Automated Functional Architecture Generation using Process Discovery

E.J. Kaats
Utrecht University, Netherlands

Supervisors:
dr.ir. J.M.E.M. van der Werf
dr. M.M. Dastani

August 5, 2015

Abstract

This thesis describes a means to build a functional architecture out of event logs. A functional architecture is an abstract overview of how parts of a system work together. Event logs are sequential descriptions of events. These are often found as logs of a central Information System (IS) and are used by Process Mining algorithms to discover processes.

Our work describes a series of steps that – once completed – enable IS architects to gain an abstract view of a software environment. Often this is still done by hand through interviews and document analysis, rather than through evidential data.

We discuss several new concepts: The feature- and module nets as ways to describe horizontal and vertical data, and the behavioral matrix to combine these into a reference net. This reference net is an overview of all horizontal and vertical communication found within an event log. Indeed, our contribution is able to discover both horizontal as well as vertical behavior while other tools are only able to find the former.

Furthermore, we discuss ways to combine these new tools with fields outside of process mining. We see opportunities in the research of vertical behavior, as discovering internal behavior has easier using our tools. With these techniques, future research may have an easier task in formalizing horizontal behavior, possibly by combining this with multi-agent systems or automata theory.

Acknowledgments

This thesis has been a journey in which I learned a lot, mostly through trial and error but thanks to the tutoring of Jan Martijn,for well thought out theoretical approaches. Another round of thanks go out to Mehdi for taking the time to be my second supervisor.

Another bunch of thanks I'd like to give to the people at Volvo Cars Gent for spending hours of working with me selflessly.

Special thanks to Mariya for keeping me sane through the countless times where the art of thesis writing seemed to be more endurance than inspiration. She also helped me spell check this thesis, which really helped me. Not only to put punctuation marks on the right spots, but it also helped to make this into a coherent story. Also thanks to Nils, who also gave great feedback.

I am happy with the knowledge and skills I gained throughout these months and I appreciate the lessons that I have been thought at Utrecht University that prepared me for this task and I am proud that by finishing my thesis, in return have proven to be a proud alumnus in the future.

Of course the university not only offered me knowledge, but also friendship, mostly in the form of the student association Sticky which I want to thank for the liters of coffee that I've consumed there over the years. With this thesis my student times are truly at an end, and I look back at years of fun and friendship, but also gaining knowledge and wisdom.

Finally, I like to thank Douwe Egberts for creating the brew that kept me going during long coding spree's, the Python team for developing a programming language that is simple and fun to use (and grand the occasional anti-gravitational effect [1]), and LaTeX $2_\varepsilon$for saving me a whole lot of markup induced headaches.

[1] https://pypi.python.org/pypi/antigravity/0.1

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Business processes describe how companies operate, but underneath exists an intricate web of elementary business rules, which are stored in databases, programs and people's minds. These elements can be regarded as a (Finite-)State Machine (Gold, 1972); constructs of whose behavior is not random. Rather, a rule set forms the basis of behavior and can range from easily mappable behavior (e.g., the internal process of a vending machine) to very complex (e.g., a credit assessment department.) Not always is it clear how these state machines exactly operate, but they exhibit a predictable behavior which makes it possible to understand their inner workings. (Peled et al., 1999) State machines are not the only way to map uncharted waters. Business Process Management (BPM) (Dumas et al., 2013; Weske, 2007) takes an abstract approach by suggesting interviews or workshops to gather data. This holistic framework is at the base of most process improvement projects and, indeed is born out of the Business Process Re-Engineering efforts of the 1990s. A third existing option is Process Mining (Van Der Aalst and Weijters, 2005). Instead of analyzing the system or its anecdotal usage, event logs are analyzed by a computer algorithm, deriving process models from evidential data. This relatively new approach unites the formal world of state machines with the abstract approach of PBM.

However, while process mining delivers a reliable way to turn traces (process data describing a sequence of events) into various process models, this has only been done for 'flat', or horizontal processes. Current tools do not discover vertical processes. This is communication across multiple modules (e.g., departments, communities, organizations, or users) interacting between each other. Current solutions are able to define a process across these modules, but not with multiple control flows. Furthermore, currently there are no techniques that can be used to uncover the inner-modular communicating processes, which decide how features react to different kind of situations. Within this thesis, we describe new approaches to identify and analyze inner- and inter-modular behavior.

1.1 Problem Statement

Currently, anyone who wants to identify an existing process can either gather process logs and use a process mining algorithm to interpret this data, or can interview those who know (part) of the process and piece together a process from there. The former option could be very precise – that is, if enough reliable data is used. However, only the horizontal process can be found by current automatic mining programs. No solution exists that charts processes of an entire company, including

departmental borders. The second option could provide such a rich picture, but is a time demanding and high cost operation. In short, these two approaches have the following shortcomings:

1. Current methods do not regard resources, or the communication between them (inter-modular);
2. There are currently no methods to automatically discover inner-modular behavior by observing external communication.

In order to better understand the current paradigm,we present several examples. An in depth case is presented as a running example, and is discussed in 1.3. For now, consider a small log, where x, y represent modules:

$\{\langle A^x, B^x, C^y, D^y, E^x\rangle, \langle < A^x, C^y, E^x\rangle\}$

Current algorithms would find a process that lacks any kind of information on modules (as depicted in 1.1). Also note that there is no information on why choices are made. A sometimes communicates to B, or to C. In the log, we can see that A is only followed by C, if C is followed by E.

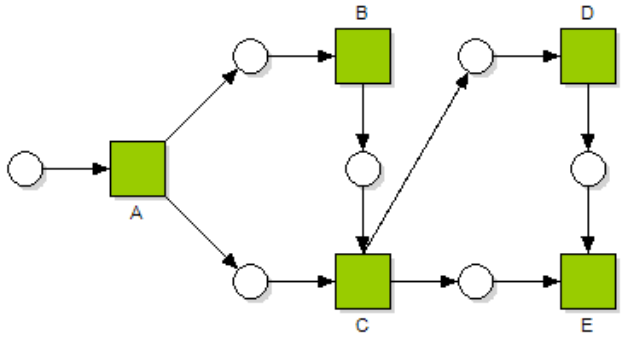


Figure 1.1: Current Algorithms (constructed manually)

The Business Process Management notation (Illustration 1.2) does support displaying choices, but as mentioned, no mining algorithm or technique exists to fully support its features – and most importantly, the distinction between different modules. Do note that the referred illustration is a simplification of the BPMN standard, and omits some of its design rules. For instance, it does not re-join the 'or' paths.

In order to solve these issues, we describe the current state-of-the-art in process mining and BPM methods regarding to horizontal behavior, and we formalize both inner and inter modular communication. Furthermore, we present a means to use current mining algorithms to map and analyze both of these types of behavior. The presented method includes requirements on source data, and other limitations which are to be solved in further research.

1.2 Relevance

This thesis addresses issues in the social environment as well as scientific field. Social issues are those that make adopting process mining in enterprises difficult. Scientific issues are those that are known to slow academic research on this subject.

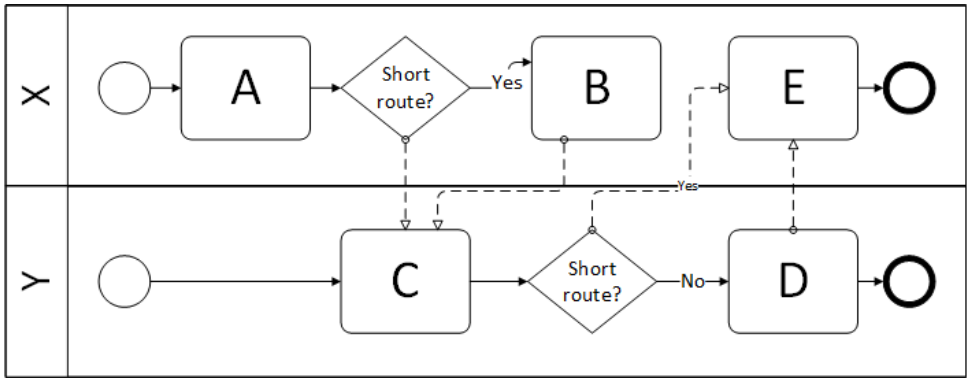


Figure 1.2: Business Process Management Notation

scientific relevance

Process mining is a field of study that has enjoyed a lot of research in the past decade. However, most research is focused on new mining algorithms rather than functional applicability. Dealing with large 'spaghetti' processes is still one of the main challenges (Van Der Aalst, 2012). Our solution makes it easier to 'chop up' these processes and allows for generating smaller, more easily analyzable models. Ideally this will drive new research that not only focuses on algorithms that produce sound models, but also allow for research in log preparation to advance.

social relevance

Process Mining is a young and promising field that has not been widely adopted outside of academia. While commercial process mining solutions are available (Turner et al., 2012), it has yet to gain wide acceptance. Although Process Mining has been championed as the new 'killer-app' to innovate the decades old field Business Process Management (Schmiedel and vom Brocke, 2015), it has a few challenges to overcome itself. One of these hurdles is flexibility (Claes and Poels, 2013). Although algorithms exist that can map a process with varying degrees of accuracy, they lack the ability to chart what actually matters – behavior in- and between systems. Without this ability, it is impossible to mine complex processes. This thesis aims to lessen the gap between practical questions, now answered using PBM and automated analysis of source data as provided by current PM algorithms.

1.3 Running Example

In order to further clarify the current challenges we address, we use the (fictional) application process of a new student applying to a university. In this process, a couple of events can happen, listed in Table 1.1 These events are always completed in a certain order. One of such sequence is called a trace, as displayed in 1.2. In this scenario, several things can happen. The simplest path is when the student simply gets accepted or rejected. When the student did not supply enough information, the university can ask the student to re-submit their application. When this is done successfully, the process continues as normal. Another alternative path is taken if the student did

Short	Event	Party
A	Send application	Student
B	Receive request	
C	Receive letter of acceptance	
D	Receive rejection + reasons	
E	Archive application	
F	Receive documents	University
G	Review documents	
H	Request additional information	
I	Make decision	
L	Send decision	
J	Request reference information	
N	Archive application	
O	Receive request	Reference
P	Send documents	

Table 1.1: Running Example Events

supply all necessary information, but the university decides to ask for additional data from a third source. Finally, combinations of these paths are also possible.

Note that each of these paths (e.g., rejection, acceptance, etc.)can be written down using three slightly different sequences. This happens when two parties are able to complete process steps without the need for extra input. For example, if the student is accepted, the university can either immediately close and archive the students files, or it may wait for archival. In those cases the student may receive the good news before the university process is completed.

The process can be moddles using a BPMN diagram, as show in Illustration 1.3. Note that each module has its own process, going from the round start node, through the solid arrows, towards the thick-round end node. This strictly means there are really three process traces per main process, which is called a module log. However, in real-life situations more often than not data is saved like as shown in Table 1.2, where an overarching Enterprise Resource Planning (ERP) system saves process data along many nodes and departments. Lastly, a diagram like this would become unreadable if analyzed with current algorithms that do not differentiate between modules (as discussed in Section 1.1)

1.4 Outline

This thesis is structured as follows: First, in Chapter 2, we discuss the scope, goals and challenges of the research, as well as the different approaches we use during the thesis.

Chapter 3 discusses background literature as well as preliminary notions used in the remainder of the thesis. This chapter is divided into three sections that describe architecture, process and feature behavior.

After introducing this theoretical basis, we describe our approached to the problems that are discussed earlier in the research approach. This is done in chapter 4 discussed the same three segments, albeit in a slightly different order.

Case	Trace	Description
1	A, F, G, I, L, D, E, N	Rejected
2	A, F, G, I, L, D, N, E	
3	A, F, G, I, L, N, D, E	
4	A, F, G, I, L, C, E, N	accepted
5	A, F, G, I, L, C, N, E	
6	A, F, G, I, L, N, C, E	
7	A, F, G, J, O, P, F, G, I, L, D, E, N	rejection +
8	A, F, G, J, O, P, F, G, I, L, D, N, E	reference
9	A, F, G, J, O, P, F, G, I, L, N, D, E	
10	A, F, G, J, O, P, F, G, I, L, C, E, N	accepted +
11	A, F, G, J, O, P, F, G, I, L, C, N, E	reference
12	A, F, G, J, O, P, F, G, I, L, N, C, E	
13	A, F, G, H, B, A, F, G, I, L, D, E, N	rejection +
14	A, F, G, H, B, A, F, G, I, L, D, N, E	additional info
15	A, F, G, H, B, A, F, G, I, L, N, D, E	
16	A, F, G, H, B, A, F, G, I, L, N, C, E	accepted +
17	A, F, G, H, B, A, F, G, I, L, C, E, N	additional info
18	A, F, G, H, B, A, F, G, I, L, C, N, E	
19	A, F, G, H, B, A, F, G, J, O, P, F, G, I, L, D, E, N	rejection +
20	A, F, G, H, B, A, F, G, J, O, P, F, G, I, L, D, N, E	additional info +
21	A, F, G, H, B, A, F, G, J, O, P, F, G, I, L, N, D, E	reference
22	A, F, G, H, B, A, F, G, J, O, P, F, G, I, L, N, C, E	accepted +
23	A, F, G, H, B, A, F, G, J, O, P, F, G, I, L, C, N, E	additional info +
24	A, F, G, H, B, A, F, G, J, O, P, F, G, I, L, C, E, N	reference

Table 1.2: Running Example Traces

The solutions are further presented in Chapter 5 where we present a proof of concept of the proposed solutions and finally concluded and discussed in Chapter **??** where we discuss future research and discuss identified limitations of our model.

For extra background information, there is a detailed discussion on the prototyping phases in Appendix B.

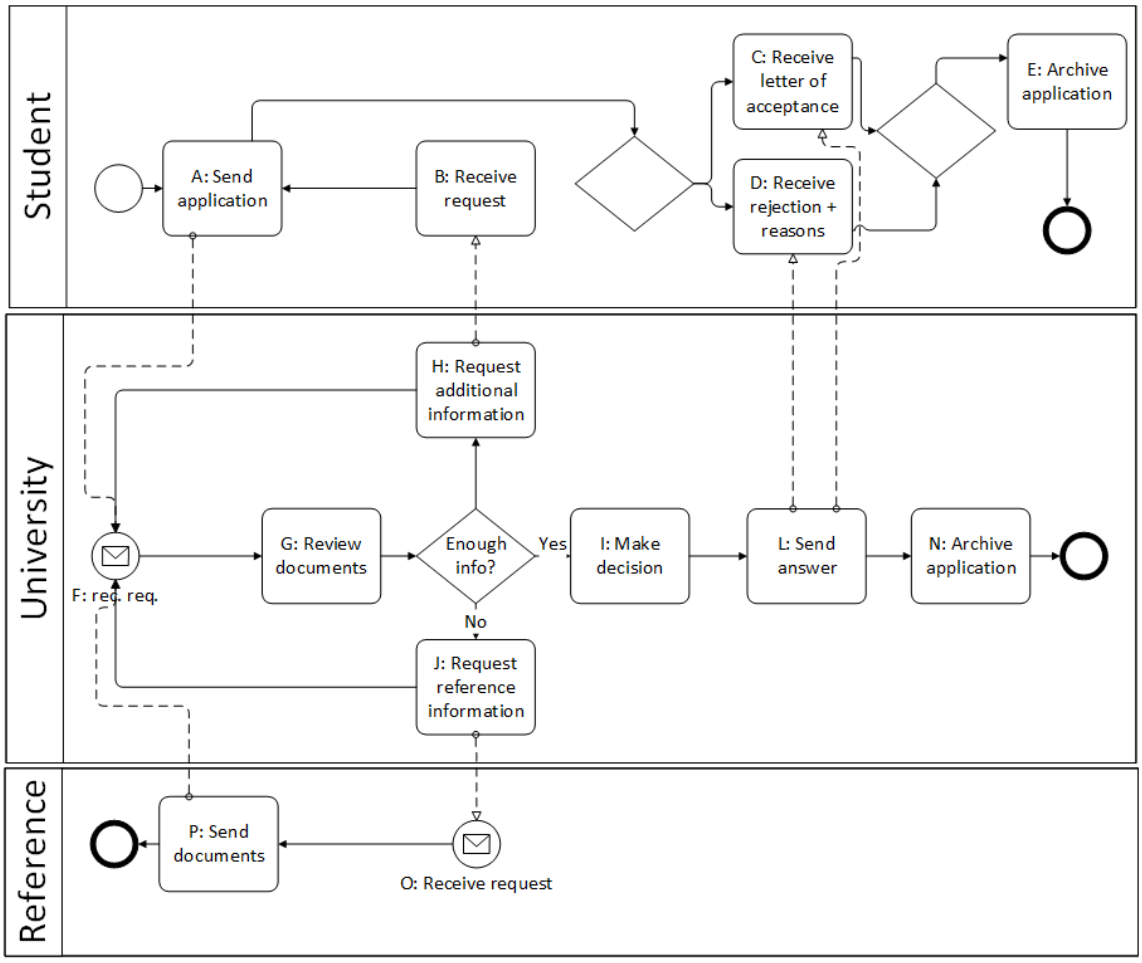


Figure 1.3: The Running Example in BPMN

Chapter 2

Research Approach

Our research is based on the principles of Design Science, rather than Behavioral Sciences. While the latter examines known, current phenomena, Design Science constructs a new artifact based on current knowledge with the goal of solving a real-life problem, or case. This approach leads to a different research framework and indeed, different research questions compared to Behavioral Sciences (Hevner et al., 2004). Figure 2.1 depicts a simplified version of the conceptual framework of Hevner et al. (2004), which shows the knowledge base, in research, and the environment interact.

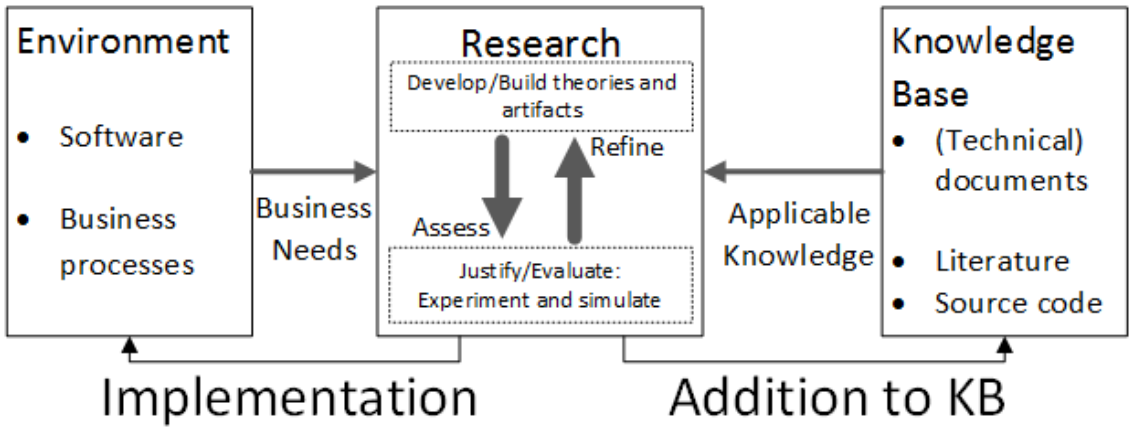


Figure 2.1: Design Science Based Approach

Within the environment, we find concepts from every day life; people, organizations and technology are the three main subjects. For this thesis, software and business processes are the most important as the research is focused on the technical challenges, and is not meant to solve an immediate, existing problem. Rather, the case is based on an abstracted concept.

The knowledge base is home of foundations and methodologies, which translate into documentation, scientific literature and source code for this case. These components are used within the research to design and develop new artifacts and to test these using simulations and case studies.

The research itself is conducted in a series of iterations; at the end of each cycle, the construct is evaluated, knowledge is added to the knowledge base and where applicable, used to guide the next cycle. A test setup assesses the ability of the artifact's to achieve the overall goals. Within the first cycles artificially constructed data is used, while in the later stages the artifact is tested using real-world data.

2.1 Research Questions

This section serves as a reading guide as well as an explanation of choices regarding the research setup. After first stating the main question that sums up the entire thesis, two questions regarding current techniques are phrased, followed by a transformative question. The remaining three questions describe the explorative nature of this thesis. First, the main research question is stated as follows:

(RQ): "How can processes in a collection of (inter-)operating systems, known or unknown, be properly *identified*, *analyzed*, and *utilized*, using current methods, techniques and notations as a basis?"

Here 'identified' is connected with Process Identification within the BPM cycle, as described by Dumas et al. (2013), later depicted in Picture 3.3 on Page 17. 'analyzed' is mapped to Process Discovery and Process Analysis and 'utilized' refers to Process Redesign and Process Implementation.

In previous research, the development of different methods to identify processes has already been described, notably Process mining (Van Der Aalst and Weijters, 2005) and Business Process Management (Weske, 2007). These methods not only exhibit different techniques, their notations also differ across fields. The current state-of-the-art needs to be assessed in order to select (the best combination of) methods, techniques and notations for the artifact.

(SQ1): "What different methods, techniques and notations exist to model behavior between systems?"

This question is answered by conducting a literature review whose setup is discussed in Section 2.2 while the outcomes are listed in Chapter 3. An aspect we are especially interested in, is how current techniques come up short, and with what consequences. For example, Brown and Kros (2003) discuss the impact of missing or noisy data. A type of data that has been absent from literature are Unknown processes; there has not been a lot of discussion surrounding unknown but suspected data. Thus, extra attention is given to the following question;

(SQ2): "What difficulties currently exist regarding behavioral modeling?"

and is discussed in Section 3.5. Having answered the first two questions using a literature review, a prototype setup is created by answering the following research question:

(SQ3): "How can current techniques be combined to improve on identified difficulties?"

This question is discussed in Chapter 4, where the challenges and outcomes of our prototype phase is discussed. Rather than solely relying on current literature to assess the usefulness of techniques described in these publications, new and novel constructs are build using insights and approaches from established methods. During several prototype iterations the usefulness of different types of communication are tested and described. This process, which is described as follows:

(SQ4): "How does inter-feature communication need to be defined in order to describe behavior, and how does it relate to horizontal behavior?"

explores not only the attributes of behavior, but also the requirements on input data. This research question is explored upon in the later prototyping stages and answered within the same chapter as SQ3 (Chapter 4). Using this information, a Functional Architecture Model can automatically be created to describe the information found in the formal, and informal flows between systems.

(SQ5): "How is communication captured in Functional Architectures?".

Having generated a FAM out of carefully formulated artificial data according to earlier discovered constraints, we describe the final process with the following question:

(SQ6): "What steps are necessary to transform event logs into a FAM?"

This is discussed in Chapter 5

2.2 Research Model

The research model as depicted on Illustration 2.2 is as follows: Research question (SQ1) and (SQ2) will be answered by conducting a literature study. Together, these steps produce our first deliverable; an overview of the current state of the art, together with contemporary issues. The prototyping phase follows hereafter and consists of research question (SQ3) and (SQ4). Finally, after a suitable prototype has been constructed it is tested by answering Question (SQ5) which assesses its ability to automatically create a FAM, and (SQ6) which tests this capability in real-life.

Literature Study

Our literature study covers a lot of areas, including Process Mining, Workflow Management, Business Process Management (BPM) and Automata Theory. Included in these areas are various process modeling techniques such as BPMN, BPEL, Petri-nets, Workflow nets, etc. These fields are briefly discussed in this chapter, while more information on these fields can be found in Chapter 3.

Process mining involves the automatic reconstruction of processes from event data (Van Der Aalst and Weijters, 2005), Workflow Management (WM) describes both the modeling of processes, the different flows that can be followed and the means by which humans can interact or partake in this process. (Georgakopoulos et al., 1995). Where only aimed at involving IT to improve current processes, BPM extends this concept with a life cycle among other tools that are business oriented

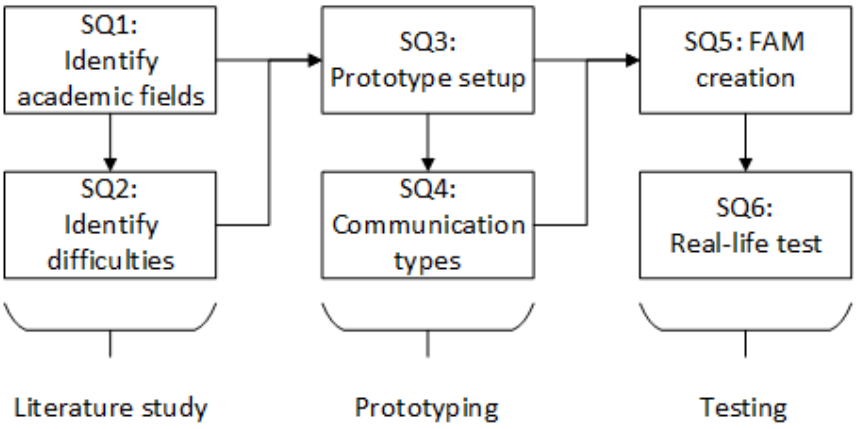


Figure 2.2: The Proposed Research Model

(Van Der Aalst et al., 2003). Finally, Automata Theory has sprouted an immense amount of related fields.

For this literature review, we did not use the often used PRISMA method (Moher and Liberati, 2009). Since our goal is not to constuct an exhaustive corpus of the aforementioned fields, but rather to use this knowledge to design and test a new artifact hence, a formal literature study protocol is not necessary.

A preliminary literature study showed that BPM literature, although plentiful turned out to be of little help as methods are too abstract to use in this thesis. Process Mining on the other hand proved to be a new field with only a handful of active authors. While useful, we found that this pool was quickly exhausted. Lastly, a lot has been published on both automata and modeling techniques, but most articles are too detailed and applicable in very limited situations. Thus these factors make a fully formal literature research unneeded.

Rather, we decided on applying the 'snowball' technique. Automata theory is a mature field and peaked in the 1970's. Being a formal mathematical field, theories can prove relevant for decades. For these subjects, we applied forward searches: by picking an influential paper from a few decades back and searching through those that have cited this publication, new and novel knowledge can be found more quickly than doing a traditional keyword search (Jalali and Wohlin, 2012).

Prototyping

This thesis is build around the goal of designing, testing, and possibly implementing a solution to an existing problem and similar future issues, and is roughly divided into two phases: the setup and execution of each iteration and the examination of outcomes. When discussing the first part within Appendix B, each iteration is discussed using an adaption of a commonly used design science model.

The prototyping phase split up into several distinct phases. Each of these iterations is based upon findings of the previous one, thus improving on earlier design. In order to structure the work-flow between these iterations, we use the model depicted in Figure 2.3. Rather than reinstating the

problem with every iteration, the method is only aligned after which current shortcomings are taken into account and the analysis phase starts a new iteration.

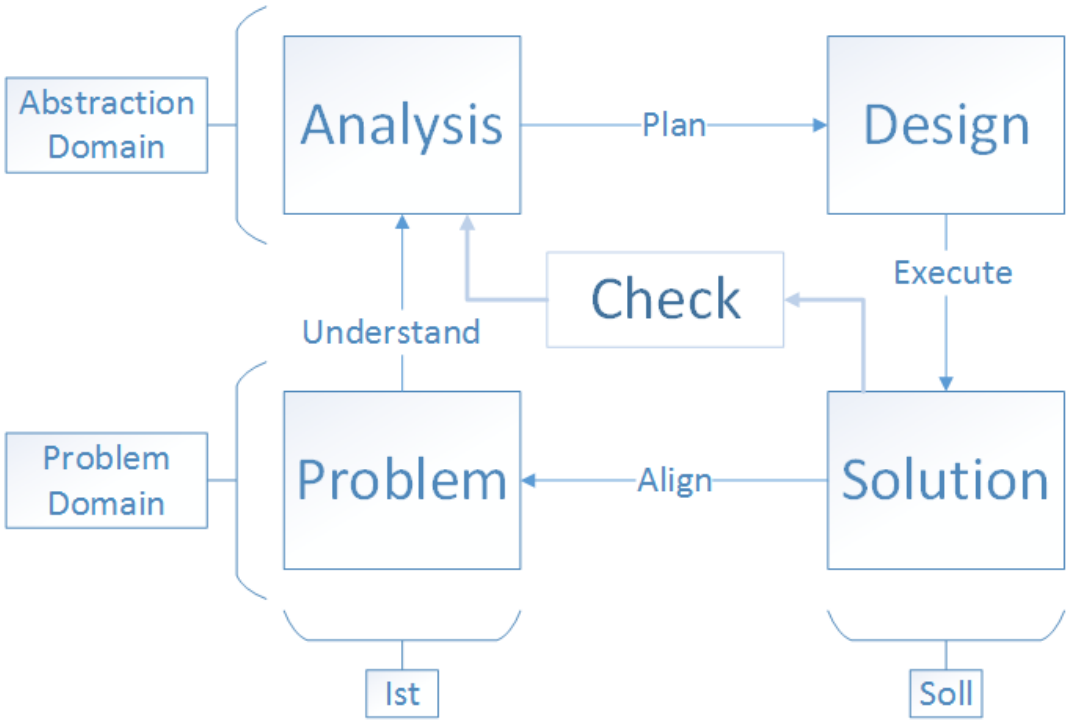


Figure 2.3: The Iterative Design Science Process

Case Study

There are two disciplines that study business processes, yet for this – and presumably many other real-life situations – none of these prove to be a good fit on their own for describing a functional architecture. We conduct a case study to present a proof of concept and is discussed in Chapter 4.

We suspect there to be shortcomings that haven't been accounted for during design time because real-life data is always less of a fit than artificially constructed data that is specifically created for testing. Because it is hard – if not impossible – to account for this, we describe identified problems in the end to make future research easier.

Chapter 3

Background

Within this chapter the four different aspects related to this thesis are discussed. In order to better understand the goals and opportunities as well as the place each aspect resides at, we discuss the current state-of-art, how these aspects work and how they can be used in the remainder of the research. With this, the first three research questions are answered in this chapter.

The order of this chapter goes from high level, abstract models to detailed concepts. First, (functional) architectures are examined, followed by (business) processes. After processes themselves, we discuss process discovery - the principles of process mining. Lastly, we discuss agent behavior and the inner processes of agents that drive behavior. All of these subjects describe challenges that make up the goals of this thesis, with the exception of process mining, which is an enabling technique used in the next chapter.

3.1 Architectures

Software architectures are found in many different shapes and forms. For our purposes, we discuss two different high-level tools that are used to identify and manage (business) processes. Functional (software) architectures represent information flow in and between software products. Business process management is used as a means to control these processes.

Functional architectures

Brinkkemper and Pachidi (2010) define a Functional Architecture as "an architectural model which represents at a high level the software product's major functions from a usage perspective, and specifies the interactions of functions, internally between each other and externally with other products." Illustration 3.1 shows on what level these architectures reside. Sometimes they include multiple information systems (IS), while other times only cover one solution.

An architecture is described in a Functional Architecture model, which is discussed in the next section. It is followed by a description of Business Process Management, which does not describe an architecture but a means to manage both high and low-level processes within a system.

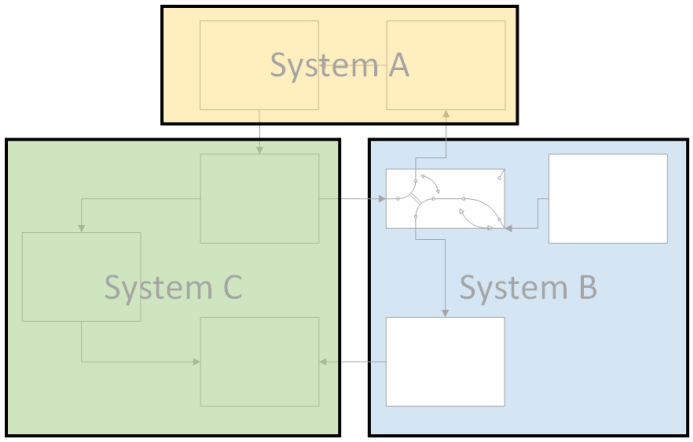


Figure 3.1: Functional Architectures

Functional Architecture Model (FAM)

A functional architecture model is a "representation of the primary functionality of a software product, consisting of its main functions and supportive operations" (Brinkkemper and Pachidi, 2010) and is used to identify functions performed by different software products, and the interactions involved in that usage. Using a FAM, an enterprise architect can identify what parts need to be renovated, which are underperforming and what measures influence the business in a positive way. In a way, a FAM is much like the building plans a contractor might use when renovating an apartment complex.

Much like actual building plans, a FAM needs to be kept up to date. Every time someone runs new cables or tubing through the walls, the plans need to be updated. Indeed, instead of only reflecting a static, envisioned state like blueprints do, a FAM needs to reflect the current situation as much as it does the past. It consists of *modules* and *features*, and their *connections*.

Definition 1 (FAM) *A* Functional Architecture Model *(FAM) is defined as a 6-tuple* $\langle \mathcal{M}, \mathcal{C}, \mathcal{F}, h, m, \rightarrow \rangle$ *where*

- $\mathcal{M}$ *is a finite set of* modules*;*
- $\mathcal{C}$ *is a finite set of* context modules*;*
- $\mathcal{F}$ *is a finite set of* features*;*
- $h : \mathcal{M} \rightarrow \mathcal{M}$ *is the* hierarchy function*, such that the transitive closure* h^* *is irreflexive;*
- $m : \mathcal{F} \rightarrow \mathcal{M} \cup \mathcal{C}$ *is a* feature map *that maps each feature to a module, possibly in the context, and this module does not have any children, i.e.* $h^{-1}(m(F)) = \varnothing$ *for all* $F \in \mathcal{F}$*;*
- $\rightarrow \subseteq \mathcal{F} \times \Lambda \times \mathcal{F}$ *is the* information flow*, with* Λ *the label universe, such that for* $(A, l, B) \in \rightarrow$ *we have* $m(A) \neq m(B)$*. The labels for the information flows are unique per feature, i.e.,* (A, l, B) *and* (A, l, C) *imply* $B = C$ *for all labels* $l \in \Lambda$ *and* $(A, l, B), (A, l, C) \in \ \rightarrow$*.*

A FAM can also contain the scenarios like ones described in 3.2 van Der Werf and Kaats (2015) describe this in more detail.

3.2 Processes

One level bellow functional architectures we position the intra-feature processes. On this level the control flow is defined. This is the path that defines a scenario and can span multiple autonomous features and make up a possible (business) process. Figure 3.2 shows an interpretation of this level. Note that the arrows describe a path that a process can take, e.g., the red arrows describe an ordering process that is relayed to an external supplier (System A) and directed back to System B. The blue arrows could show a process that also starts internally and is relayed to an external partner through another internal department. Where architectures describe paths that *can* be taken by processes, control flow describes which path *are* taken by process instances.

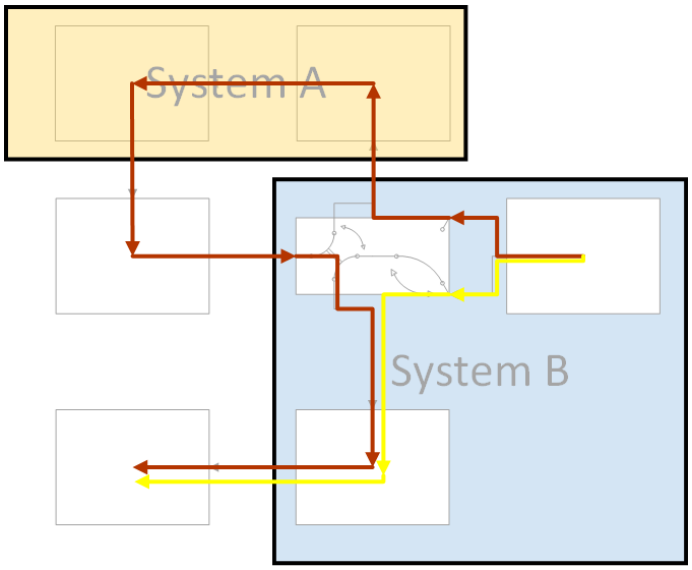


Figure 3.2: Horizontal Processes

Most processes are likely to be solved in different ways, depending on the decisions made at each feature. Each time a process is run and its steps are recorded, it is done in a trace. A trace is a sequential list of events connected to one process 'instance'. First we discuss these traces in more detail after which we discuss process mining, which is a technique to automate discovering processes using logs of traces. Lastly, we discuss Business Process Modeling and the many notations to describe processes.

Petri Nets

Petri nets form the foundation of many behavioral modeling techniques and are used as a modeling language for the feature- and module nets. Petri nets are defined as follows:

Definition 2 (Petri Net) *A* Petri net *(Reisig, 2012) is a tuple* $N = \langle P, T, F \rangle$ *where (1)* P *and* T *are two disjoint sets of* places *and* transitions *respectively; and (2)* $F \subseteq (P \times T) \cup (T \times P)$ *is the* flow relation.

The elements from the set $P \cup T$ *are called the* nodes *of* N*. Elements of* F *are called* arcs*. Places are depicted as circles, transitions as squares. For each element* $(n_1, n_2) \in F$*, an arc is drawn from* n_1 *to* n_2*.*

A petri net can have a marking that can be used to denote scenarios within the process. However, scenario's are not used in the final model. van Der Werf and Kaats (2015) discusses this subject in more detail.

Business Process Modeling

Throughout the past decades, a lot of different modeling notations have been developed (Ko et al., 2009; Aguilar-Savén, 2004), each of which with different applications. Business Process Modeling Language is generally regarded as the standard when it comes to process design (Chinosi and Trombetta, 2012), but certainly is not the only one. Ko et al. (2009) divide modeling techniques into six different sets: Theory (Petri-Net, Calculus), Graphical (EPC, Petri-Net,UML AD, YAWL, BPMN), Interchange (BODM, XPDL), Execution (BPEL, BPML, etc), Diagnosis (BPQL, BPRI, etc), and B2B info exchange (Rosetta-Net, UBL, etc). Most of these are industry-developed, except for EPC, Petri-Net, Calculus, and YAWL, which all have academic origins.

Aguilar-Savén (2004) describes 18 different model types and maps them on either as leading to an active- (Workflows, UML, Petri-Nets, etc.), or passive change (Gantt chart, flow chart, etc.). Another distinction is the purpose of the model: descriptive for learning (Gantt chart, Role activity diagram, Etc), Process development/design (Flow chart, UML, Workflow), Process execution (UML, Workflow, etc.) and enactment (Petri-Nets, etc.). Most of these models serve multiple purposes.

Models can be either informal, i.e., they do not abide to strict (mathematical) rules, or they are formal and follow a concise set of rules. BPMN can be considered an informal language. While there are constraints (e.g., a XOR gate needs to be closed with the same type or, a process always needs to have a start and an end), it is often a lot easier to doodle up an informal diagram to convey a message than it is to construct a formal one. According to Chinosi and Trombetta (2012) over 50 percent of the BPMN users use it for documenting purposes and one third to execute business processes. However, when analyzing a process or system, this level of formalism is necessary.

BPEL is often seen as the executable extension to BPMN and is easily convertible into Petri-Nets (Hinz et al., 2005; Lohmann et al., 2009; Lohmann, 2008). Automata are sometimes also generated from BPEL sources, as discussed by Fahland (2005) and Wombacher (2004).

Business Process Management

BPM is a holistic management discipline that started during the 80s when business processes became more and more standardized. In particular, the car industry improved its processes using Business Process Re-engineering (BPR), which lead to a leaner structure and less overhead. Using then emerging technologies like ERP systems, old processes could either be omitted completely, or be otherwise automated (Weske, 2007; Dumas et al., 2013). In the end of the 90s, the notion of process improvement lead to BPM by adding notions like process ownership, and adding a complete framework with techniques, notations and methods to shape processes throughout their life-cycle.

Indeed, while BPR was initially only developed to improve on individual processes, BPM takes the entire process architecture and its development in consideration.

BPM is a somewhat de-focussed field; there is a common consensus that the business process life-cycle is the center of the framework, but the shape and scope of this life-cycle differs from source to source. Often, these four fundamentals are found: process design, system configuration, process enactment and diagnosis. Weske (2007) adds the notion of administration and stakeholders playing a central role in it and clarifies the different phases somewhat. Mendling (2008) further adds monitoring and analysis to the model and Dumas et al. (2013) rewrote the model as discussed in the coming subsection. Although each iteration adds new improvements, there are enough reasons to use an older version. As Ko et al. (2009) argue, there are specific situations that benefit from either a more applied setup, or an abstract approach, or perhaps a focused model.

BPM Cycle

The BPM life-cycle describes the different phases a process undergoes. It has countless variants, each with its own specific focus. We chose to use the model of Dumas et al. (2013) as it is the most complete version. The first two phases of the life-cycle are very relevant for our research and we discuss them in the following subsections.

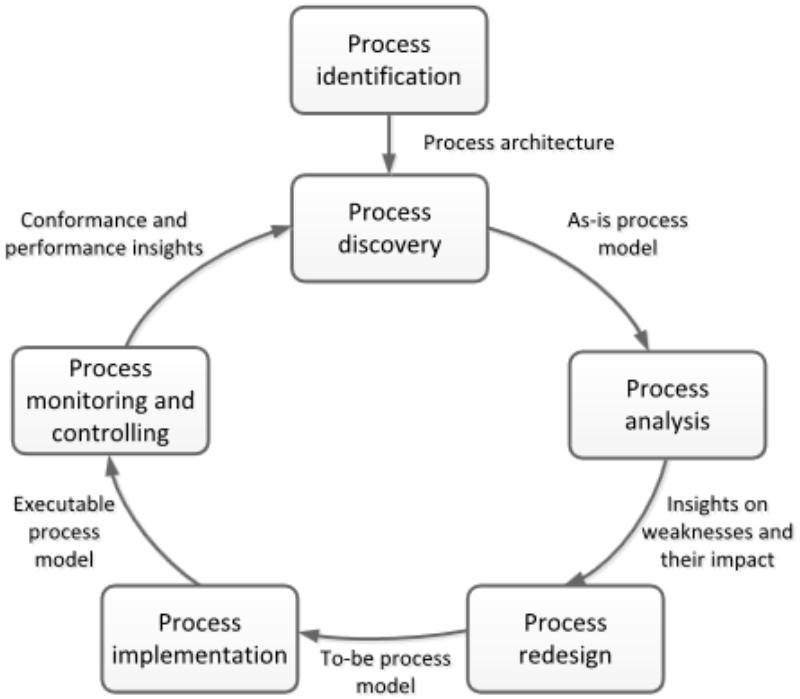


Figure 3.3: The BPM cycle (Adapted from Dumas et al. (2013))

Process identification Surprisingly, not a lot has been written about the identification of processes. Damij et al. (2008) and Weske (2007) discuss this step, but do not describe any detailed methods, other than conducting interviews. Damij et al. (2008) do discuss the creation of a Process Table, in which they link the Business Processes with the operational Work Processes. Furthermore, Grosskopf et al. (2010) discuss a workshop-like approach to complement interviews but to the best of our knowledge, there is no quantitative approach to identify a process. Indeed, while process mining is often used to discover (but not identify) processes, the amount of data is often too vast to easily convert into a usable dataset without tacit knowledge about the processes (Van Der Aalst and Weijters, 2005).

A reason for this knowledge gap might be found in the different systems used within companies. While EDI (Electronic Data Interchange) systems are still the most frequently used way of automatic data transfer in and between companies (Engel et al., 2011), implementation differs greatly between companies. This system is often no more than a set of formal rules for formatting messages. Furthermore, these systems do not have any process awareness and thus, data mining is done bottom-up, without any prior process knowledge, as opposed to systems that are process aware (Dumas et al., 2005). The latter directly provide the event logs needed for process mining activities (Van Der Aalst and Weijters, 2005).

The end product of Process Identification is the Process Architecture (Dumas et al., 2013). While individual work processes might not be identified yet, this high-level overview can already be constructed without detailed domain and business knowledge.

Process discovery The discovery phase can be conducted in three ways, or any combination thereof (Dumas et al., 2013). There is Evidence-Based discovery, where either documents analyses takes place, or processes are observed. Process mining is one of the direct ways of observing activities. Another way is through interview-based or workshop-based discovery. The traditional difference between Process identification and discovery is that the latter not only 'acknowledges' the existing of the process, but also identifies several attributes. Dumas et al. (2013) describe the following phases:

1. Identify the process boundaries
2. Identify activities and events
3. Identify resources and their hand-overs
4. Identify the control flow
5. Identify additional elements.

Another difference between identification and discovery is that the former does not check for process soundness, while process discovery does. It involves syntactic quality and semantic soundness tests. This ensures processes are valid before they are tested in the later stages.

3.3 Process Discovery

Process mining is a set of techniques that makes it possible to 'map' processes using event-based data (Van Der Aalst and Weijters, 2005). This is a collective term for the automatic, evidence-based

techniques available today. It provides a quantitative approach to discover, explore and benchmark processes across time.

In recent years process mining has been suggested as a solution for mapping unknown processes (Van Der Aalst, 2011) but so far this area is yet to be explored. Often, getting the data is one of the biggest challenges when dealing with unknown systems. Dumas et al. (2013) identify three different methods: Evidence-Based (e.g., Document analysis, Process mining), Interview-Based and Workshop-Based. Dumas et al. (2013) note that evidence based techniques are objective but conversely do not receive immediate feedback, whereas interviews and workshops are less objective but more immediate and receive feedback from the organization.

While formally not restricted to input or output formats, feeding the source data into workflow nets and other Petri-Net derivatives seem to be the best choice (Van Der Aalst and Weijters, 2005; van Hee et al., 2013), as it is being based on formal languages but also easy to understand. Sources for process mining are most likely EDI systems using a standard like EDIFACT, ANSI 12 or in some cases the newer and more favored XML standard to send messages back and forth (Nurmilaakso, 2008). In order to process this data, it is generally converted to an eXtensible Event Stream (XES)(Verbeek and Buijs, 2011) and analyzed with a specialized application (ProM). There are a number of different process mining algorithms that are in the center of this application, for more information on the algorithms themselves, refer to Appendix 3.3.

Event log and Traces

Let T be a set of Tasks (Activities) and R a set of Resources. An event log L is a sequence of events over tuples of activities and resources, i.e., $L \subseteq (T)^*$ A trace is an element of an event log, i. e., a sequence over tuples of activities and resources. A horizontal trace can contain events from multiple modules. Depicted here are three traces within log $\mathcal{L}$:
$\mathcal{L} = \{\langle A_1, B_1, C_2, D_2, E_1, F_1\rangle, \langle C_2, D_2, E_1, F_1\rangle, \langle Z_1, A_1, B_1, C_2, D_2, E_1, F_1 >\rangle\}$.
Two modules are present within the log (1 and 2), each containing various features. When this log is converted to a module log, the traces of module 2 looks like this:
$t_1 = \{C_2, D_2\}, t_2 = \{C_2, D_2\}, t_3 = \{C_2, D_2\}$.
Activities are similar to features and resources to modules, as used in the field of Requirements Management (Schobbens et al., 2006).

Modules and Features

A module is an atomic concept that is used in an event and is part of a system or of other modules. They can contain either sub-modules, or features, but not both (van Der Werf and Kaats, 2015). Modules are used in Functional architectures where Requirements Engineering might use resources.

Like a module, a feature is an atomic concept. We define a feature as follows: "A feature is a discrete unit of unique and attractive functionality of a system". Every feature belongs to exactly one module. In Requirements engineering, a feature is called a concepts (Schobbens et al., 2006).

Behavioral Profile

From the order of events in traces we derive the event relations as noted down in the functional architecture matrix. This is done as follows: we select an event and its direct successor from a trace such that $a <_{\mathcal{L}} b$ if a sequence $\sigma \in \mathcal{L}$ and $1 \leqslant i \leqslant |\sigma|$ exist, such that $\sigma(i) = a$ and $\sigma(i+1) = b$.

This relationship is defined in one for the following ways:

1. causality relation, where $\rightarrow_c$ is defined by $a \rightarrow_c b$ iff $a <_{\mathcal{L}} b$ and $b \not<_{\mathcal{L}} a$
2. concurrency relation, where $\|_c$, which is defined by $a \|_c b$ iff both $a <_{\mathcal{L}} b$ and $b <_{\mathcal{L}} a$
3. exclusive relation, where $+_c$ is defined by $a +_c b$ iff both $a \not<_{\mathcal{L}} b$ and $b \not<_{\mathcal{L}} a$ (Weidlich and Van Der Werf, 2012).

If the context is clear, we omit the subscript. These relations together form the behavioral profile (Weidlich and Van Der Werf, 2012):

Definition 3 *Given an event log $\mathcal{L}$, we define the* successor relation *by $a <_{\mathcal{L}} b$ if a sequence $\sigma \in \mathcal{L}$ and $1 \leqslant i \leqslant |\sigma|$ exist, such that $\sigma(i) = a$ and $\sigma(i+1) = b$. Using the successor relation, we define the* behavioral profile *$(\rightarrow_c, \|_c, +_c)_{\mathcal{L}}$ as three relations: (1) the causality relation $\rightarrow_c$ is defined by $a \rightarrow_c b$ iff $a <_{\mathcal{L}} b$ and $b \not<_{\mathcal{L}} a$, (2) the concurrency relation $\|_c$, which is defined by $a \|_c b$ iff both $a <_{\mathcal{L}} b$ and $b <_{\mathcal{L}} a$, and (3) the exclusive relation $+_c$ is defined by $a +_c b$ iff both $a \not<_{\mathcal{L}} b$ and $b \not<_{\mathcal{L}} a$. If the context is clear, we omit the subscript.*

We use this profile later on to establish which features are actually connected and which are not.

Mining Algorithms

While there have been attempts to standardize and categorize process mining algorithms as suggested by Rozinat and de Medeiros (2008), a formal standard is not available, which complicates the comparison process. We limited ourselves to the miners which are supported in ProM and the top 5 most used techniques as stated by Claes and Poels (2013). Apart from that, we only describe techniques that we use or are of special interest to us. Consult Buijs (2014) for a more complete overview,

Alphaminer The first process algorithm (Van Der Aalst et al., 2004) is simplistic in design and forms the basis of the mining algorithms. It only considers event order within traces, and is very sensitive to noise, since each occurrence is mapped to the model, even 'unfinished' traces or faulty ones.

Heuristics and Genetic Miner The Heuristics Miner is the second oldest process mining algorithm and solves several limitations the Alpha miner. Rather than using a particular trace directly in a graph, it first computes the dependencies between all events (Weijters, 2006). The only parameter used within this measurement is the event sequence for which the event name (the value of 'concept:name' in XES event logs) is used. Like most other algorithms, no other information is taken into account. It is a proven algorithm and still the most widely used (Claes and Poels, 2013). However, apart from threshold adjustments, the algorithm itself is not easily adaptable because of its reliance on a dependency matrix.

Genetic Miner uses the casual matrix of Heuristics miner and improves on it by using a genetic algorithm to include non-local causality data, that is, it improves the association between events that are not directly connected (de Medeiros, 2007). Output is generated as a Petri-Net.

Evolutionary Tree Miner Another tree mining algorithm (Buijs, 2014), capable of generating process trees is depicted in Illustration 3.4. The tree is read as follows: starting with the branch, the arrow shows that first an order is received, after which either the branch of message 2 is taken, or the first message is sent. Afterwards, the order is shipped. Trees are usable since (parts of) a branch can be hidden without interrupting the process. If we did not have any information on the additional steps needed to send message 2, these would be 'hidden' within the tree. However, like most process mining algorithms, this tree mining algorithm also only concerns the process perspective. The author of this method did create a comparison method in order to compare business processes within different companies, but this method is limited to comparing process models.

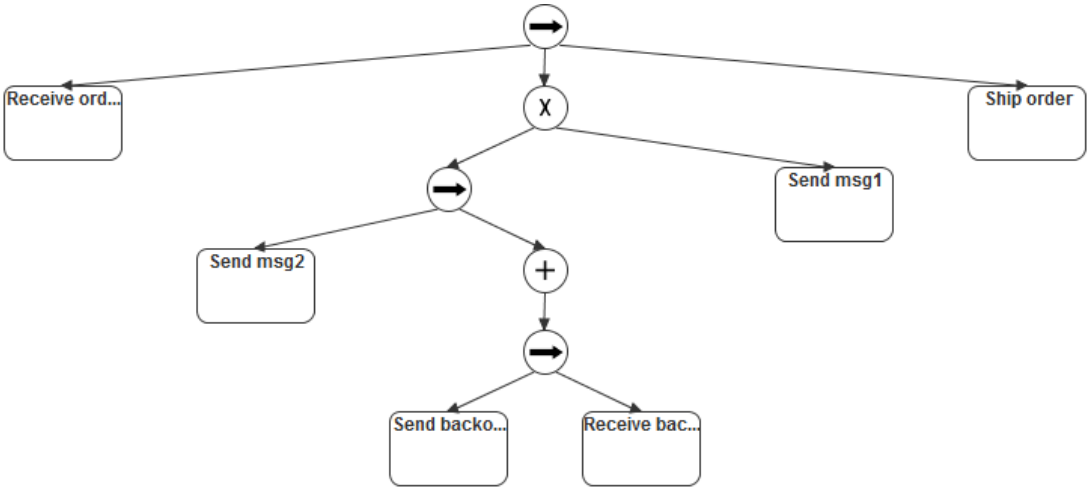


Figure 3.4: A Process Tree of Both Interfaces of B Generated with ETM

Inductive Miner Just as the Passage Miner, the Inductive miner 'cuts up' traces into smaller parts using a 'divide-and-conquer' technique (Leemans, 2014). Frequencies are ignored when iterating through a log but are used to differentiate between frequent and infrequent behavior. This miner only regards event order within traces, just as most of the discussed techniques.

Social Network Miner This algorithm is atypical, as it does not regard control flow at all. Rather, it mines the organizational perspective. This results in a matrix consisting of collaboration data between resources. It constructs 4 distinct metrics (Van Der Aalst et al., 2005):

- Metrics based on (possible) causality (Handover of work);
- Metrics based on joint cases;
- Metrics based on joint activities;
- Metrics based on special event types.

All of these metrics are based on organizational meta-data within the log, such as the person, department or company executing the task. Control flow is not used directly but rather as a

constraint (e.g. causality within events in 'handover of work', or 'subcontracting' between events) although individual traces or events are never included in the generated Social Networks.

3.4 Agent Behavior

Under agent behavior we group disciplines that study the role, reactions and decisions made by an agent, and in particular those who are situated within a multi-agent environment. The object of study are the *internal* processes, as can be seen in Illustration 3.5, where an agent has certain input and output nodes and somehow has to decide when to interact in what manner. This image shows the internal process of a module. This vertical behavior differs from horizontal behavior in that it describes in- and output but not anything after that.

This example shows the process of a procurement system. It has a stock management function, can analyze a tender on validity and evaluate it in more detail. When looking at vertical behavior, we can see how decisions are made. While horizontal behavior only shows what happened opposed to why. We see that a request of office supplies is either granted or declined based on a simple stock management function, while a bigger tender request is first analyzed and evaluated. The vertical behavior also describes internal routes. A tender is analyzed and either rejected or evaluated further, but never accepted directly after the analyze tender phase.

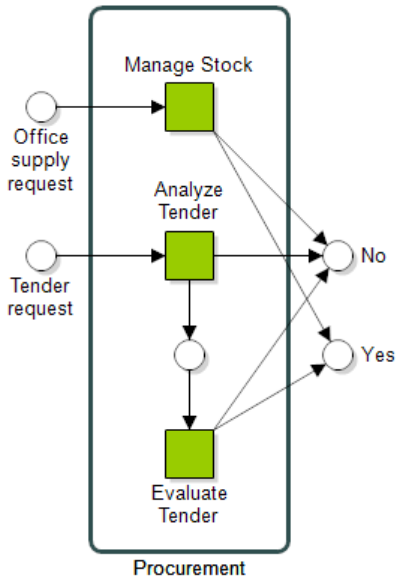


Figure 3.5: Vertical processes or Agent Behavior

We discuss two disciplines: Automata theory and Multiple-Agent Systems. These fields describe the behavior of features that have been discussed before. While neither of these disciplines contribute directly to this research, they do serve as influences and could benefit from our findings in return.

Automata theory

One of the approaches to identify the inner workings of a system is found within the Automata theory (Peled et al., 1999; Gold, 1972). An automaton is an abstract, self operating machine which can take many forms. It can either be infinite or finite, meaning it can have a known, limited set of states, or possess an unknown and unlimited amount of places and states. Transitions between states can also change the automaton type. In deterministic systems, the next state is always known, while states in a non-deterministic automaton have the ability to output to multiple other states. These systems are relevant to this research in that both automata and feature nets seek to describe the inner processes of a feature.

Finite State machines (FSM) have also been suggested as a conceptual model for e-services (Berardi et al., 2003) and there are solutions that are able to generate FSMs from Abstract State Machines (ASM) (Grieskamp et al., 2002). Further trends are found in Learning Automata. e.g., self-learning systems (Raffelt et al., 2009) which can be used when implementing and later, controlling a system.

Multiple-Agent Systems

Just as Automata, Multi-Agent Systems can be divided into several different categories. The two main approaches are those that represent the cognitive state of rational agents, and those seek to represent the strategic structure of a multi-agent environment (der Hoek and Wooldridge, 2008). Cognitive state agents are comparable to the inner-feature behavior that we seek to identify. It is based on the notion that agents – be it human or artificial – make decisions that drive behavior. When agents interact, this behavior forms a process that can be described as a control flow.

In order to understand the actions and intentions of an agent most systems use 'folk psychology', which uses terms such as 'anger', 'desire' and 'belief' to describe the state of a system. A system that is especially interesting is the BDI Logics by Rao and Georgeff (1995). It describes a system that has certain beliefs (about the environment), has certain desires and intentions. Using these concepts, one can describe scenarios such as: *Mail server*: 'I Believe this incoming message is not spam and want to keep my cache empty so I forward it to the recipient'.

Because Multiple-Agent Systems are studied as a group of agents rather than a single entity, internal behavior cannot be seen separately from the environment. Take the mail server as an example, it 'believes' the email is a legitimate message but in order to know this it has to have an understanding of what is and is not spam. Furthermore, the email itself is an external concept. Indeed, an agent is often presented with vast amounts of external signals. Broersen et al. (2001) describe an approach to cope with internal conflicts as a result of this information overflow.

Internal processes are not the only aspect that is subject to research. Multiple-Agent Systems have also been proposed as a solution for the increasing demands in flexibility in distributed manufacturing (Leitão, 2009) of example. This kind of research differs from that of Enterprise Architecture and approaches such as BPM in that there is no standard process. Rather, decisions are made in a distributed fashion (Maík and McFarlane, 2005), as depicted in Figure 3.6. Where a conventional system works by 'telling' every system what to do, a distributed approach based on agent systems works by enabling individual agents to make decisions. In a traditional system, one might install a load balancer to divide tasks between systems while agents in a distributed system would accept a tasks when they are able to and reject it when they already have enough work to do. In short: Decision making is done by agents, rather than by a central entity.

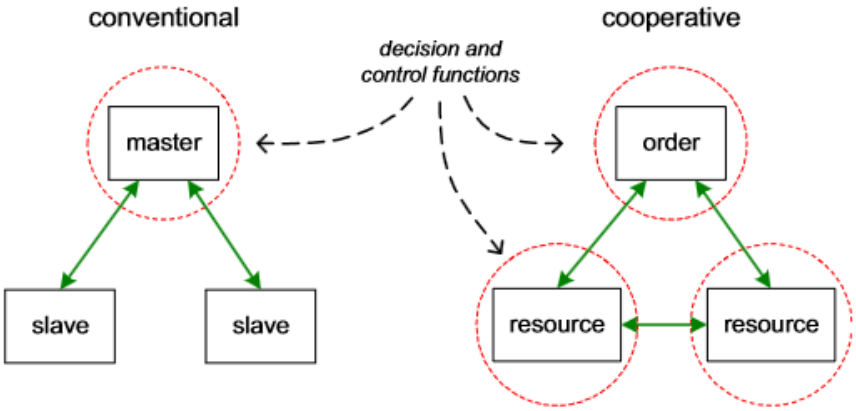


Figure 3.6: Distributed Decision Making (Adopted from (Maík and McFarlane, 2005))

3.5 Conclusions

Every subject discussed in this chapter has its own set of challenges. We briefly summarize these here. These challenges are further discussed in the next chapter.

Architectures The primary challenge on architecture level is to be able to keep apart the different levels of communication, as well as keeping architecture size manageable, for example by implementing a modular design.

processes The processes within a module need to be properly identified and mined so that inner-feature processes become possible, as well as inner-modular behavior. This also needs to be done while keeping the control flow intact.

inner-feature behavior The behavior of a feature is not easily accessible from a process log. In order to describe the intentions of an agent, the logs need to be split before using a process mining tool, while maintaining the same overall control flow.

Chapter 4

Communication Discovery

In this chapter we propose several techniques to remedy the challenges discussed in the previous chapter. In order to do this, we first describe the current problem in the next section, followed by the proposed solutions, each in their own section.

4.1 Problem statement

The current techniques and their shortcomings in regard of FAM generation have briefly been discussed in Chapter 1. To give a better overview of the abilities and issues of current tools, we analyze the running example using the inductive miner as described by Leemans (2014). Illustration 4.1 shows the output of the full log. It consists of one start and one end node and shows the general control flow.

From the mined model, it is difficult to analyse how the different systems communicate. Also, the social network miner does not profile any new and useful insights as shown in Figure 4.1.

The solutions are listed in a different order as the challenges in the Background chapter. This is because solutions such as identifying modules and communication between them are parts of the solution, and need to be completed in order. This is discussed in Section 4.2. The second issue we solve is inner-feature behavior discovery in Section 4.3. Finally, the communication behavioral profile and module net presented as the first solution, and the feature nets presented in the second are combined into a Functional Architecture Model in 4.4.

4.2 Identify communication between modules

The first challenge is to correctly identify what modules are present in the event log and to find the control flow within every module. This is done by creating a module log that describe the process of each module separately.

In the remainder of this chapter, let T denote the set of activities and let $L \subseteq (T)^*$ be an event log. Let $\mathfrak{R}()$: $A \rightarrow R$ be a mapping of activities to resources.

A module log is an event log which consists of the events of one particular module and its underlying features.

Definition 4 (Module Log) *Let $\mathcal{L} \subseteq T^*$ be an event log. Let $M \in Rng(\mathfrak{R})$ be a module. The* Module log $\mathcal{L}_M$ *is defined by $\mathcal{L}_M = \{\sigma_{|\{F|\mathfrak{R}(F)=M\}} \mid \sigma \in \mathcal{L}\}$.*

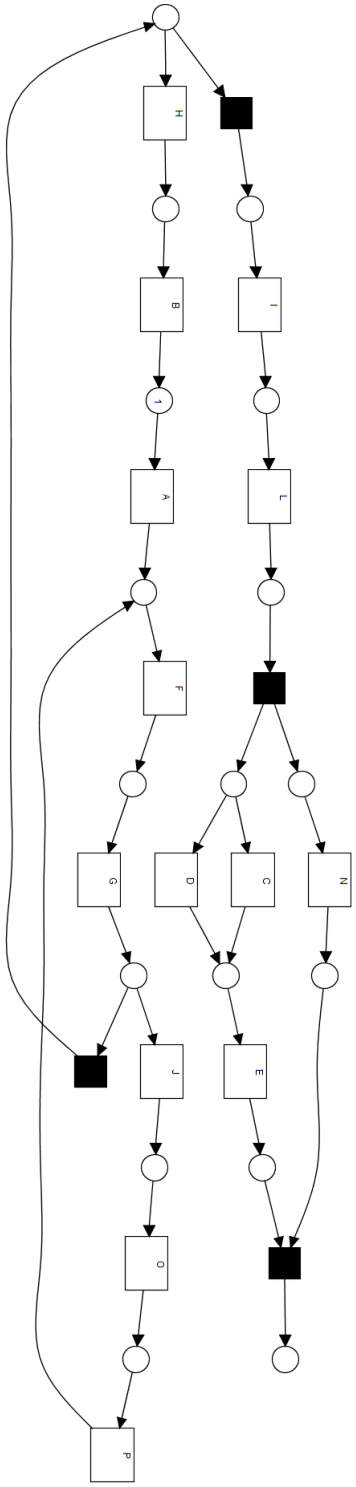


Figure 4.1: The Running Example Mined with Current Tools

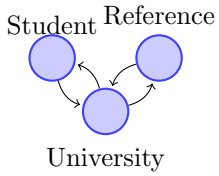
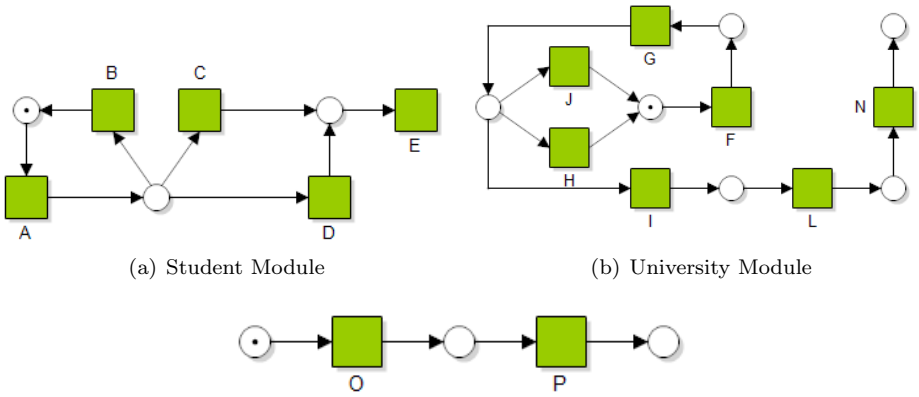


Figure 4.2: Handover of Work Social Network of Running Example

Using these logs, a process net is mined for every log. This is simply called a module net. A module net contains the entire control flow of one module. Figure 4.3 shows the module nets of the running example. A module net itself is not used on its own but is combined with feature nets. It can be seen as an intermediate step to create the FAM.



(a) Student Module

(b) University Module

(c) Reference Module

Figure 4.3: The Three Modules from Running Example

While module nets show the process flow in a module, they also lose information on communication between modules. In order to preserve this information we generate a behavioral profile first.

This sort of profile is based on the fact that communication between modules is always asynchronous. While module 'student' and 'university' both do send and receive data from each other, there are multiple features to facilitate this communication. When a feature is found to be both sending to (i.e. appears directly in front in a trace) or receiving from (i.e. appears after the feature), it is deemed to not be a valid communication path. Communication successors are defined as follows:

	A	B	C	D	E	F	G	H	I	J	L	N	O	P
A		I←				→								
B	I→							←						
C					I→						←	‖		
D					I→						←	‖		
E			I←	I←								‖		
F	←						I→							←
G						I←		I→	I→	I→				
H		→					I←							
I							I←				I→			
J							I←						→	
L			→	→					I←			I→		
N			‖	‖	‖						I←			
O											←			I→
P						→							I←	

Table 4.1: The Behavioral Matrix of the Running Example.

Definition 5 (Communication successor) *Let $\mathcal{L} \subseteq T^*$ be an event log. We define the* communication successor *relation $\lessdot_{\mathcal{L}} \subseteq T \times T$ by $A \lessdot_{\mathcal{L}} B$ iff $\mathfrak{R}(A) \neq \mathfrak{R}(B)$, $\sigma(i) = A$, and $\sigma(i+1) = B$ for some $\sigma \in \mathcal{L}$ and $1 \leqslant i < |\sigma|$.*

Considering the first three traces of the running example log:
$\{\langle A.F, G, I, L, D, E, N\rangle, \langle A, F, G, I, L, D, N, E\rangle, \langle A, F, G, I, L, N, D, E\rangle\}$
In this case, A always appears before F and are part of different modules, this makes it a valid communication path. N and E, while both being part of different modules both appear in front and behind each other and are not a correct communication path.

This communication is saved in a behavioral matrix, as for example shown in Illustration 4.1, where the matrix of the running example is depicted. This matrix is created around the communication behavioral profile, which is defined as follows:

Definition 6 (Communication behavioral profile) *Let $\mathcal{L} \subseteq T^*$ be an event log, and $\lessdot_{\mathcal{L}} \subseteq T \times T$ the corresponding communication successor relation.*

The communication behavioral profile *is the 3-tuple $(\rightarrow_c, \|_c, +_c)^{Com}_{\mathcal{L}}$ defined by:*

- *$A \rightarrow_c B$ iff $A \lessdot_{\mathcal{L}} B$ and $B \not\lessdot_{\mathcal{L}} A$;*
- *$A \|_c B$ iff both $A \lessdot_{\mathcal{L}} B$ and $B \lessdot_{\mathcal{L}} A$; and*
- *$A +_c B$ iff both $A \not\lessdot_{\mathcal{L}} B$ and $A \not\lessdot_{\mathcal{L}} B$.*

Within the figure, that means that the connections within a module are shown in gray and are annotated with either I← or I→. These are currently not used but might be used in the future when researching inner feature behavior. Connections between different modules are depicted as ← or →. Connections that appear as ‖ are coincidentally seen next to each other but are not a valid communication path.

4.3 Inner-feature behavior

At the lowest level, we propose the use of feature nets. Feature nets are a means to describe inner-feature behavior. The logs of which these nets are generated are called feature logs and are defined as follows:

Definition 7 (Feature log) *Let $\mathcal{L} \subseteq T^*$ be an event log, and let $F \in T$ be some feature. Let $(\rightarrow_c, \|_c, +_c)$ be the corresponding communication behavioral profile. The* feature log *$\mathcal{L}_F$ is defined by $\mathcal{L}_F = \{\sigma_{|C(F)} \mid \sigma \in \mathcal{L}, F \in \sigma\}$ where $C(F) = \{A \mid A \rightarrow_c F \vee A \rightarrow_c F\}$.*

Meaning that feature log only holds events of directly connected features outside of its own module that are direct predecessors or successors. For example, feature log L from the running example is as follows: $\{\langle D\rangle, \langle D\rangle, \langle C\rangle, \langle C\rangle$,etc..$\}$.

From these feature logs, feature nets are generated. These describe the internal decision making process of a feature and are defined as follows:

Definition 8 (Feature Net) *Let $\mathcal{L} \subseteq T^*$ be an event log, and let $F \in T$ be some feature. Let $(\rightarrow_c, \|_c, +_c)$ be the corresponding communication behavioral profile. The* Feature net *$\mathcal{N}_F$ is the OPN $\langle P, I, O, T, F, i, f\rangle$ defined by:*

- $P = \bar{P}$, $T = \bar{T}$, $i = [\bar{i}]$, $f = [\bar{f}]$;
- $I = \{p_{A-F} \mid A \rightarrow_c F\}$;
- $O = \{p_{F-A} \mid F \rightarrow_c A\}$;
- $F = \bar{F} \cup\{(t, p_{F-A}) \mid t \in T, \lambda(t) = A, F \rightarrow_c A\}$
  $\cup\{(p_{A-F}, t) \mid t \in T, \lambda(t) = A, A \rightarrow_c F)\}$.

where $\langle \bar{P}, \bar{T}, \bar{F}, \bar{i}, \bar{f}\rangle$ is the discovered workflow net.

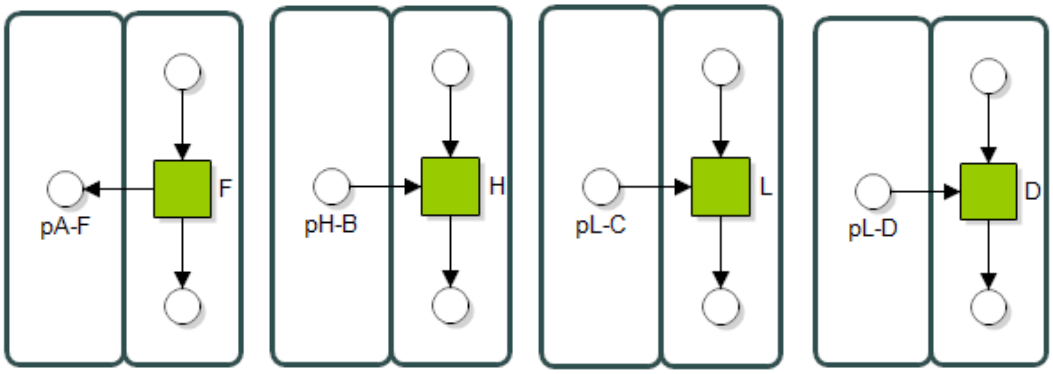
The resulting petri-net is a representation of the internal links of the feature. All features that partake in external communication have such nets. An overview of all feature nets of the running example is shown in Illustration 4.3. Feature A for example has an output to Feature F, called 'pA-F' while feature F has the same place but as an input.

4.4 Functional Architecture Model

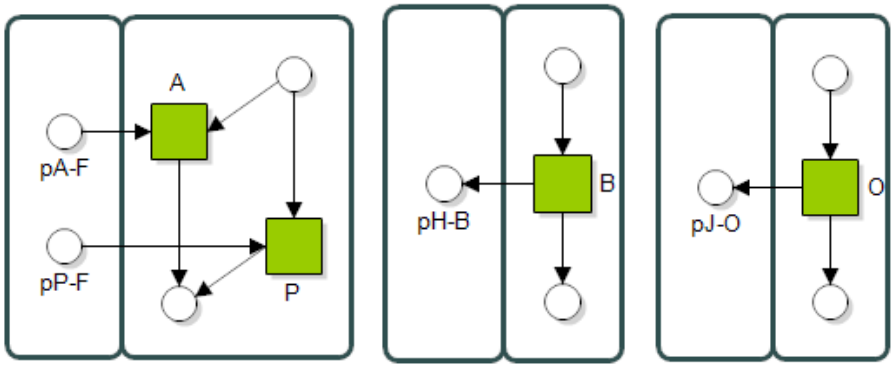
In order to generate a FAM, the communication behavioral profile, module- and feature-nets are combined. This combination is defined as follows:

Definition 9 (Generated FAM) *Let $\mathcal{L}$ be an event log, and $\{\rightarrow_c, \|_c, +_c\}$ be its communication behavioral profile. Its corresponding functional architecture model $\langle \mathcal{M}, \mathcal{C}, \mathcal{F}, h, m, \rightarrow\rangle$ is defined by:*

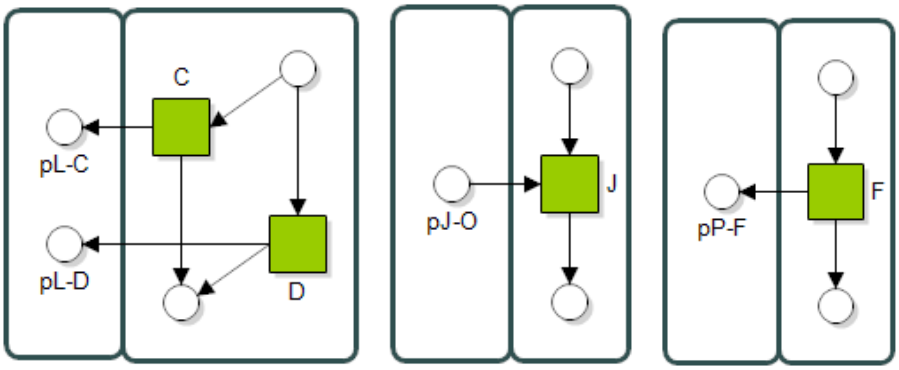
- $\mathcal{M} = \mathfrak{R}(\mathcal{L})$;
- $\mathcal{C} = \varnothing$;
- $\mathcal{F} = T$;



(a) Feature A (b) Feature B (c) Feature C (d) Feature D



(e) Feature F (f) Feature H (g) Feature J



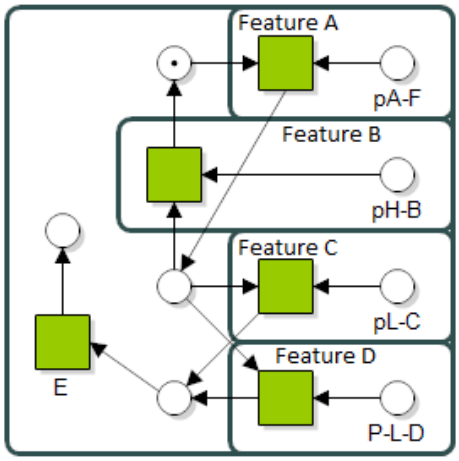
(h) Feature L (i) Feature O (j) Feature P

Figure 4.4: The Feature Nets of the Running Example

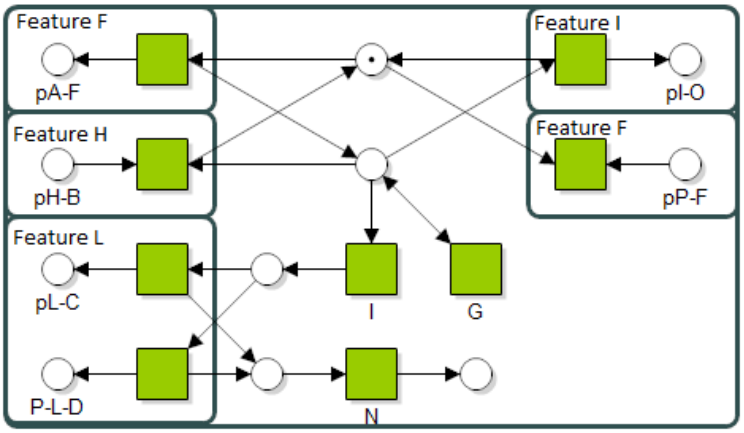
- $h = \varnothing$;

- $m = \mathfrak{R}$; *and*
- $\rightarrow = \{(A, x, B) \mid A \rightarrow_c B,\ and x \in \Lambda\ a\ fresh\ label\}$.

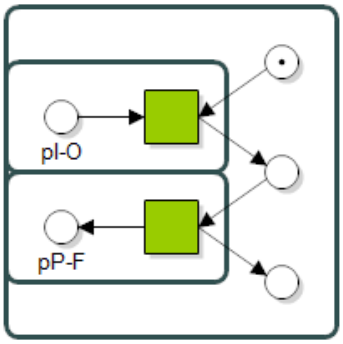
The completed FAM of the running example looks as depicted in Illustration 4.6. Chapter 5 shows how this process is completed using our tools. Whenever a feature connects to two different features, another feature is created to accommodate this extra connection. For example, F and L in Figure 4.5 both get an extra transition to connect to $\langle L, T\rangle$, $\langle L, D\rangle$ and $\langle P, F\rangle$, $\langle F, A\rangle$ respectively. This illustration shows all of the external and internal communication, which means that the three processes within the modules are preserved, while adding information on inter-modular communication and is called a Reference Net.



(a) Student Module



(b) University Module



(c) Reference Module

Figure 4.5: Reference Net of Running Example as Discovered by our Tools.

The FAM appears after omitting the internal communication. This only shows all external communication and is depicted in Illustration 4.6. Not every function has an external connection: Feature E in student, as well as G, I and N in University never directly communicate with an outside feature and as such do appear in the functional architecture, but not with a connection to the outside.

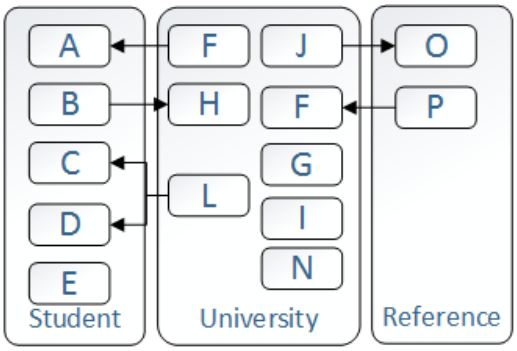


Figure 4.6: The Resulting FAM of the Running Example.

Chapter 5

Tool support

In this chapter we describe how our tooling works and how it generates the concepts described in the previous chapter. A Python program is constructed in order to automate this. In this chapter we only discuss the parts that are directly used to generate the concepts that are introduced in the last chapter. For a discussion on how the tool is made, including previous iterations, we wrote a summary in Appendix B.

5.1 Creating the behavioral matrix

The behavioral matrix is the first concept that is created, which is achieved by iterating over the event log and adding each module pair that connects to another. The loop starts by checking if the module is already known. If that is not the case, it is added to the 'rows' dictionary. Each entry has a name (the module name) and a dictionary as value (the known connections)

```
rows = {} # dict of dicts

for i, trace in enumerate(self.log):
    for prev, cur, next in zip([None]+trace[:-1],
            trace, trace[1:]+[None]):
        # Add current feature to rows if it doesn't exist yet
        if rows.has_key(cur['concept:name']) is False:
                rows[cur['concept:name']] = {}
                knownfeatures = rows[cur['concept:name']]
        # add event to module
        self.modules[cur['org:resource']].addevent(i, cur)
```

Next, two loops check if the previous, and respectively next feature belong to another module. If that is true, the value of that relationship gets changed in either a '>' if there is a match with the previous feature, or a '<' if the next feature matches. However, if the relationship already has a value, and it appears to be the opposite of what we want to set, the value instead gets set to '——' to indicate that both a feature has both appeared before and after it.

```
if prev['org:resource'] != cur['org:resource']:
```

```
if knownfeatures.has_key(prev['concept:name']) is False:
    knownfeatures[prev['concept:name']] = ">"
elif knownfeatures[prev['concept:name']] == "<":
    knownfeatures[prev['concept:name']] = "||"
```

At the end of the loop, 'rows' is filled with all relations and is converted into a Pandas [1] matrix that is the behavioral matrix and can be exported into excel if needed.

5.2 Feature logs

Both feature and module logs are generated simply by looping through the event log and only adding an event to the respectable log. The module logs are actually already generated during the matrix, by adding the events to the resource it belongs to. The feature nets are generated in the same fashion but with some extra steps as it only contains the externally connected features. The first part of the code loops through every feature and every trace.

```
for feature in self.features:
    for i, trace in enumerate(log):
        for prev, cur, next in zip([None]+trace[:-1], \
                trace, trace[1:]+[None]):
```

In every loop, it checks if either the event that is currently being investigated is the same feature. If this is the case it checks if the previous or next feature are considered an external connection. For this it uses the behavioral matrix. The only features that are added are those with a direct connection ('←' or '→' in the matrix) 'matrix.celliscausal()' is a function that returns whether or not there is a direct connection between two features. This test is also done for the next feature in the log.

```
if cur['concept:name'] == feature.name:
    if prev is not None:
        if matrix.celliscausal(feature.name, prev["concept:name"]) is True:
                feature.addevent(i, prev)
```

At this point, the process instances are as illustrated in Figure 5.1. A module instance contains a module log, and all features that it houses. These features contain a feature log of all direct connections of the feature.

5.3 Module and feature nets

Module and feature nets are generated outside of our own program, but we have automatized the steps to export the logs to ProM and let it generate petrinets for us. This is done in two steps: First we generate the logs in XES format, and after that we send every XES file individually to ProM.

[1] http://pandas.pydata.org/

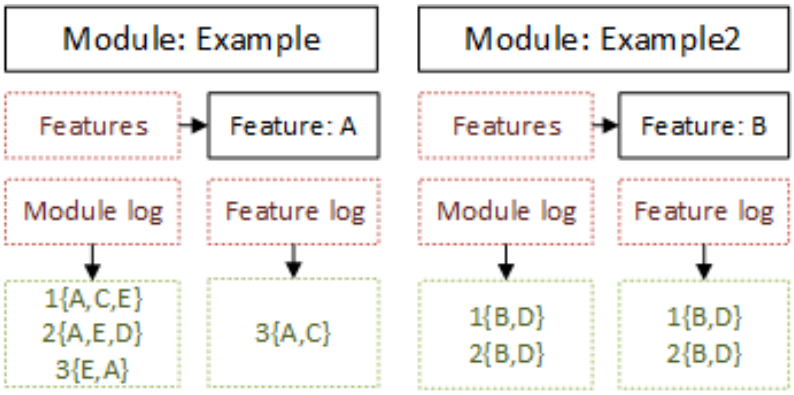


Figure 5.1: Logs in Modules and Features

Generate XES logs

The module logs as they are used within the program are loaded into First, Yggdrasil, the XES handling class. Next, it is written into the temporary directory, with the module as its name. This process is repeated for each feature within the module, those are saved in separate directories from modules.

```
modulelog = Yggdrasil(vertlog=module[1].log)
modulelog.writeXES("%s/xes/modules/%s.xes" % (tempdir, module[0]))
for feature in module[1].features:
    featurelog = Yggdrasil(vertlog=feature.log)
    if featurelog.log.__len__() > 0:
        featurelog.writeXES("%s/xes/features/%s.xes"
                % (tempdir, feature.name))
```

Generate ProM config files

For the actual process mining steps, we use ProM. Because ProM is a Java application and does not have a terminal interface our model depends on modifying configuration files. Every log needs its own configuration file and needs to be called independently, making this a time consuming step.

```
script = open('../assets/script_indm_template.txt', "r")
newfile = []
for line in script:
    line = line.replace("myLog.xes", "%s/xes/%s/%s.xes"
            % (tempdir, logtype, name))
    line = line.replace("filename.pnml", "%s/pnml/%s/%s.pnml"
            % (tempdir, logtype, name))
    newfile.append(line)

newfileoutput = open('../assets/currentscript.txt', "w")
```

```
newfileoutput.writelines(newfile)
newfileoutput.close()
if dryrun is False:
    subprocess.call([prompath, '-f', \
            '../assets/currentscript.txt'])
```

The template used for this script can be found in Appendix A.1. The last line calls the prom cli interface and presents the customized configuration file to the program. While ProM is running our program is put on hold. In the end, ProM saves a pnml file in the temporary folder and our program continues.

5.4 Build reference net

The next step is to import the generated petrinets and combine them into the reference net. First, all modules are combined into one net. This is done by opening the module net PNML files (which are really just XML files), and combine the separate files into one. For this XML element attributes need to be renamed as every XML element has to have a unique ID. This is done by the function ReferenceNet.renamenodes() and will not be discussed in further detail.

As Feature nets need to be inserted into an already exported module net, they follow a more complex route. In Illustration 5.2 a fictitious module is displayed. In order to insert the inner-processes of module B, the transition in red needs to be deleted.

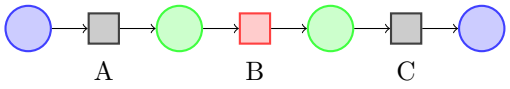


Figure 5.2: A module Net Example.

This is easily done by searching for the transition with the name of the Module, after all, we assume modules are uniquely named. This transition is than deleted, along with its receiving and sending arcs. Figure 5.2 shows this process. The id's of the green places are remembered, so that the feature net can be parsed into the module net at these places. The following code searches for

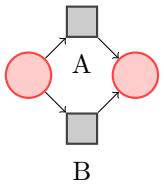


Figure 5.3: A feature net example.

the feature, remembers the ID and deletes it.

```
if trans[0][0].text == featurename:
    oldtrans = trans.attrib['id']
    remove.append(trans)
```

After this, the incoming and outgoing arcs are deleted in a similar fashion. The following code searches for the incoming arc and searches for the source and target places, respectively (which are shown in green in Illustration 5.2).

```
if arc.attrib['target'] == oldtrans:
    # find the place connected to this arc
    src = [e for e in page.iter('place') \
           if e.attrib['id'] == arc.attrib['source']]
    srcname = arc.attrib['id']
    remove.append(arc)
```

The following part relies on ProM 6.4 and possibly the inductive miner algorithm. Searching for the old end place inside of a feature log (right circle in Illustration 5.3) is done as follows:

```
for place in root.iter('place'): # iter through feature net
    if not [a for a in root.iter('arc') \
            if a.attrib['source'] == place.attrib['id']]:
        oldend = place
        remove.append(oldend)
```

This query assumes that an end place has no outgoing arcs. The query basically states that "if no arc has the place ID as source, this is the end place (drain)" unfortunately this rhetoric does not hold true for a start place (source), as a loop will lead to a source with both one (or multiple) incoming and outgoing arcs. In this case, the program looks for the place with the text 'source'. This is however a ProM specific field and may not be used by other tools.

Finally, the arcs of the feature net are connected to the places in the main net.

```
if arc.attrib['source'] == oldstart.attrib['id']:
    arc.attrib['source'] = src[0].attrib['id']
```

This is done four times; for both incoming and outgoing arcs to the source and target.

5.5 Creating the FAM

The functional architecture model is not automatically generated as there are no tools that automatically generate them. Rather, this is done manually. In our case, we use the reference net by omitting all internal module communication and only keep the external behavior as can be seen in the previous chapter.

Chapter 6

Conclusion and Future Work

In this chapter, we first conclude our research followed by a discussion on the limitations of the research. This chapter is concluded by a discussion on future work.

6.1 Conclusion

Within this thesis three phases of research are discussed with the goal to answer the question of **how can processes in a collection of (inter-) operating systems, known or unknown, be identified, analyzed and utilized, using current methods, techniques and notations as a basis?** In order to answer this question, we answered the following questions.

What different methods, techniques and notations exist to model behavior? This question is discussed in Chapter 3, where we discussed notations such as Petri nets and the business process modeling notation, techniques such as process mining algorithms and overarching methods such as BPM, automata theory and multi-agent systems. We categorized these concepts as either architectures, (horizontal) behavior or vertical behavior. Little formal notations exist on the highest level of abstraction. The functional architecture model is one of the only formal models and is used as a design goal for us. In contrast, the field of process modeling offers a lot of (formal) notations. Organizations often use the EDI standard as a means to set up automatically communicating systems between enterprises while BPMN is often used by process engineers. While BPMN comes with a set of design rules, it also offers great flexibility as these rules are not always enforced. This does not hold for Petri nets and state machines, which are both at the basis of a lot of different modeling techniques. Petri nets are – among other usages – utilized as a behavioral modeling technique. It is a formalized language and as such, is used in a wide array of applications, from process simulation and validation to graphical representations of processes. Automata are used for as many applications, exist in many forms and are also of a formal nature. However, where petri-nets are used to describe horizontal behavior, automata are used to research and describe horizontal behavior.

What difficulties currently exist regarding behavioral modeling? While answering the previous question, we identified several difficulties for each of the three categories. Within architectures, it is difficult to distinguish between different levels of communication, as modules may have other child modules. Another difficulty is to keep the size of the architecture under control. This is

especially important when discovering a functional architecture from event data as the level of abstraction can differ greatly. For (horizontal) processes it is important to keep the control flow intact while discovering vertical behavior. This is a challenge, because changing the level of abstraction can lead to very long processes. Finally, vertical behavior is perhaps the biggest challenge as this is not easily accessible in event logs and ways to identify these processes have not been researched in detail. We solved this by discovering events that describe the behavior of a certain feature, as described in the following paragraph.

How can current techniques be combined to improve on identified difficulties? The challenge of managing large processes is partly solved by identifying modules and features, as we discussed in Chapter 4. Here we use a behavioral matrix to identify communication between modules. This modularity enables process miners to divide one control flow into one flow for each module and is discussed in Section 4.2 on Page 25. Indeed, by separating modules and identifying feature behavior, process analysts are able to 'zoom in' as processes that are otherwise too complex to analyze using process mining techniques can now be processed in smaller chunks while maintaining the overall structure.

How does inter-feature communication need to be defined in order to describe behavior, and how does it relate to horizontal behavior? In order to structure information on the most detailed level, we use feature logs and -nets. These differ from their module counterparts and are discussed in Section 4.3 on Page 29. Where module nets are based on a collection of events that are connected with the module (the event is performed by a feature that belongs to a module), feature nets contain all events that are an in- or output of the feature (by it being identified as a valid connection in the behavioral matrix). Indeed, while describing the behavior of a particular feature, a feature net does not contain events of the feature itself. This leads to a log that only contains true vertical behavior, rather than only a slice of a horizontal process.

How is communication captured in Functional Architectures? When feature nets are inserted in a functional architecture model they allow for the FAM to be formed. With the internal feature nets retaining the vertical behavior and the module nets describing the horizontal process, a combination of the two forms the reference net. This construct can then be transformed into a FAM by extracting the horizontal behavior. This behavior is different from the initial module nets in that additional communication which is found in the communication behavioral profile is included. This is described in Section 4.4 on Page 29. Because a reference net describes both horizontal as vertical behavior, it can become a very complex model indeed. Although the reference net will be better readable than one long horizontal process, it can not be considered a true functional architecture. By omitting both the horizontal and vertical information while maintaining the information on communication between features (as maintained by the behavioral matrix) we end up with a functional architecture that describes which features connect, and in what order. This in turn enables a software architect to describe which systems are connected.

What steps are necessary to transform event logs into a FAM? In a proof of concept we describe the steps and their order to transform event logs into a functional architecture. This is described in Chapter 5 and discusses the complete process. First, the behavioral matrix is created, followed by module and feature logs. These are mined using existing algorithms into feature and

Case	Trace
1	A, F, G, I, L, C, E, N
2	A, F, G, I, L, C, N, E
3	A, F, G, I, L, N, C, E

Table 6.1: Incomplete Log Example

module nets. These combined become the reference net which is manually transformed into the functional architecture model.

6.2 Limitations

During the research, we found a couple of limitations and oddities that warrant a discussion. First we discuss limitations in our approach and then we propose goals for future research in the next chapter. There are several limitations in our approach that we have not been able to test, solve or describe in more detail. We propose the following two items as future work regarding model adaptation.

Difference in algorithms We have not been able to test how different algorithms behave when used to generate a reference model. What we do know is that algorithms need to be sound. This is because unsound nets can differ between modules and features (i.e., the behavioral matrix can suggest other ways of communication than the nets do), allowing for differences when using the communication behavioral profiles. However, we do not know how our tools react to other sound algorithms. As discussed in Chapter 4, our tool substitutes elements of different petri nets in order to build the reference net. This process is build around the output of the inductive miner and makes some assumptions on how the PNML file is constructed. Other algorithms may follow different design guidelines, making them incompatible with the current tools.

Concepts belonging to multiple resources Our tools assume a concept (feature) is only part of one resource (module). This makes sense as a feature describes an ability of a module. However, during tests performed on real-life data we did see cases where this was not true. Previously this might not have been a huge problem as algorithms only look at the sequence of features in a trace and not the modules involved. If existing logs indeed have data where features are part of multiple modules, it could lead to problems with our tools.

Incomplete logs Because of the way the behavioral matrix is populated, communication paths need to appear in every possible combination. For example, Table 6.1 shows the shortest acceptance path from the running example. If trace 3 would be omitted, 'N' would never appear before 'C', making it a valid connection rather than a $\|$ relation as depicted in the behavioral matrix (Table 4.1). This could lead to false positives because an invalid connection is only found after it is first flagged as a valid one. For instance, if only trace one and two are processed, we know that A $\rightarrow$ F, F $I \rightarrow$ G, G $I \rightarrow$ I, I $I \rightarrow$ L, L $\rightarrow$ C, C $I \rightarrow$ E, C $\rightarrow$ N and E $\|$ N. Indeed, at this point we assume C to have a direct successor in N. It is only when the third trace is processed that we know that this is in fact a false connection. For future work we propose ways to assess log completeness and flag false positives.

6.3 Future work

As discussed in the previous section, there are a couple of limitations and hurdles still to tackle. This research would benefit from an in-depth study on which algorithms are usable in this technique. Log compatibility is another subject that needs to be researched.

There are also a couple of steps within our process that are now done manually. The most complex one being the transformation from combined module and feature nets into a true reference net, and from reference net to FAM.

Apart from limitations, we have also identified new opportunities. We hope that this research will open the door in making process mining more modular. While some mining algorithms already have a modular approach, the generated models are not, and often end up being a 'spaghetti' process. The FAM as well as module and feature nets help making processes more modular and readable, but a formal approach is still absent.

Finally there are fields of study that could benefit from discovering vertical behavior. Multi-agent systems and automata have been used to describe this behavior but do so in a different way than our feature nets. An effort to 'translate' discovered models into existing models and notations would be beneficial indeed.

Appendix A

Used scripts

This chapter contains some of the scripts and program code used during the research.

A.1 Prom CLI template

```
System.out.println("Loading log");
log = open_xes_log_file("myLog.xes");

System.out.println("Mining model");

System.out.println("Setting classifier");
classifier = basic_event_classifier();

System.out.println("Creating Inductive miner settings");
org.processmining.plugins.InductiveMiner.mining.MiningParametersEKS parameters = new org.processmining.

net = mine_petri_net_with_inductive_miner_with_parameters(log, parameters);

File net_file = new File("filename.pnml");

pnml_export_petri_net_(net[0], net_file);

System.out.println("done.");
System.exit(0)
```

Appendix B

Prototyping

In this chapter the prototyping phases are discussed together with the constructed and adopted concepts and formalizations used in the prototype. First general concepts are discussed followed by the prototype phases and concepts that are specifically designed in their respective phases.

B.1 Prototype Iterations

The first step is to grasp the problem at hand. This is done by abstracting away from the problem seen at first hand and 'deconstructing' it into workable pieces. Next up is design planning. note that this plan does not have to remedy the entire problem. Rather, it focuses on a small, specific subset of the problem. This is to be expected; within design science, implementing and assessing new measures is part of the research process. In each stage, there is a solution that is tested in the problem environment to see how well this works.

The phases themselves can be roughly bundled into four stages, the early phase on its own has as goal to explore current technologies. The two 'pair' phases introduce a new approach and finally the last phase revises this approach into the final form.

B.2 Early Phase

During the first phase, which focused on setting up the test infrastructure, several modules are created that mostly found their usage in later phases. The problem we investigate is as follows: "How to set-up a test environment for horizontal process mining"

Analysis

As discussed in Chapter 3, there are several current solutions that aid our research. At the start of the prototyping process, this involves frameworks such as ProM, libraries and scripts like XES and SNAKES, and programming languages and environments. Most of these solutions turned out to be very poorly documented, which impact the usefulness greatly. Current mining techniques diver greatly from our envisioned artifact, as such, only the SNA method is further analyzed for possible ways to adapt it.

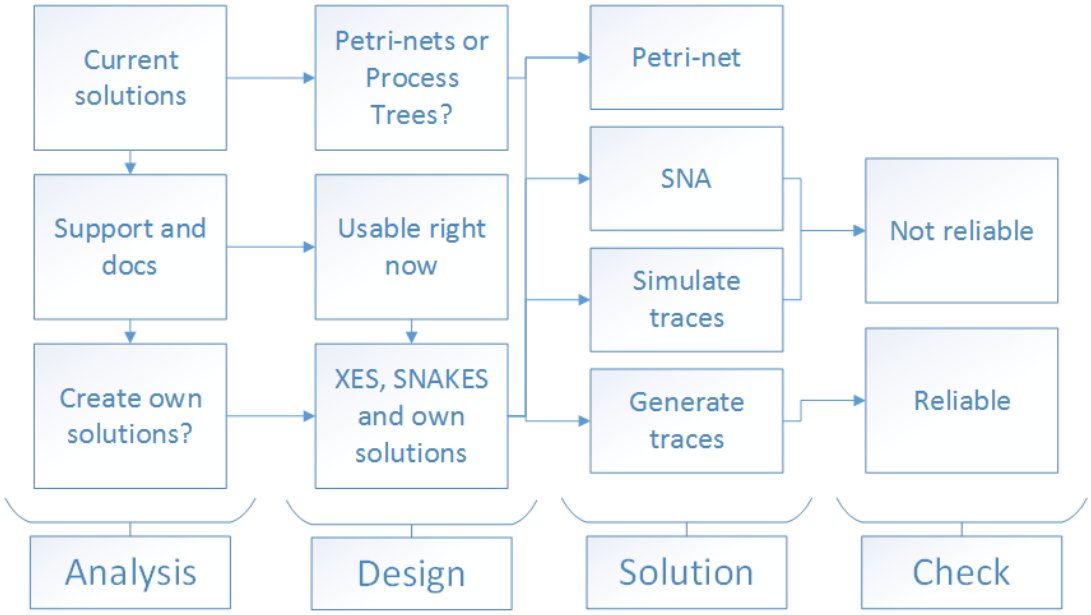


Figure B.1: Flowchart of First Iteration

Design

For the first tests, ProM is not used in the automated process, rather, it serves as a confirmation tool to gauge the effectivity of our tool. The before-mentioned libraries form the center of a couple of modules, which are now discussed in further detail. In order to quickly create test logs within the .XES format, a modified Python library[1]is used, together with a simple series of for-loops to generate pseudo-random logs with several modifiable variables. This is done using the script *"Yggdrasil"* named after the mythical tree from Norse mythology. Among other theme's, it is supposed to be the source of the runic alphabet; the god Odin, or in Germanic regions *"Wodan"* retrieved these runes and handed them to the pagan tribes. This metaphor works because the logs are a collection of symbols which once translated, form a usable alphabet. This script is based upon the SNAKES library[2], which allows for generating petri-nets and the inspection of process behavior. As there are no libraries available to enable these actions using process trees, and because Petri Nets are easier to read and to combine into open petri-nets, we use the latter.

Test use cases are are saved in *"input.py"*, with which a test log is generated that is used to conduct our first batch of tests. Additionally, XLSX files can serve as a convenient way to insert new traces directly without importing and exporting to PNML. Within our process we also use other tools: *Yasper* to graphically model and view (open) petri-nets and export them to *PNML*, and *ProM* to run analyses on generated event logs.

[1]https://github.com/jsumrall/xes
[2]https://pypi.python.org/pypi/SNAKES/0.9.17

solution

With this structure, simulation and testing of petri-nets is done without the help of ProM, which indeed raises many concerns. For one, by not using proven algorithms but rather our own implementation, correct outcomes were not guaranteed. Furthermore, during this phase, the plan was to work with permutations, i.e., shuffle traces so all possible combinations are tested. Along with a non-functioning set-up for testing petri-net correctness, these parts are replaced in later iterations.

Furthermore, no communication between features could be analyzed with this setup. However, this phase allowed us to generate test data and petri-nets, much of which has been used in later stages. Lastly, the SNA approach got shelved for the time being and focus on communication between modules as found in direct succession within traces is seen as the right way to go.

While not using ProM and the aim to randomize traces proved that this first try did not meet our overall goals, its methods are used within the next iteration.

B.3 Communication pairs without renaming

The second phase explores the usage of communication pairs to map communication between modules. During this phase, ProM is integrated via command line interface, broadening the amount of usable algorithms. From this point on, the Inductive Miner is used because it is available via CLI and is considered a reliable algorithm. The ability to generate and split up traces as setup in iteration 1 is again used in this cycle. Figure B.2 depicts the flow of the second iteration.

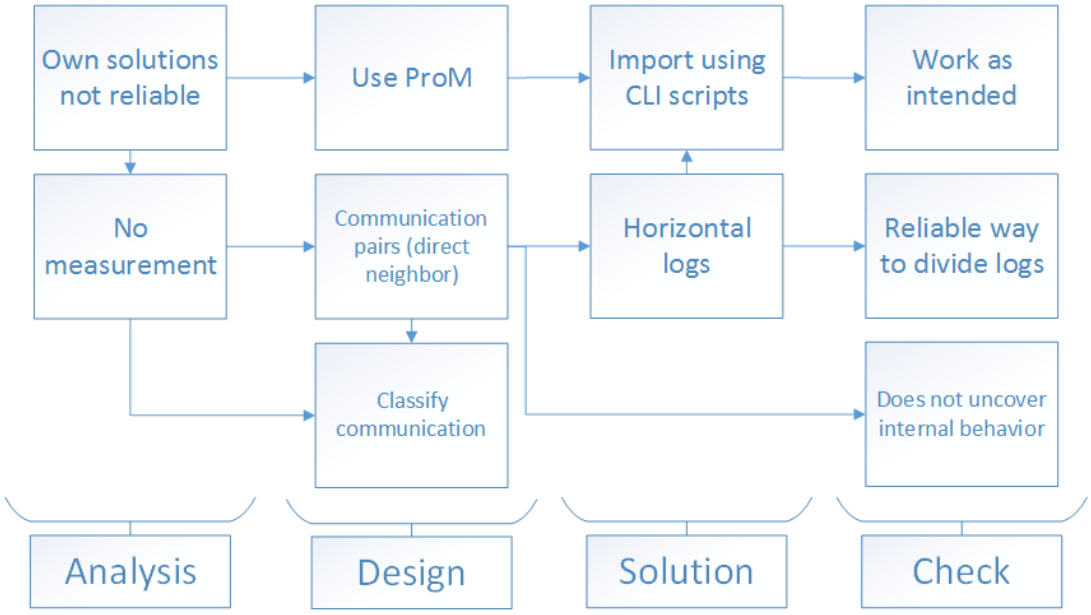


Figure B.2: Flowchart of Second Iteration

From this point on, the approach is formalized as discussed in Chapter **??**, and communication pairs are theorized to identify and describe horizontal behavior, i.e., communication between pairs.

Analysis

The last iteration did not work because of several reasons; not only were arbitrary methods used, there was also no measurement devices in place. I.e., even if the method proved to be valid, there is no way to assess its outcome. Metrics are needed to categorize communicative behavior between modules. To describe connections between modules, connection pairs are used.

Pairing

Behavior between modules is always a pair $\langle output, input \rangle$, where output and input are distinct features from two modules(e.g., $< A, B >$, $< Shipping, Receiving >$, etc). Pairs can be categorized depending on the behavior of other pairs.

$\mathcal{P}(x, y)$ *a pair* $< x, y >$ *consisting of two unique features, of two different modules that are connected, as:*

$\forall m \in \mathcal{M} \forall e_1, e_2 \in \mathcal{E} : [contains(e_1, m) \neg contains(e_2, m) \rightarrow pair(e_1, e_2)]$

- A pair is considered 'strong' if x $\mathcal{P}$ is unique for y and if y is unique for x
- A pair is considered to have a 'weak input' if x in $\mathcal{P}$ exists in other pairs with the same y
- A pair is considered to have a 'weak output' if y in $\mathcal{P}$ exists in other pairs with the same x
- A pair is considered to be 'weak' if y in $\mathcal{P}$ exists in other pairs with the same x, and y exists for x.

An overview of these types of behavior can be found in Figure B.3. The transitions (rectangles) symbolize the transition from and to a module, the place (circle) is an interface that connects the two modules (or petrinets).

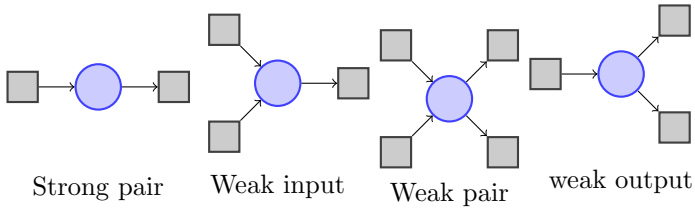


Figure B.3: Different Sorts of Horizontal (Pairwise) Behavior

Communication pairs are used in conjunction with horizontal logs. Each trace is separated into chunks that specifically belong to one module. This leads to as many logs as modules:
$L = \{< A^1, B^1, C^1, D^2, E^2, F^1 >, < A^1, B^1, C^1, E^2, F^1 >, < A^1, B^1, C^1, D^2, C^1, E^2, F^1 >\}$
$L^1 = \{< A^1, B^1, C^1, F^1 >, < A^1, B^1, Ce^1, F^1 >, < A^1, B^1, C^1, C^1, F^1 >\}$
$L^2 = \{D^2, E^2 >, < E^2 >, < D^2, E^2 >\}$ Horizontal logs are designed to describe inner-modular behavior, while communication pairs form inter-modular connections.

design and solution

The process is restructured: ProM using induction miner, is used for all process mining tasks, while communication classification is deemed to be the main deliverable of this iteration. Most of the process around the mining process is re-purposed by generating module logs and sending these to ProM in turn: Only import from excel is used, after which the log is chopped into a part for every module. This log is fed back into Yggdrasil and exported as a .XES file which is imported into prom. The PNML (a file type based on XML) files are than connected by editing the file and adding pair connection. B.4 shows a flowchart of this process.

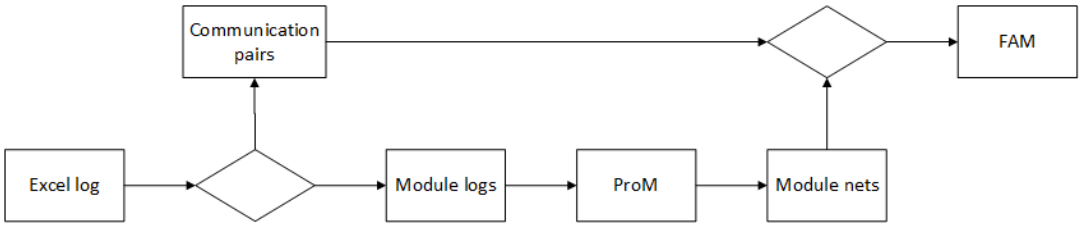


Figure B.4: Design of Second Iteration

check

While this setup did introduce a means to combine horizontal nets, and the mining process itself was reliable, a few problems became apparent. The most obvious effect of this approach is the loss of detail. e.g., module A sometimes communicates to B, and sometimes to C. This behavior would get mapped, but the explanation why would disappear. Furthermore, and even more inconvenient, while the pairs did reveal communication between modules, nothing about the inner-modular processes became visible. For instance, given two traces:
Trace 1 = $\{A, B, C, D, E\}$ Trace 2 = $\{B, C, E\}$.
In this simplified case, a message would always be passed from 'A' to 'E' in alphabetical order, unless 'A' is omitted, in that case, 'C' send directly to 'E'. However, this information is lost with the solutions developed in the second iteration. Rather, we would merely know that 'C' sometimes sends to 'D', and sometimes to 'E'.

B.4 Communication pairs with renaming

The third phase added renaming of features in order to identify specific communication between modules. Technically, this phase differed very little from the previous phase, the only big change was to rename events to reflect their incoming and outgoing connections. once again, a flowchart of this stage can be found in Illustration B.5

Analysis and Design

Because the process of the second iteration delivers a reliable work-flow, but the measures are wrong, improvement means adding an additional step. Communication pairs need to be altered in such a way that one can connect behavior in different situations to changes in pair behavior.

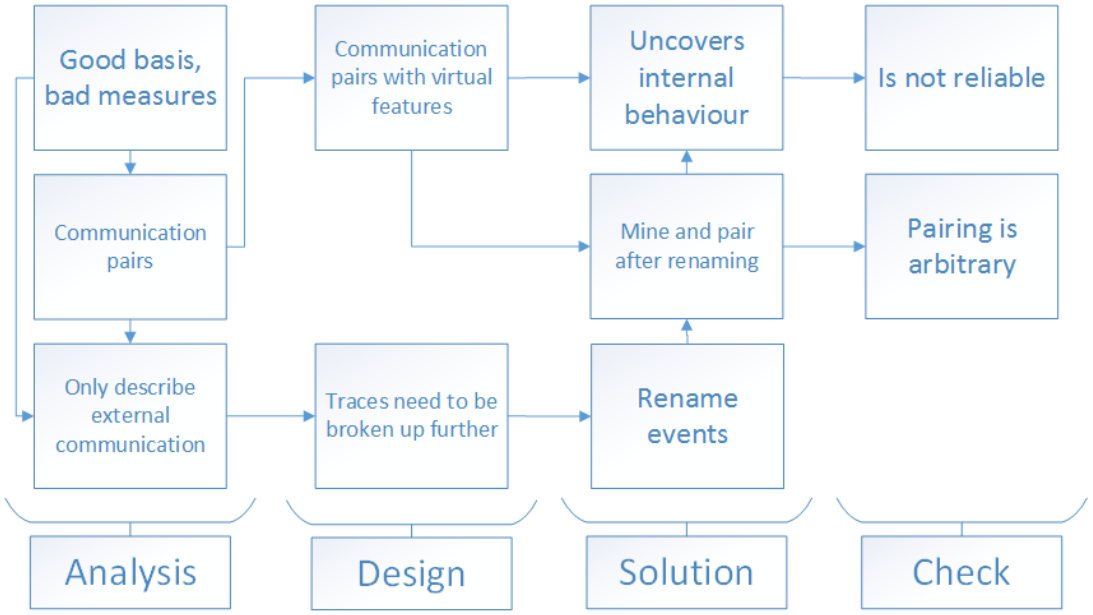


Figure B.5: Flowchart of Third Iteration

Only a small change in infrastructure is needed. Before logs are send to ProM, the events are renamed in such a way that incoming or outgoing communication yields a new virtual feature. Using this approach, the mining algorithm still mines the same petri-net, but with added transitions where behavior differs.

solution

This approach solved the problems of the previous iteration. e.g., B.6 shows a log is generated using the new solution. As can be seen, different virtual features are generated where different behavior can be found.

Renaming events is done as follows:

- $\forall e \in L : e[-1] \rightarrow \notin module$; add previous feature name to the left in lowercase
- $\forall e \in L : e[+1] \rightarrow \notin module$; add next feature name to the right in lowercase

This renames all events to represent an incoming or outgoing dependency while maintaining inner-modular behavior. We use lowercase to indicate the connection, while the position relative to the capital letter denotes in- or output. The log is thus transformed as follows: L = $\{< A^1, B^1, Cd^1, cDe^2, dE^2, F^1 >, < A^1, B^1, Ce^1, Ef^2, F^1 >, < A^1, B^1, Cd^1, cDc^2, dCe^1, cEf^2, F^1 >\}$

While the former stage also included log connection, it is first successfully used in this phase, as previously inner-modular behavior stayed hidden. Connecting modules is done in the following

manner:

for each $< x, y >$: there is a feature x in module 'a' and a feature y in module 'b', but only if $x \neq y$, and $a \neq b$

Features are connected using interfaces, as shown in Figure **??**. Strong connections always consist of one interface with an in- and output. Weak interfaces however, can contain multiple in- or outputs. To identify different forms of weakness, we test for behavior as discussed in B.3, after which either a new interface is created, or a new connection arc is drawn towards the existing interface.

After connecting these petri-nets, the end result can be depicted in multiple ways. A traditional petri-net can be used, but does not allow separate nets to be viewed as distinct entities. An open petri-net as depicted in Illustration B.6 provides a better overview.

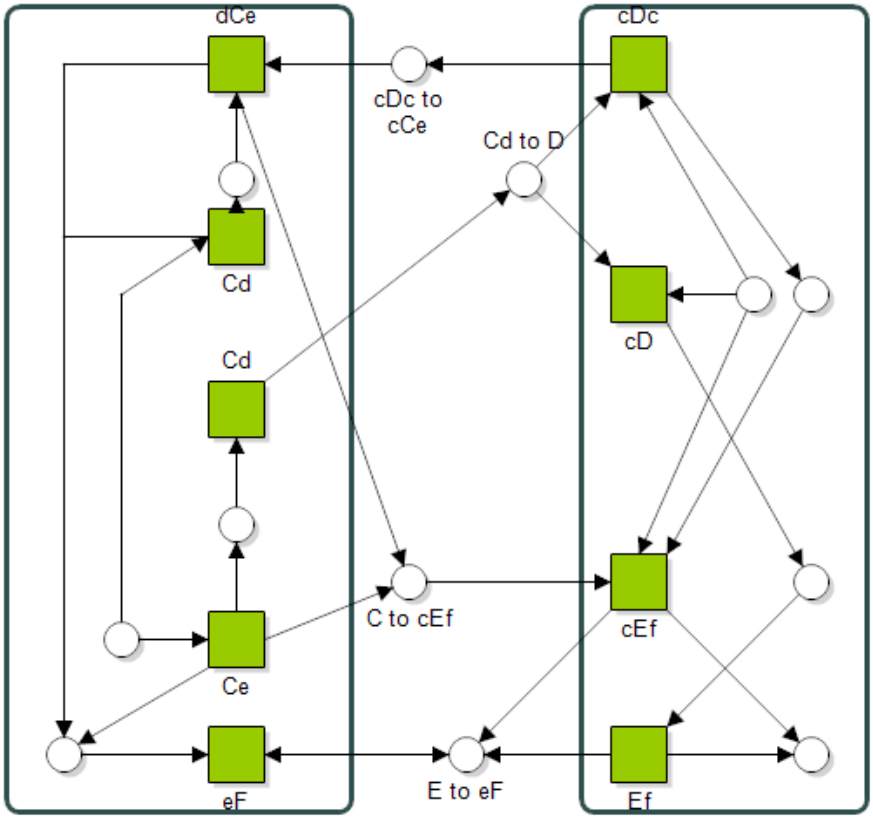


Figure B.6: Communication Pairs

check

While this iteration was able to generate petri-nets and connection pair lists that are able to depict different sorts of process model behavior, the pair-central approach does have some disadvantages. The most troubling on is that pair generation is not always as predictable. This happens because different interpretations existed, a pair might be strong when only virtual features are taken into account, but prove to be weak when renaming is done slightly differently. i.e., a standardized approach is needed.

B.5 Behavioral profile matrix

The final iteration combined and expanded upon the two earlier solutions. The pair system got replaced by a more sophisticated system based on the Social Network Analysis algorithm of which an alpha miner generated matrix is used to identify connections between modules. This means that renaming is not needed anymore and internal behavior can be found by also generating the feature nets from this same matrix.

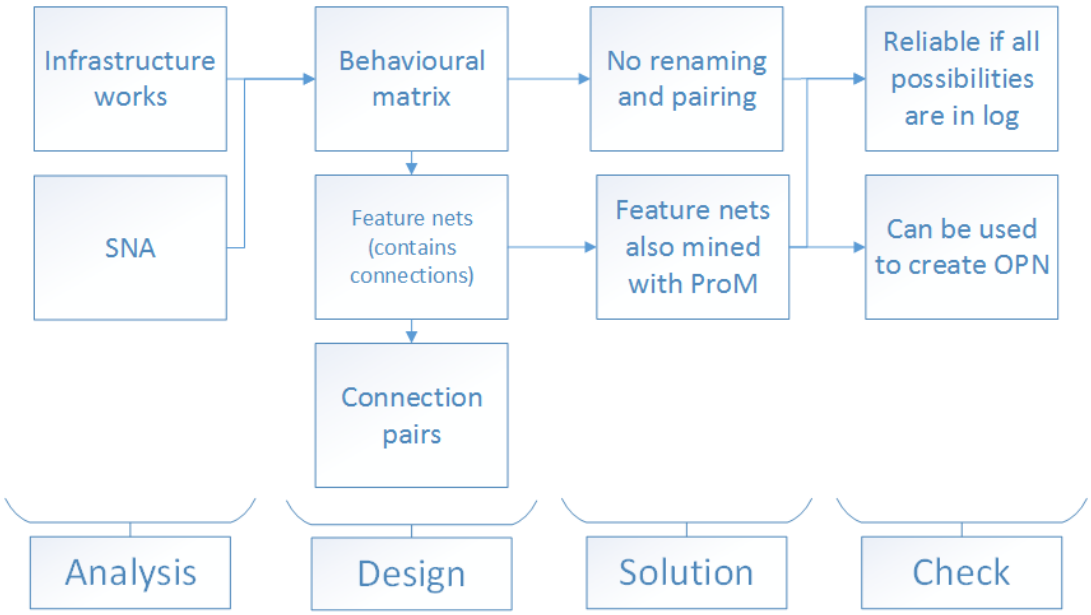


Figure B.7: Flowchart of the Last Iteration

Analysis

There are two problems with the previous solutions that need to be resolved: the ambiguity from renaming events – although this seemed to work, there was no standardized way that let yielded reliable results. Furthermore, pairs still did not work even after renaming. This is due to only

minding the previous and next feature in a trace, rather than the relative positions inside the whole log.

To fix both issues, the Social Network Analysis (Van Der Aalst et al., 2005) is used with a slightly different implementation compared to the original. The algorithm looks for (possible) causality by counting the amount of times a module is succeeded by another module. Normally, it is used to count the amounts in which modules communicate with each other. However, in our new model we use the Handover-of-Work approach on features.

For example, the SNA handover of work approach functions as follows:
$< A, B, A, B, B >: A > B = 2, B > A = 1, B > B = 1$
This way, a matrix is build with the count of each modules' communication to another. However, our method merely checks for causality rather than keeping score. Our implementation works as follows: $/< A, B, C, A >, < A, B, A >, < A, C, B > / : A > B, A > C, B||C$
'A' always appears before 'B', but 'B' never before 'a', thus $A > B$. 'b' and 'c' show up before each others, so $A||C$. Every connection that gets flagged as either ' $<'$ or ' $>'$ is a valid connection from feature to feature.

Design

The resulting design if very similar to the previous iteration with a few notable exceptions. This time – instead of creating a pair list, the matrix is generated. From this matrix, the feature and module nets are generated which are all fed into ProM. This approach ensures models are generated with tools that are known to work. Illustration B.8 shows a diagram of this iteration.

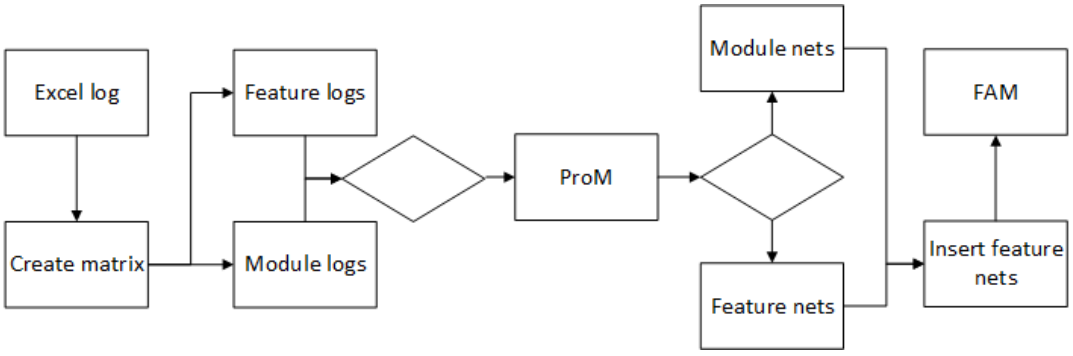


Figure B.8: Design of Last Iteration

Appendix C

Published paper

Discovery of Functional Architectures From Event Logs

Jan Martijn E.M. van der Werf and Erwin Kaats

Department of Information and Computing Science
Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
j.m.e.m.vanderwerf@uu.nl, e.j.kaats@students.uu.nl

Abstract. The functional architecture focuses on decomposing functionality into modules that offer certain features. These features require interactions in order to complete their functionality. However, functional architectures typically only focus on the static aspects of the system design. Additional modeling techniques, such as message sequence charts are often used in the early phases of software design to indicate how the software should behave.
In this paper we investigate the use of process discovery techniques to discover from these scenarios the internal behavior of individual components. Based on event logs, this paper presents an approach (1) to derive the information flows between features, (2) identify the internal behavior of features, and (3) to discover the order between features within a module. The approach results in a sound workflow model for each module. We illustrate the approach using a running example of a payment system.

1 Introduction

One of the principle tasks of a software architect is to design a software system [17], i.e., to organize the software elements the system is composed of in sets of structures, to allow reasoning about the system [4]. Many different Architectural Description Languages (ADLs) exist to document software architecture. However, due to the large competitive market in the software product industry, architecture is often neglected in software product organizations [14]. Hence, not many ADLs are used in practice. As experienced in [14], in software product organizations, architects rather use informal architectural models as an instrument of communication and discussion.

An important aspect of software architecture is the functionality it offers. To decompose and specify the functionality of software, the authors of [6] introduced the Functional Architecture Model (FAM), which offers the desired modeling technique used by many software architects in software product organizations [14]. The FAM separates the functionality into so-called features that are offered by the different modules the system is decomposed into.

Features interact with other features via information flows to offer their functionality. However, FAM only offers a static view on this interaction, i.e., the information flow only shows possible interactions, but imposes no order on or dependencies between these flows. Thus, to show how functionality is offered by the system, the architect requires additional models. One way is to define scenarios on top of the models, in which the architect can specify which features interact in which order. These scenario then result in event logs, that can be analyzed using process mining techniques [2]. Another source for discovering the possible interactions between features is the use of system execution data [21], mapping events to the (partial) execution of features. In this way, execution data can be used to reconstruct a software architecture.

In software product organizations, time to market is often a more important priority than having a properly documented software architecture. Consequently, architecture documentation is often outdated or even missing [9]. Therefore, discovering architectural models help such organizations in maintaining their software products. In this paper, we investigate the possibility to use process mining techniques to discover, the functional architecture of the system from an event log. We thereby focus on three basic questions on the functional architecture:

1. Which features interact?
2. What is the internal behavior of features?
3. What is the order in which features are executed within a module?

The first question focuses on the discovery of information flows: given an event log, is it possible to derive which features interact? Next, we investigate whether it is possible to derive the internal behavior of features based on event logs. In other words, we focus on the question how does a feature use its information flows to complete its functionality. The last question deals with the high-level view of the functional architecture. To execute the system's functionality, the features within a module are called in a certain order. Can process discovery techniques be used to discover these orders?

The remainder of this paper is structured as follows. To illustrate the approach, Sec. 2 presents a running example which we will use throughout the paper. Next, Sec. 3 presents the basic notions used in the paper. Section 4 introduces the functional architecture model in more detail, after which in Sec. 5 we will focus on solving the three questions posed in the introduction. Section 6 concludes the paper.

2 Running Example

As an running example, consider the Payment System as introduced in [10]. The system consists of three modules, *Debtor*, *Payment* and *Creditor*. The payment module serves as an intermediate between the Debtor and the Creditor. An example of such a payment module is the european SEPA standard. The payment module initiates a transaction, which the debtor needs to accept. If the debtor accepts, the payment is continued, and the creditor is contacted to start the

transaction. If for some reason the creditor rejects the transaction, the debtor is notified, and the transaction is terminated. Similarly, if the creditor accepts, the payment is passed to the debtor, and finally, the creditor receives the final payment information.

As the software evolved into the current system, no precise model exists that specifies the behavior of this system. The system only recorded the order in which the different features of the modules have been called in an event log, as shown in Tbl. 1. Each pair in the table represents the feature and the module to which that feature belongs. For readability, the features and modules are abbreviated in this event log.

The system is decomposed into three modules: the *Debtor* module (X), the *Payment* (Y) module, and the *Creditor* (Z). Based on the event log, the software architect finds the following features:

- Receive transaction request (A);
- Reject transaction (B);
- Accept transaction (C);
- Cancel transaction (D);
- Initiate payment (E);
- Send payment details (F);
- Archive transaction request (G).
- Send transaction request (H);
- Reject transaction request (I);
- Initiate creditor (J);
- Cancel transaction (K);
- Initiate payment (O);
- Handle payment (M);
- Archive transaction (N).
- Start transaction (Q);
- Handle transaction (S).

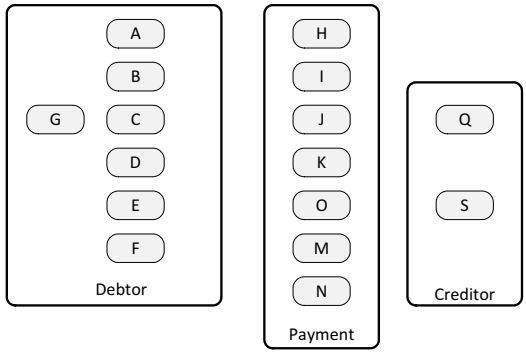


Fig. 1. Initial functional architecture model of the running example

Case	Trace
1	(H, Y), (A, X), (B, X), (G, X), (I, Y), (N, Y)
2	(H, Y), (A, X), (B, X), (I, Y), (G, X), N, Y)
3	(H, Y), (A, X), (B, X), (I, Y), (N, Y), (G, X)
4	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (K, Y), (D, X), (G, X), (N, Y)
5	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (K, Y), (D, X), (N, Y), (G, X)
6	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (K, Y), (N, Y), (D, X), (G, X)
7	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z),(O, Y), (E, X), (F, X), (G, X), (M, Y), (S, Z), (N, Y)
8	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (G, X), (M, Y), (N, Y), (S, Z)
9	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (G, X), (S, Z), (N, Y)
10	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (G, X), (N, Y), (S, Y)
11	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (S, Z), (G, X), (N, Y)
12	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (S, Z), (N, Y), (G, X)
13	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (N, Y), (G, X), (S, Z)
14	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (N, Y), (S, Y), (G, X)

Table 1. System execution data of the payment system

Based on this information, the architect can draw the modules with their features, as shown in Fig. 1. In the remainder of this paper, we investigate a method to use event logs, such as the one shown in Tbl. 1, to complete the diagram and derive a behavioral specification of the system.

3 Preliminaries

Let S be a set. The powerset of S is denoted by $\mathcal{P}(S) = \{S' \mid S' \subseteq S\}$. We use $|S|$ for the number of elements in S. Two sets U and V are *disjoint* if $U \cap V = \varnothing$. Some set S with relation $\leqslant$ is a partial order, denoted by $(S, \leqslant)$, iff $\leqslant$ is reflexive, i.e. $a \leqslant a$ for all $a \in S$, antisymmetric, i.e. $a \leqslant b$ and $b \leqslant a$ imply $a = b$ for all $a, b \in S$, and transitive, i.e. $a \leqslant b$ and $b \leqslant c$ imply $a \leqslant c$ for all $a, b, c \in S$. Given a relation $R \subseteq S \times S$ for some set S, we denote its transitive closure by R^+, and the transitive and reflexive closure by R^*.

A *bag* m over S is a function $m : S \to \mathbb{N}$, where $\mathbb{N} = \{0, 1, \ldots\}$ denotes the set of natural numbers. We denote e.g. the bag m with an element a occurring once, b occurring three times and c occurring twice by $m = [a, b^3, c^2]$. The set of all bags over S is denoted by $\mathbb{N}^S$. Sets can be seen as a special kind of bag where all elements occur only once; we interpret sets in this way whenever we use them in operations on bags. We use $+$ and $-$ for the sum and difference of two bags, and $=, <, >, \leqslant, \geqslant$ for the comparison of two bags, which are defined in a standard way.

A *sequence* over S of length $n \in \mathbb{N}$ is a function $\sigma : \{1, \ldots, n\} \to S$. If $n > 0$ and $\sigma(i) = a_i$ for $i \in \{1, \ldots, n\}$, we write $\sigma = \langle a_1, \ldots, a_n \rangle$. The length of a sequence is denoted by $|\sigma|$. The sequence of length 0 is called the *empty sequence*, and is denoted by ϵ. The set of all finite sequences over S is denoted by S^*. We write $a \in \sigma$ if a $1 \leqslant i \leqslant |\sigma|$ exists such that $\sigma(i) = a$. *Concatenation* of two sequences $\nu, \gamma \in S^*$, denoted by $\sigma = \nu; \gamma$, is a sequence defined by $\sigma : \{1, \ldots, |\nu| + |\gamma|\} \to S$, such that $\sigma(i) = \nu(i)$ for $1 \leqslant i \leqslant |\nu|$, and $\sigma(i) = \gamma(i - |\nu|)$ for $|\nu| + 1 \leqslant i \leqslant |\nu| + |\gamma|$. A sequence σ can be projected over some set U, denoted by $\sigma_{|U}$, and is inductively defined by $\epsilon_{|U} = \epsilon$, $(\langle a \rangle; \sigma)_{|U} = \langle a \rangle; \sigma_{|U}$ if $a \in U$, and $(\langle a \rangle; \sigma)_{|U} = \sigma_{|U}$ otherwise.

Petri Nets A *Petri net* [16] is a tuple $N = \langle P, T, F \rangle$ where (1) P and T are two disjoint sets of *places* and *transitions* respectively; and (2) $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*. The elements from the set $P \cup T$ are called the *nodes* of N. Elements of F are called *arcs*. Places are depicted as circles, transitions as squares. For each element $(n_1, n_2) \in F$, an arc is drawn from n_1 to n_2.

Let $N = \langle P, T, F \rangle$ be a Petri net. Given a node $n \in (P \cup T)$, we define its *preset* ${}^\bullet_N n = \{n' \mid (n', n) \in F\}$, and its *postset* $n^\bullet_N = \{n' \mid (n, n') \in F\}$. We lift the notation of preset and postset to sets. Given a set $U \subseteq (P \cup T)$, ${}^\bullet_N U = \bigcup_{n \in U} {}^\bullet_N n$ and $U^\bullet_N = \bigcup_{n \in U} n^\bullet_N$. If the context is clear, we omit the N in the subscript.

A *marking* of N is a bag $m \in \mathbb{N}^P$, where $m(p)$ denotes the number of *tokens* in place $p \in P$. If $m(p) > 0$, place p is called *marked* in marking m. A Petri net N with corresponding marking m is written as (N, m) and is called a *marked Petri net*. Given a marked Petri net (N, m), transition t is enabled, denoted by $(N, m)[t\rangle$, if ${}^\bullet t \leqslant m$. If transition t is enabled in (N, m), it can fire, resulting in a new marking m', denoted by $(N, m)[t\rangle(N, m')$, such that $m' + {}^\bullet t = m + t^\bullet$. We lift the firing of transitions to the firing of sequences in a standard way, i.e., a sequence $\sigma \in T^*$ of length n is enabled in (N, m) if markings $m_0, \ldots, m_n$ exist, such that $m = m_0$ and $(N, m_{i-1})[\sigma(i)\rangle(N, m_i)$ for all $1 \leqslant i \leqslant n$. A marking m' is reachable from some marking m in N, denoted by $(N, m)[*\rangle(N, m')$, if a firing sequence $\sigma \in T^*$ exists such that $(N, m)[\sigma\rangle(N, m')$. A marking m' is a *home marking* of (N, m), if for all markings m'' with $(N, m)[*\rangle(N, m'')$, we have $(N, m'')[*\rangle(N, m')$.

A special class of Petri nets are the workflow nets [1]. A workflow net is a tuple $\langle P, T, F, i, f \rangle$ with $\langle P, T, F \rangle$ a Petri net, (2) $i \in P$ is the only place with no incoming transitions, (3) $f \in P$ is the only place with no outgoing transitions, i.e., ${}^\bullet i = f^\bullet = \varnothing$, and (4) all transitions have at least one incoming and one outgoing arc, i.e., ${}^\bullet t \neq \varnothing \neq t^\bullet$ for all $t \in T$.

Open Petri Nets Within a network of asynchronously communicating systems, messages are passed between the elements within the network. The approach we follow is based on Open Petri nets [5]. Communication in an open Petri net (OPN) is represented by special places, called the *interface places*. An interface place is either an *input place*, receiving messages from the outside, or an *output place* that sends messages to the outside of the OPN. An input place is a place that has only outgoing arcs, and an output place has no incoming arcs.

Definition 1. *An* Open Petri net *is defined as an 7-tuple* $\langle P, I, O, T, F, i, \Omega \rangle$ *where (1)* $\langle P \cup I \cup O, T, F \rangle$ *is a Petri net; (2)* P *is a set of* internal places*; (3)* I *is a set of* input places*, and* ${}^\bullet I = \varnothing$*; (4)* O *is a set of* output places*, and* $O^\bullet = \varnothing$*; (5)* P*,* I *and* O *are pairwise disjoint; (6)* $i \in \mathbb{N}^P$ *is the* initial marking*, and (7)* $\Omega \subseteq \mathbb{N}^P$ *is the set of* final markings*. We call the set* $I \cup O$ *the* interface places *of the OPN. An OPN is called* closed *if* $I = O = \varnothing$*.*

An important behavioral property for OPNs is termination: an OPN should always have the possibility to terminate properly. We identify two termination properties: weak termination and soundness.

Definition 2. *Let $\langle P, I, O, T, F, i, f\rangle$ be an OPN. It is* weakly terminating*, if for every reachable marking of the marked Petri net $(\langle P \cup I \cup O, T, F\rangle, i)$ a marking $f \in \Omega$ can be reached.*

It is sound*, if for every reachable marking of the marked Petri net $(\langle P, T, F\rangle, i)$ a marking $f \in \Omega$ can be reached.*

Communication between OPNs is done via the interface places. Two OPNs can only communicate if the input places of the one are the output places of the other, and vice versa.

Definition 3. *Two OPNs A and B are* composable*, denoted by $A \oplus B$, if and only if $(I_A \cap O_B) \cup (O_B \cap I_A) = (P_A \cup T_A \cup I_A \cup O_A) \cap (P_B \cup T_B \cup I_B \cup O_B)$.*

If A and B are composable, they can be composed into a new OPN, denoted by $A \oplus B$, with $A \oplus B = \langle P, I, O, T, F, i, \Omega\rangle$ where $P = P_A \cup P_B \cup G$; $I = (I_A \cup I_B) \setminus G$; $O = (O_A \cup O_B) \setminus G$; $T = T_A \cup T_B$; $F = F_A \cup F_B$; $i = i_A + i_B$; and $f = \Omega_A \cup \Omega_B$ with $G = (I_A \cap O_B) \cup (O_B \cap I_A)$.

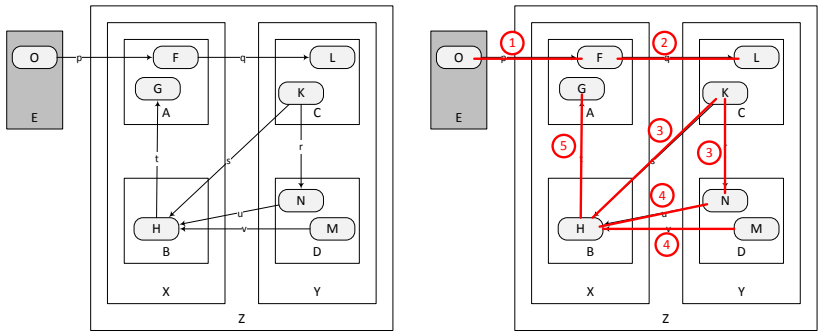
Event Logs and Behavioral Profiles Although event logs are defined as a tuple consisting of a set of case identifiers, events, and an attribute mapping [2], it is in this paper sufficient to consider an event log, denoted by $\mathcal{L}$, as a set of sequences over some alphabet T, i.e., $\mathcal{L} \subseteq T^*$. Given an event log $\mathcal{L}$, we define the *successor relation* [20] by $a <_\mathcal{L} b$ if a sequence $\sigma \in \mathcal{L}$ and $1 \leqslant i \leqslant |\sigma|$ exist, such that $\sigma(i) = a$ and $\sigma(i+1) = b$. Using the successor relation, we define the *behavioral profile* $(\rightarrow_c, \|_c, +_c)_\mathcal{L}$ as three relations: (1) the causality relation $\rightarrow_c$ is defined by $a \rightarrow_c b$ iff $a <_\mathcal{L} b$ and $b \not<_\mathcal{L} a$, (2) the concurrency relation $\|_c$, which is defined by $a \|_c b$ iff both $a <_\mathcal{L} b$ and $b <_\mathcal{L} a$, and (3) the exclusive relation $+_c$ is defined by $a +_c b$ iff both $a \not<_\mathcal{L} b$ and $b \not<_\mathcal{L} a$ [20]. If the context is clear, we omit the subscript.

Given a marked Petri net (N, m) with $N = \langle P, T, F\rangle$, an event log $\mathcal{L} \subseteq T^*$ is called *complete* with respect to (N, m) iff traces $\sigma_1, \sigma_2 \in T^*$ exist such that $(N, m)[\sigma_1; \langle a, b\rangle; \sigma_2\rangle(N,)$ implies $a <_\mathcal{L} b$ for all $a, b \in T$.

4 Functional Architectures

To model the overview of a system, the modules it consists of, and the features these modules offer, we propose the use of the *functional architecture model* (FAM). The functional architecture of a system is "an architectural model which represents at a high level the software products major functions from a usage perspective, and specifies the interactions of functions, internally between each other and externally with other products" [6]. It offers modules containing features. Features of different modules interact via so-called *information flows*.

An example is shown in Fig. 2(a). The FAM contains 1 context module, E, 7 modules, A, B, C, D, X, Y and Z. Modules have features, depicted by the rounded rectangles. For example, module C contains two features, K and L. Between features of different modules, information flows exist, e.g., the information flow (F, q, L) between modules A and C.



(a) Example Functional Architecture Model

(b) Scenario as an overlay

Fig. 2. Example Functional Architecture Model and corresponding scenario as overlay

Definition 4. *A* Functional Architecture Model *(FAM) is defined as a 6-tuple* $\langle \mathcal{M}, \mathcal{C}, \mathcal{F}, h, m, \rightarrow \rangle$ *where*

- $\mathcal{M}$ *is a finite set of* modules*;*
- $\mathcal{C}$ *is a finite set of* context modules*;*
- $\mathcal{F}$ *is a finite set of* features*;*
- $h : \mathcal{M} \rightarrow \mathcal{M}$ *is the* hierarchy function*, such that the transitive closure* h^* *is irreflexive;*
- $m : \mathcal{F} \rightarrow \mathcal{M} \cup \mathcal{C}$ *is a* feature map *that maps each feature to a module, possibly in the context, and this module does not have any children, i.e.* $h^{-1}(m(F)) = \varnothing$ *for all* $F \in \mathcal{F}$*;*
- $\rightarrow \subseteq \mathcal{F} \times \Lambda \times \mathcal{F}$ *is the* information flow*, with* Λ *the label universe, such that for* $(A, l, B) \in \rightarrow$ *we have* $m(A) \neq m(B)$*. The labels for the information flows are unique per feature, i.e.,* (A, l, B) *and* (A, l, C) *imply* $B = C$ *for all labels* $l \in \Lambda$ *and* $(A, l, B), (A, l, C) \in \rightarrow$*.*

Although the information flows define the possible interactions between modules, it remains a static overview of the system. Therefore, one can use scenarios on top of the functional architecture, e.g. by creating an overlay, highlighting the information flows that are executed and the order in which they should occur. Formally, we represent a scenario as a partial order.

Definition 5. *Let* $F = \langle \mathcal{M}, \mathcal{C}, \mathcal{F}, h, m, \rightarrow \rangle$ *be a FAM. A* scenario *of* F *is a pair* $(S, <)$ *with* $S \subseteq \rightarrow$*, such that* $(S, \leqslant)$ *with* $\leqslant = <^*$ *is a partial order.*

An example is shown in Fig. 2(b). The scenario implied by the overlay can be represented by a partial order induced by $(O, p, F) < (F, q, L)$, $(F, q, L) < (K, s, H)$, $(F, q, L) < (K, r, N)$, $(K, r, N) < (N, u, H)$, $(K, s, H) < (H, t, G)$, $(N, u, H) < (H, t, G)$, $(K, r, N) < (M, v, H)$, and $(M, v, H) < (H, t, G)$.

However, such scenarios are typically not specified. Another important drawback of such scenarios is their analyzability. Although each scenario can be checked, the consistency between the different scenarios remains a difficult task. Therefore, in the remainder of this paper, we search for a method to derive the behavioral specification as a network of asynchronously communicating systems, given the system execution data produced by the actual system in the form of event logs.

5 Discovery of a Functional Architecture

In this section, we study the possibilities process mining [2] offers to generate Petri nets for each of the different modules a system consists of. Event logs describe the order in which features of a system have been executed. Such event logs are system wide. Instead of each module having its own event log, only global sequences exist, i.e., sequences concatenate the executed features over all modules. As FAM only allows features to be contained in a single module, we assume that each feature belongs to exactly one module. Also, FAM prescribes communication to be one-directional, i.e., given two communicating features A and B, we assume that either A sends a message to B, or vice versa, that B sends a message to A, but not both.

The behavioral specification of a system is three-fold: (1) communication between modules via their features, (2) the internal behavior within each feature, and (3) the order in which features are called within a module. In this section, we explore all three types of behavioral specification to come to a composed system of asynchronously communicating systems.

In the remainder, let $\mathcal{L}$ be an event log over a set of features T, and let $\mathfrak{R} : T \to M$, with M the set of modules, be a function that maps each feature onto the module that contains that feature.

5.1 Communication between Features

Communication between modules within a system is asynchronous of nature: messages are sent between features in order to complete their functionality. Within an event log, we need to consider the order in which events or features occur. For example, given some trace σ, if the resource is different for two subsequent events, i.e., $\mathfrak{R}(\sigma(i)) \neq \mathfrak{R}(\sigma(i+1))$, then this might indicate that the former sends a message to the latter. This is expressed by the communication successor.

Definition 6 (Communication successor). *Let $\mathcal{L} \subseteq T^*$ be an event log. We define the* communication successor *relation $\lessdot_{\mathcal{L}} \subseteq T \times T$ by $A \lessdot_{\mathcal{L}} B$ iff $\mathfrak{R}(A) \neq \mathfrak{R}(B)$, $\sigma(i) = A$, and $\sigma(i+1) = B$ for some $\sigma \in \mathcal{L}$ and $1 \leqslant i < |\sigma|$.*

Although at first sight the communication successors seem to work, we need to remember the concurrent nature of asynchronous communication. Consider for example the communication between modules M and N as depicted in Fig. 3, and

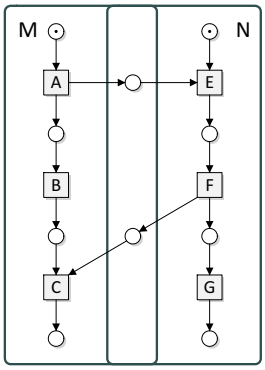


Fig. 3. Modules M and N

Case	Trace
1	A, E, F, G, B, C
2	A, E, F, B, G, C
3	A, E, B, F, G, C
4	A, E, B, F, C, G
5	A, E, F, B, C, G
6	A, B, E, F, G, C
7	A, B, E, F, C, G

Table 2. Corresponding event log

	E	**F**	**G**
A	$\rightarrow$	+	+
B	$\parallel$	$\parallel$	$\parallel$
C	+	$\leftarrow$	$\parallel$

Table 3. Communication behavioral profile

the corresponding allowed sequences in Tbl. 4. We have $A \lessdot E$, which is indeed the communication as modeled in the composition $M \oplus N$. However, we also find $G \lessdot B$, indicating a possible communication between G and B. Listing all communication successors, we get $A \lessdot E$, $G \lessdot B$, $F \lessdot B$, $B \lessdot G$, $G \lessdot C$, $E \lessdot B$, $B \lessdot F$, $F \lessdot C$, $C \lessdot G$, and $B \lessdot E$. Observe that because of the asynchronous nature of the communication, features B and E are concurrently enabled in Fig. 3. Assuming the event log to be complete, this should become visible in the communication successor relation, as for the normal successor relation on event logs.

Definition 7 (Communication behavioral profile). *Let $\mathcal{L} \subseteq T^*$ be an event log, and $\lessdot_{\mathcal{L}} \subseteq T \times T$ the corresponding communication successor relation.*

The communication behavioral profile *is the 3-tuple $(\rightarrow_c, \parallel_c, +_c)^{Com}_{\mathcal{L}}$ defined by:*

- *$A \rightarrow_c B$ iff $A \lessdot_{\mathcal{L}} B$ and $B \not\lessdot_{\mathcal{L}} A$;*
- *$A \parallel_c B$ iff both $A \lessdot_{\mathcal{L}} B$ and $B \lessdot_{\mathcal{L}} A$; and*
- *$A +_c B$ iff both $A \not\lessdot_{\mathcal{L}} B$ and $A \not\lessdot_{\mathcal{L}} B$.*

Calculating the behavioral profile of the communicating transitions using the communication successor relation, results in the communication behavioral profile as shown in Tbl. 3. It shows that B and E are concurrently enabled. Following the behavioral profile, we see that the causal relation of the behavioral profile correctly identifies the feature communication.

Using the communication behavioral profile, we can construct the information flows from an event log as follows. If $A \rightarrow B$ in the communication behavioral

	Debtor							Payment							Creditor	
	A	B	C	D	E	F	G	H	I	J	K	O	M	N	Q	S
A								←	+	+	+	+	+	+	+	+
B								+	→	+	+	+	+	+	+	+
C								+	+	→	+	+	+	+	+	+
D								+	+	+	←	+	+	‖	+	+
E								+	+	+	+	←	+	+	+	+
F								+	+	+	+	+	→	+	+	+
G								+	‖	+	+	+	‖	‖	+	‖
H	→	+	+	+	+	+	+								+	+
I	+	←	+	+	+	+	‖								+	+
J	+	+	←	+	+	+	+								→	+
K	+	+	+	→	+	+	+								+	←
O	+	+	+	+	→	+	+								+	←
M	+	+	+	+	+	←	‖								+	→
N	+	+	+	‖	+	+	‖								+	‖
Q	+	+	+	+	+	+	+	+	+	←	+	+	+	+		
S	+	+	+	+	+	+	‖	+	+	+	→	→	←	‖		

Table 4. Communication behavioral profile for the running example

profile of the event log, then an information flow (A, x, B) exists, with x a fresh label. This results in the following translation:

Definition 8 (Generated FAM). *Let $\mathcal{L}$ be an event log, and $(\rightarrow_c, \|_c, +_c)^{Com}_{\mathcal{L}}$ be its communication behavioral profile. Its corresponding functional architecture model $\langle \mathcal{M}, \mathcal{C}, \mathcal{F}, h, m, \rightarrow \rangle$ is defined by:*

- $\mathcal{M} = \mathfrak{R}(\mathcal{L})$;
- $\mathcal{C} = \emptyset$;
- $\mathcal{F} = T$;
- $h = \emptyset$;
- $m = \mathfrak{R}$; *and*
- $\rightarrow = \{(A, x, B) \mid A \rightarrow_c B,$ *and* $x \in \Lambda$ *a fresh label*$\}$.

After constructing the communication behavioral profile for the running example, shown in Tbl. 4, we can complete the functional architecture model. Based on the given system execution data, we see for example that feature H communicates with feature A, and feature S sends messages to features K and O, and receives messages from feature M. The complete functional architecture of the running example is shown in Fig. 4.

5.2 Internal Behavior of Features

As can be seen in the running example, features can send and receive multiple messages. For example, feature S sometimes sends a message to feature K and sometimes to feature O. Therefore, the next step in discovering the functional

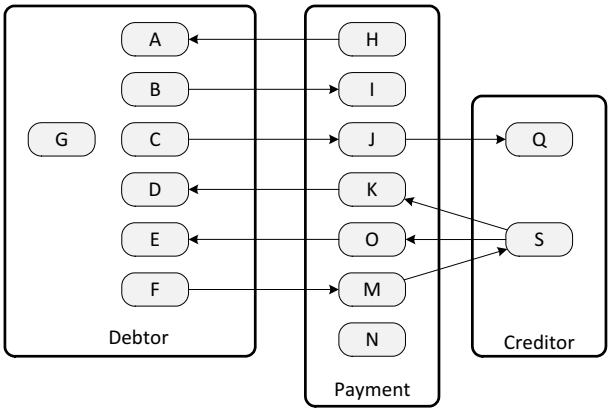


Fig. 4. Functional architecture model of the running example

architecture is to reconstruct the internal behavior of each of the features. For this, we create for each of the features an event log, containing the features that it communicates with. We call this the *feature log.*

Definition 9 (Feature log). *Let $\mathcal{L} \subseteq T^*$ be an event log, and let $F \in T$ be some feature. Let $(\rightarrow_c, \|_c, +_c)^{Com}_{\mathcal{L}}$ be the corresponding communication behavioral profile. The* feature log *$\mathcal{L}_F$ is defined by $\mathcal{L}_F = \{\sigma_{|C(F)} \mid \sigma \in \mathcal{L}, F \in \sigma\}$ where $C(F) = \{A \mid A \rightarrow_c F \vee F \rightarrow_c A\}$.*

Consider for example feature S in the running example. This feature communicates with features K, O and M, i.e., $C(S) = \{K, O, M\}$. Its feature log is the projection of the log on these features, i.e., $\mathcal{L}_S = \{\langle K \rangle, \langle O, M \rangle\}$.

On these feature logs, we apply the inductive miner [13], that always returns a sound workflow net. Next, we transform the discovered workflow net into an open Petri net, to visualize the messages sent and received by the feature. This results in a *feature net* for each of the features present in the event log.

Definition 10 (Feature Net). *Let $\mathcal{L} \subseteq T^*$ be an event log, and let $F \in T$ be some feature. Let $(\rightarrow_c, \|_c, +_c)^{Com}_{\mathcal{L}}$ be the corresponding communication behavioral profile. The* Feature net *$\mathcal{N}_F$ is the OPN $\langle P, I, O, T, F, i, \Omega \rangle$ defined by*

- *$P = \bar{P}$, $T = \bar{T}$, $i = [\bar{i}]$, $\Omega = \{ [\bar{f}] \}$;*
- *$I = \{p_{A-F} \mid A \rightarrow_c F\}$;*
- *$O = \{p_{F-A} \mid F \rightarrow_c A\}$;*
- *$F = \bar{F} \cup \{(t, p_{F-A}) \mid t \in T, \lambda(t) = A, F \rightarrow_c A\}$*
  $\cup \{(p_{A-F}, t) \mid t \in T, \lambda(t) = A, A \rightarrow_c F)\}$.

where $\langle \bar{P}, \bar{T}, \bar{F}, \bar{i}, \bar{f} \rangle$ is the discovered workflow net.

In our running example, each of the 16 features are transformed into a feature net. Most of the features are simple, like for feature H and A, consisting of

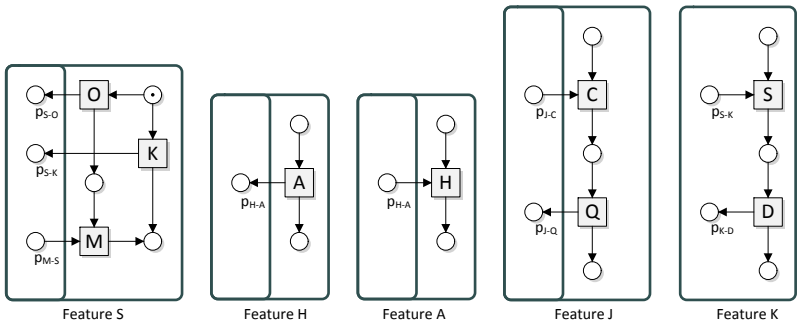


Fig. 5. Some of the feature nets for the running example

a single transition sending a message to A, and receiving a message from H, respectively. A more complex feature net is the net for feature S, which internally decides whether it sends a message to K or to O. Figure 5 depicts some of the feature nets generated using the inductive miner [13].

5.3 Feature Interaction within Modules

Now that each feature has its internal behavior defined by means of a feature net, the next step is to determine the order in which features are executed within each of the modules. As for the features, we first create event logs for each of the modules, by filtering each trace on the features it contains. This results in a *module log* for each of the modules.

Definition 11 (Module Log). *Let $\mathcal{L} \subseteq T^*$ be an event log. Let $M \in Rng(\mathfrak{R})$ be a module. Let $(\rightarrow_c, \|_c, +_c)$ be the corresponding communication behavioral profile. The* Module log *$\mathcal{L}_M$ is defined by $\mathcal{L}_M = \{\sigma_{|\{F|\mathfrak{R}(F)=M\}} \mid \sigma \in \mathcal{L}\}$.*

Within the running example, we obtain three module logs, one for each of the modules. For example, module *Debtor*, has module log $\mathcal{L}_{\text{Debtor}} = \{\langle A, B, G\rangle, \langle A, C, D, G\rangle, \langle A, C, E, F, G\rangle\}$, and for *Creditor* we have $\mathcal{L}_{\text{Creditor}} = \{\langle Q, S\rangle,$

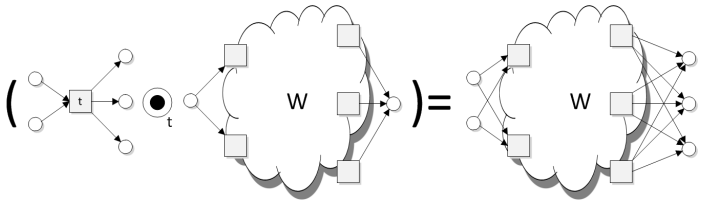


Fig. 6. Refinement of a transition by a workflow net

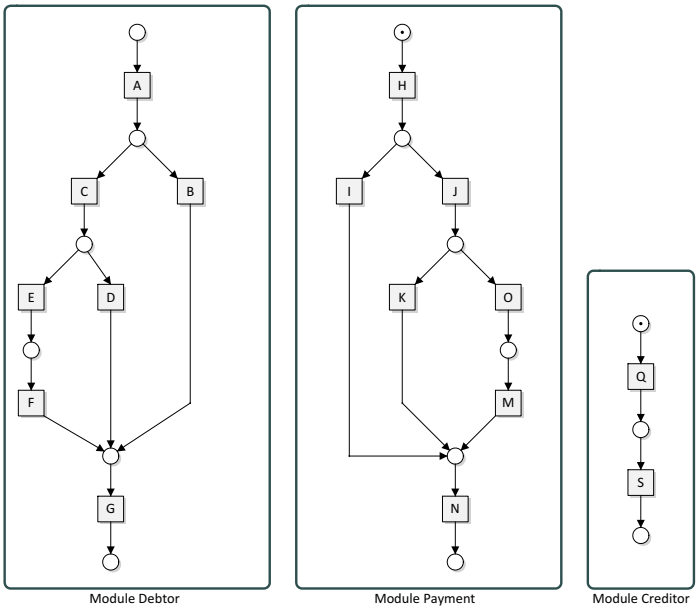


Fig. 7. Module nets for the running example

$\langle Q, S, S \rangle\}$. Again applying the inductive miner results in the three workflow nets as depicted in Fig. 7. Notice that, although feature S occurs twice in one of the sequences, the algorithm only adds a single feature S in the resulting workflow model.

5.4 Composition of Feature Nets and Module Nets

Last step in the process is to combine the feature nets generated for each of the features with the generated module nets. This results in an open Petri net for each of the modules, defining the interaction between the different modules.

In the module net, each feature is represented by a single transition. Next step is to refine each feature by its feature net. For this, we first define the refinement of a transition by a workflow model on open Petri nets, as shown in Fig. 6. This refinement connect each input place of the refined transition with each of the transitions in the postset of the initial place of the workflow, and similarly each output place of the refined transition with each of the transitions in the preset of the final place of the refining workflow. It is straight-forward to prove that if (1) the initial net is sound, (2) each input place of the refined transition is 1-bounded, i.e., it can contain at most one token, and (3) workflow net W is sound, then the refinement yields a sound result.

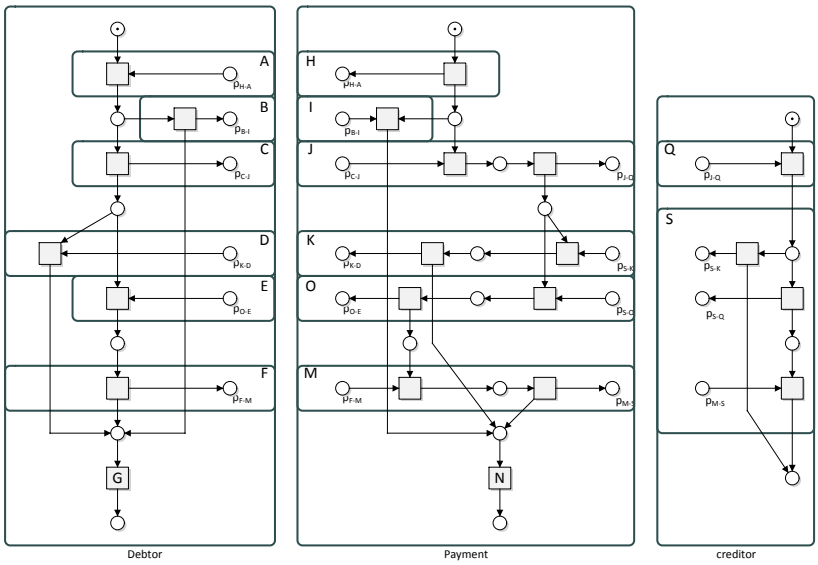


Fig. 8. Composed nets generated for the running example

The result of refining each feature by its feature net is shown in Fig. 8. As features G and N have no feature net defining communication, these transitions are not refined.

To verify whether the resulting open Petri nets are a true representation of the system, one can compose the nets into a single Petri net, and execute each of the sequences of the event log of Tbl. 1 on the resulting model, which in this example is possible. Further analyzing the resulting model shows that its only deadlocks are desirable markings: either all modules reach their final place, without any pending tokens, or the *Creditor* module remains untouched, while the *Debtor* and *Payment* module reach their final place.

6 Conclusions

Within this paper, we discussed a method to automatically generate a functional architecture model from an event log together with a mapping of each feature to the module that offers that functionality. We showed how the information flows can be derived from the communication behavioral profile. This profile not only identifies the information flow for the static structure of the functional architecture, but additionally offers sufficient information to construct the internal behavior for each of the features, and between the features within a module. Lastly, we showed how to compose feature and module nets into an open Petri net.

Discovering the interaction between different modules is not new. Tchniques like service mining [3], apply process mining on event logs to discover a process model of how the services are orchestrated. In the approach presented in this paper, we focus on the discovery of the behavior of each of the modules, rather than a complete orchestration.

In [15], the authors discover the internal behavior of services based on the interaction between two services, guaranteeing deadlock freedom of the discovered service. In the setting of this paper, the exact interaction between modules is unknown, and needs to be discovered first.

The core idea of this paper is twofold: firstly to derive the information flows for a Functional Architecture Model, and secondly to derive the internal behavior for each of the modules within the architecture. Within software architecture, this is called Software Architecture Reconstruction [12]. Although some techniques take the dynamic aspects of the software operation into account, most techniques only focus on the static aspects of software architecture models, using solely the available source code [8]. For example, system execution data is used to enrich architectures with performance data [11] or to visualize traces on how the software is used [19]. In this paper, we propose a method to not only visualize software usage, but to discover module communication and to generate the internal behavior of modules within a software architecture.

Although the approach presented in this paper shows an application of the behavioral profile to discover feature interaction, additional research is required. First, the current approach requires the event log to be complete, i.e., if the log grows, the successor relation should not change. Further, for the generation of the internal feature behavior, we assume that if the sending feature is present in the event log, it enables all possible events, which is possibly a too strict assumption that deserves further investigation.

The approach in this paper is very flexible, as we derive individual models for the features and modules. For this, we apply standard process discovery algorithms returning sound workflow models. However, their composition in general does not result in a sound system of asynchronously communicating systems. Further research is required to study the conditions under which this can be guaranteed. For this, we want to identify conditions which on the one hand result in correct models, and on the other hand have a positive effect on model quality as described by [7].

Not only does this approach provide useful insights for the software architect, we expect the approach applicable to business process management as well, as for the discovery of separate business processes, the Business Process Modelling and Notation offers the swimlane notion. Therefore, we plan to implement the approach in the Process Mining toolkit ProM [18] to experiment and apply the approach on real-life examples.

Acknowledgements The authors would like to thank the anonymous reviewers for their valuable feedback, and Sjaak Brinkkemper, Fabiano Dalpiaz, Garm Lucassen, Leo Pruijt and Erik Jagroep for the fruitfull discussions and valuable input on architecture and software products.

References

1. W.M.P. van der Aalst. Verification of workflow nets. In *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407 – 426. Springer, Berlin, 1997.
2. W.M.P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, Berlin, 2011.
3. W.M.P. van der Aalst. Challenges in service mining: Record, check, discover. In *Web Engineering*, volume 7977 of *Lecture Notes in Computer Science*, pages 1–4. Springer, Berlin, 2013.
4. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Series in Software Engineering. Addison Wesley, Reading, MA, USA, 2012.
5. D. Bera, K. M. van Hee, and J.M.E.M. van der Werf. Designing weakly terminating ros systems. In *Applications and Theory of Petri Nets, (33th International Conference, Petri Nets 2012)*, volume 7347 of *Lecture Notes in Computer Science*, pages 328 – 347. Springer, Berlin, 2012.
6. S. Brinkkemper and S. Pachidi. Functional architecture modeling for the software product industry. In *ECSA 2010*, volume 6285 of *Lecture Notes in Computer Science*, pages 198 – 213. Springer, Berlin, 2010.
7. J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. On the role of fitness, precision, generalization and simplicity in process discovery. In *On the Move to Meaningful Internet Systems: OTM 2012*, volume 7565 of *Lecture Notes in Computer Science*, pages 305–322. Springer, Berlin, 2012.
8. S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *Software Engineering, IEEE Transactions on*, 35(4):573–591, July 2009.
9. S.A. Fricker. Software product management. In *Software for People*, Management for Professionals, pages 53–81. Springer, 2012.
10. K.M. van Hee, N. Sidorova, and J.M.E.M. van der Werf. When can we trust a third party? In *Transactions on Petri Nets and Other Models of Concurrency VIII*, volume 8100 of *Lecture Notes in Computer Science*, pages 106–122. Springer, Berlin, 2013.
11. T. Israr, M. Woodside, and G. Franks. Interaction tree algorithms to extract effective architecture and layered performance models from traces. *Journal of Systems and Software*, 80(4):474 – 492, 2007. Software Performance 5th International Workshop on Software and Performance.
12. R.L. Krikhaar. *Software Architecture Reconstruction*. PhD thesis, VU Amsterdam, 1999.
13. S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst. Discovering block-structured process models from event logs - a constructive approach. In *Application and Theory of Petri Nets and Concurrency*, volume 7927 of *Lecture Notes in Computer Science*, pages 311–329. Springer, Berlin, 2013.
14. G. Lucassen, J.M.E.M. van der Werf, and S. Brinkkemper. Alignment of software product management and software architecture with discussion models. In *IWSPM 2014*, pages 21–30. IEEE, Aug 2014.
15. R. Müller, C. Stahl, W.M.P. van der Aalst, and M Westergaard. Service discovery from observed behavior while guaranteeing deadlock freedom in collaborations. In *Service-Oriented Computing*, volume 8274 of *Lecture Notes in Computer Science*, pages 358–373. Springer, Berlin, 2013.
16. W. Reisig. *Petri Nets: An Introduction*, volume 4 of *Monographs in Theoretical Computer Science: An EATCS Series*. Springer, Berlin, 1985.

17. R.N. Taylor, N. Medvidovic, and E.M. Dashofy. *Software Architecture: Foundations, Theory, and Practice.* John Wiley & Sons, 2010.
18. H.M.W. Verbeek, J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. XES, XESame, and ProM 6. In *Information System Evolution*, volume 72 of *Lecture Notes in Business Information Processing*, pages 60–75. Springer, Berlin, 2011.
19. R.J. Walker, G.C. Murphy, J. Steinbok, and M.P. Robillard. Efficient mapping of software system traces to architectural views. In *Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '00, pages 12–. IBM Press, 2000.
20. M. Weidlich and J.M.E.M. van der Werf. On profiles and footprints – relational semantics for petri nets. In *Applications and Theory of Petri Nets (ICATPN 2012)*, volume 7347 of *Lecture Notes in Computer Science*, pages 148 – 167. Springer, Berlin, 2012.
21. J.M.E.M. van der Werf and H.M.W. Verbeek. Online compliance monitoring of service landscapes. In *BPM 2014 International Workshops*, volume 202 of *Lecture Notes in Business Information Processing*, pages 89–95. Springer, Berlin, 2015.

Bibliography

Aguilar-Savén, R. S. (2004). Business process modelling: Review and framework. *International Journal of Production Economics*, 90(2):129–149.

Berardi, D., Rosa, F. D. E., Santis, L. D. E., Mecella, M., La, R., and Roma, I. (2003). Finite State Automota As Conceptual Model For E-Services. *Journal of Integrated Design and Process Science*, 8(2):105—-121.

Brinkkemper, S. and Pachidi, S. (2010). Functional Architecture Modeling for the Software Product Industry. *Software Architecture SE - 16*, 6285:198–213.

Broersen, J., Dastani, M., and Hulstijn, J. (2001). The BOID architecture: conflicts between beliefs, obligations, intentions and desires. *Proceedings of the fifth ...*.

Brown, M. and Kros, J. (2003). Data mining and the impact of missing data. *Industrial Management & Data Systems*.

Buijs, J. (2014). *Flexible Evolutionary Algorithms for Mining Structured Process Models*. PhD thesis, Technische Universiteit Eindhoven.

Chinosi, M. and Trombetta, A. (2012). BPMN: An introduction to the standard. *Computer Standards & Interfaces*, 34(1):124–134.

Claes, J. and Poels, G. (2013). Process mining and the ProM framework: an exploratory survey. *Business Process Management Workshops*.

Damij, N., Damij, T., Grad, J., and Jelenc, F. (2008). A methodology for business process improvement and IS development. *Information and Software Technology*, 50(11):1127–1141.

de Medeiros, A. (2007). Genetic process mining: an experimental evaluation. *Data Mining and ...*.

der Hoek, W. V. and Wooldridge, M. (2008). Multi-agent systems. *Foundations of Artificial Intelligence*.

Dumas, M., der Aalst, W. V., and Hofstede, A. T. (2005). *Process-aware information systems: bridging people and software through process technology*. John Wiley & Sons.

Dumas, M., Rosa, M. L., Mendling, J., and Reijers, H. (2013). *Fundamentals of business process management*. Springer Berlin Heidelberg.

Engel, R., Krathu, W., and Zapletal, M. (2011). Process mining for electronic data interchange. *E-Commerce and Web ...*.

Fahland, D. (2005). *Complete abstract operational semantics for the web service business process execution language*. PhD thesis, HU Berlin.

Georgakopoulos, D., Hornick, M., and Sheth, A. (1995). An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153.

Gold, E. (1972). System identification via state characterization. *Automatica*.

Grieskamp, W., Gurevich, Y., Schulte, W., and Veanes, M. (2002). Generating finite state machines from abstract state machines. In *ACM SIGSOFT Software Engineering Notes*, volume 27, page 112, New York, New York, USA. ACM Press.

Grosskopf, A., Edelman, J., and Weske, M. (2010). Tangible business process modelingmethodology and experiment design. *Business Process Management ...*.

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

Hinz, S., Schmidt, K., and Stahl, C. (2005). Transforming BPEL to Petri Nets. *Computer*, 3649:220–235.

Jalali, S. and Wohlin, C. (2012). Systematic literature studies. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12*, page 29, New York, New York, USA. ACM Press.

Ko, R. K., Lee, S. S., and Wah Lee, E. (2009). Business process management (BPM) standards: a survey. *Business Process Management Journal*, 15(5):744–791.

Leemans, S. (2014). Discovering block-structured process models from event logs containing infrequent behaviour. *Business Process ...*.

Leitão, P. (2009). Agent-based distributed manufacturing control: A state-of-the-art survey. *Engineering Applications of Artificial Intelligence*, 22(7):979–991.

Lohmann, N. (2008). A feature-complete Petri net semantics for WS-BPEL 2.0. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4937 LNCS, pages 77–91.

Lohmann, N., Verbeek, E., Ouyang, C., and Stahl, C. (2009). Comparing and evaluating Petri net semantics for BPEL.

Maík, V. and McFarlane, D. (2005). Industrial adoption of agent-based technologies.

Mendling, J. (2008). *Metrics for Process Models*, volume 6 of *Lecture Notes in Business Information Processing*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Moher, D. and Liberati, A. (2009). Preferred reporting items for systematic reviews and meta-analyses: the PRISMA statement. *Annals of internal ...*.

Nurmilaakso, J. (2008). EDI, XML and e-business frameworks: A survey. *Computers in Industry*.

Peled, D., Vardi, M., and Yannakakis, M. (1999). Black box checking. *Formal Methods for Protocol ....*

Raffelt, H., Steffen, B., Berg, T., and Margaria, T. (2009). LearnLib: a framework for extrapolating behavioral models. *International Journal on Software Tools for Technology Transfer*, 11(5):393–407.

Rao, A. and Georgeff, M. (1995). BDI agents: From theory to practice. *ICMAS*.

Reisig, W. (2012). *Petri nets: an introduction*. Springer Berlin Heidelberg.

Rozinat, A. and de Medeiros, A. (2008). The need for a process mining evaluation framework in research and practice. *Business Process ....*

Schmiedel, T. and vom Brocke, J. (2015). Business process management: Potentials and challenges of driving innovation. *BPM-Driving Innovation in a Digital World*.

Schobbens, P. Y., Heymans, P., Trigaux, J. C., and Bontemps, Y. (2006). Feature Diagrams: A survey and a formal semantics. In *Proceedings of the IEEE International Conference on Requirements Engineering*, pages 136–145.

Turner, C. J., Tiwari, A., Olaiya, R., and Xu, Y. (2012). Process mining: from theory to practice. *Business Process Management Journal*, 18(3):493–512.

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

Van Der Aalst, W. M. P. (2011). Intra- and inter-organizational process mining: Discovering processes within and between organizations. *Lecture Notes in Business Information Processing*, 92 LNBIP:1–11.

Van Der Aalst, W. M. P. (2012). Process Mining: Overview and Opportunities. *ACM Transactions on Management Information Systems*, 3(2):1–17.

Van Der Aalst, W. M. P., Hofstede, A. T. A. H. M., and Weske, M. (2003). Business Process Management : A Survey. *Business*, 2678:1–12.

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

Van Der Aalst, W. M. P. and Weijters, A. (2005). *Process Mining*, volume 136. Springer Berlin Heidelberg.

van Der Werf, J. M. and Kaats, E. (2015). Discovery of Functional Architectures From Event Logs. In *International Workshop on Petri Nets and Software Engineering*, pages 227–243.

van Hee, K., Sidorova, N., and van der Werf, J. (2013). Business Process Modeling Using Petri Nets. *Transactions on Petri Nets ....*

Verbeek, H. and Buijs, J. (2011). Xes, xesame, and prom 6. *Information Systems ....*

Weidlich, M. and Van Der Werf, J. M. (2012). On profiles and footprints - Relational semantics for Petri nets. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7347 LNCS:148–167.

Weijters, A. (2006). Process mining with the heuristics miner-algorithm. *... Eindhoven, Tech. Rep ....*

Weske, M. (2007). *Business Process Management: Concepts, Languages, Architectures*, volume 54. Springer Berlin Heidelberg.

Wombacher, A. (2004). Transforming BPEL into annotated deterministic finite state automata for service discovery. *Web Services, 2004. ....*