource(), newList);
466         }
467       } else if(allPlaceIds.contains(arc.getTarget())
            ){
468         //arc is from transition (source) to place (
              target)
469         if(targetPlaceIdToSourceTransitionIds.
              containsKey(arc.getTarget())) {
470           targetPlaceIdToSourceTransitionIds.get(arc.
                getTarget()).add(arc.getSource());
471         } else {
472           List<String> newList = new ArrayList<>();
473           newList.add(arc.getSource());
474           targetPlaceIdToSourceTransitionIds.put(arc.
                getTarget(), newList);
475         }
476       }
477     }
478
479     for (Map.Entry<String, List<String>> entry :
          targetPlaceIdToSourceTransitionIds.entrySet())
```

```
        {
480       List<String> sourceTransitionIds = entry.
             getValue();
481       List<String> targetTransitionIds =
             sourcePlaceIdToTargetTransitionIds.get(entry.
             getKey());
482
483       if(targetTransitionIds != null) {
484         for(String targetTransitionId :
               targetTransitionIds){
485           for(String sourceTransitionId :
                 sourceTransitionIds){
486             List<String> refersToMessageIds = new
                   ArrayList<>();
487             if(transitionIdRefersToMessageIds.
                   containsKey(sourceTransitionId)){
488               refersToMessageIds =
                     transitionIdRefersToMessageIds.get(
                     sourceTransitionId);
489             } else if (transitionIdRefersToMessageIds
                   .containsKey(targetTransitionId)){
490               refersToMessageIds =
                     transitionIdRefersToMessageIds.get(
                     targetTransitionId);
491             }
492
493             Arc arc = new Arc(String.valueOf(arcId),
                   sourceTransitionId, targetTransitionId,
                    refersToMessageIds);
494
495             directLinksBetweenTransitions.add(arc);
496             arcId++;
497           }
498         }
499       }
500     }
501
502     return directLinksBetweenTransitions;
503   }
504
505   private <K,V> void addToMapList(HashMap<K,List<V>>
         map, K key, V value){
506     if(map.containsKey(key)){
507       map.get(key).add(value);
508     } else {
509       List<V> newList = new ArrayList<>();
510       newList.add(value);
```

```
511         map.put(key, newList);
512       }
513     }
514
515     private String getStateType(List<Arc> arcs, String
          transitionId, HashMap<String, List<Message>>
          placeIdToMessageMap){
516       String stateType = "DoState";
517
518       Predicate<Arc> sourcePredicate = p -> p.getSource
            ().equals(transitionId);
519       List<Arc> arcsWithTransitionIdAsSource = arcs.
            stream().filter(sourcePredicate).collect(
            Collectors.toList());
520       for(Arc arc : arcsWithTransitionIdAsSource) {
521         if(placeIdToMessageMap.containsKey(arc.
              getTarget())){
522           stateType = "SendState";
523           break;
524         }
525       }
526
527       if(stateType.equals("DoState")){
528         Predicate<Arc> targetPredicate = p -> p.
              getTarget().equals(transitionId);
529         List<Arc> arcsWithTransitionIdAsTarget = arcs.
              stream().filter(targetPredicate).collect(
              Collectors.toList());
530         for(Arc arc : arcsWithTransitionIdAsTarget) {
531           if(placeIdToMessageMap.containsKey(arc.
                getTarget())){
532             stateType = "ReceiveState";
533             break;
534           }
535         }
536       }
537
538       return stateType;
539     }
540
541     private boolean isInitialState(Set<Arc> arcs,
          String transitionId){
542       Predicate<Arc> targetPredicate = p -> p.getTarget
            ().equals(transitionId);
543       return arcs.stream().filter(targetPredicate).
            count() < 1;
544     }
```

```
545
546   private boolean isEndState(Set<Arc> arcs, String
          transitionId){
547     Predicate<Arc> sourcePredicate = p -> p.getSource
          ().equals(transitionId);
548     return arcs.stream().filter(sourcePredicate).
          count() < 1;
549   }
550
551   private HashMap<String, Arc> getArcIdMap(final
          Element net) {
552     final HashMap<String, Arc> pnmlArcIdMap = new
          HashMap<>();
553     final NodeList arcNodes = net.
          getElementsByTagName("arc");
554
555     for (int i = 0; i < arcNodes.getLength(); i++) {
556       final Element arcNode = (Element) arcNodes.item
            (i);
557       String id = arcNode.getAttributes().
            getNamedItem("id").getNodeValue();
558       String source = arcNode.getAttributes().
            getNamedItem("source").getNodeValue();
559       String target = arcNode.getAttributes().
            getNamedItem("target").getNodeValue();
560
561       pnmlArcIdMap.put(id, new Arc(id, source, target
            ));
562     }
563
564     return pnmlArcIdMap;
565   }
566 }
```

Listing A.2: Source code of the Petri net to OWL transformation service

Bibliography

[1] Aagesen, G., and Krogstie, J. BPMN 2.0 for Modeling Business Processes. In *Handbook on Business Process Management 1*, J. vom Brocke and M. Rosemann, Eds., 2 ed. Springer, 2015, pp. 219–250.

[2] Barros, A. Process Choreography Modelling. In *Handbook on Business Process Management 1*, J. vom Brocke and M. Rosemann, Eds., 2 ed. Springer, 2015, pp. 279–300.

[3] Börger, E. A Subject-Oriented Interpreter Model for S-BPM. `http://www.di.unipi.it/boerger/Papers/Bpmn/SbpmBookAppendix.pdf` [Online: accessed 18/08/2017].

[4] Börger, E. The Subject-Oriented Approach to Software Design and the Abstract State Machines Method. In *S-BPM ONE - Scientific Research* (2012), C. Stary, Ed., Springer, pp. 1–21.

[5] Elstermann, M. Proposal for Using Semantic Technologies as a Means to Store and Exchange Subject-oriented Process Models. In *Proceedings of the 9th International Conference on Subject-Oriented Business Process Management* (2017), ACM.

[6] Fleischmann, A., Schmidt, W., Stary, C., Obermeier, S., and Börger, E. *Subject-Oriented Business Process Management*. Springer, 2012.

[7] Geisriegler, M., Kolodiy, M., and Stani, S. S-BPM Modelling and Execution Platform Github Repository. `https://github.com/stefanstaniAIM/IPPR2016/` [Online: accessed 13/08/2017].

[8] Geisriegler, M., Kolodiy, M., Stani, S., and Singer, R. Actor Based Business Process Modeling and Execution: a Reference Implementation Based on Ontology Models and Microservices. In *Proceedings of the 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2017)* (2017).

[9] Hand, D., Mannila, H., and Padhraic, S. *Principles of Data Mining*. The MIT Press, 2001.

[10] Höver, K. M., and Mühlhäuser, M. S-BPM-Ont: An Ontology for Describing and Interchanging S-BPM Processes. In *S-BPM ONE 2014 - Scientific Research* (2014), Springer, pp. 41–52.

[11] Kindler, E. Concepts, Status, and Future Directions. In *Entwurf Komplexer Automatisierungssysteme, EKA 2006* (2006), pp. 35–55.

[12] Leymann, F., Karastoyanova, D., and Papazoglou, M. P. Business Process Management Standards. In *Handbook on Business Process Management 1*, J. vom Brocke and M. Rosemann, Eds., 2 ed. Springer, 2015, pp. 595–624.

[13] Martens, A. Usability of Web services. In *Fourth International Conference on Web Information Systems Engineering Workshops, 2003. Proceedings.* (2003), pp. 182–190.

[14] Massuthe, P., Reisig, W., and Schmidt, K. An Operating Guideline Approach to the SOA. *Annals of Mathematics, Computing & Teleinformatics 1(3)* (2005), 35–43.

[15] SINGER, R., AND ZINSER, E. Business Process Management – S-BPM a New Paradigm for Competitive Advantage? In *S-BPM ONE - Setting the Stage for Subject-Oriented Business Process Management* (2010), Springer, pp. 48–70.

[16] STANI, S. Aufbau einer Microservice-basierten Architektur zur Speicherung und Ausführung von subjekt-orientierten Prozessen. Master's thesis, 2017.

[17] VAN DER AALST, W. *Process Mining: Data Science in Action*, 2 ed. Springer, 2016.

[18] VAN DER WERF, J., AND POST, R. EPNML 1.1: an XML format for Petri nets, 2004.