UTRECHT UNIVERSITY

MASTER THESIS

Finding Semantical Patterns in Collections of Workflow Models

Author:
Daniël Stekelenburg
ICA-4153286

Supervisors:
dr. Ad Feelders
dr. ir. Jan Martijn van der Werf

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Algorithmic Data Analysis Group
Department of Information and Computing Sciences
Faculty of Science

August 7, 2018

UTRECHT UNIVERSITY

Abstract

Faculty of Science
Department of Information and Computing Sciences

Master of Science

Finding Semantical Patterns in Collections of Workflow Models

by Daniël Stekelenburg

This work introduces the reader to a thesis study focusing on process mining. The goal of this thesis is to propose a method for describing semantical workflow patterns and to discover their occurrences in a given set of workflow models. Where many others study patterns solely on their syntax, we are interested in patterns with semantical value. This way, we capture patterns at a higher level and are able to reason about their behavior in a set of workflow models. Gaining insight into the usage of workflow models helps workflow management systems to improve their engine. As a case study, we use data from the ERP-company AFAS Software BV. We show that our proposed method is able to recognize a given pattern in such workflow data. The results show that multiple variations of the approval pattern .

Acknowledgements

First of all, I want to thank AFAS Software for giving me the opportunity to do this research internally. My motivation is much higher when I study a problem which is not only a theoretical issue but also has a practical perspective. Over the past year, AFAS not only gave me the resources I needed but really showed their interest in everything I did. They gave me the freedom to tackle this thesis the way I wanted to, but could also give me the time or advice wherever I needed it.

I also want to thank Ad Feelders, my first examinator, and main supervisor. He has helped me a lot throughout this study and gave great advice. He is easy-going, always had time to meet me and really thought along whenever I got stuck.

Furthermore, I want to thank my parents for always supporting me in whatever I do. Without their support, I would never have come this far.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Optimizing the management of business processes becomes more and more important. Companies often use a software application to manage and provide information about all enterprise's data. Such systems are called ERP, which stands for Enterprise Resource Planning. ERP has two major functions:"*(1) a unified enterprise view of the business that encompasses all functions and departments; and (2) an enterprise database where all business transactions are entered, recorded, processed, monitored, and reported*" (Umble et al., 2013).

An example of ERP-software is the software-package *Profit*, provided by a Dutch IT-company AFAS Software BV. For our case study, we will use historical workflow logging data from Profit. This thesis proposes a method for discovering occurrences of a given workflow pattern in a set of workflow models with its main purpose is to get a better understanding of what process (sub)structures represent the behavior in workflows the best.

Workflow management is a technology for engineering business processes. Although the term *workflow* does not have a clear definition, we say that it refers to an *organized set of tasks* to accomplish some business process (Georgakopoulos et al., 1995). The basic idea of a workflow model is to capture dependencies between process tasks in a graph. For instance, when this graph contains a relation $A \rightarrow B$, this means that task A generally precedes task B in an instance of this workflow. Thus, task B cannot be executed until task A is finished. Workflow modeling makes the intended behavior or so-called flow of a business process clearer and can be used to steer such processes in the right direction.
The goal of *workflow mining* is to revert this process, meaning that we try to find a suitable workflow based on a given set of execution logs of a business process.

The remainder of this thesis is organized as follows. At first, we explain a short example as a way to introduce the reader to the problem studied. Then, we discuss the goal and purpose of this study. In Chapter 2 the workflow logging data available at AFAS are described. In Chapter 3 we detail the problem and research questions. Chapter 4 we introduce basic concepts used throughout this thesis. Chapter 5 surveys a literature study and consists of related work. In Chapter 6 techniques used in this thesis are explained. In Chapter 7, we present an approach for solving the problem stated. In Chapter 8 we discuss several results of our approach by studying a workflow pattern. Finally, Chapter 9 concludes this thesis and suggests several directions for future work.

1.1 A Simple Example

To make things more concrete, we present a workflow represented as a petri net (see Definition 5). Figure 1.1 shows the process of buying additional leave. In this case, a manager and an administrator (admin) need to approve the request filled in by

an employee. When one of them does not approve the request, the employee needs to adjust the request and try again. The petri net in Figure 1.1 models the business process described here. Petri nets consist of a set of so-called places (circles) and transitions (squares). When you consider workflows as a finite-state machine, then the places in the model are states and the transitions change the current state. The starting state is represented by *Start*, whereas the workflow is finished when state *End* is reached. Each transition stands for a specific event. According to Definition 2 we define an event as a combination of a task and a corresponding action. Each transition is labeled by the description of the task it represents and the description of the corresponding action. To make a clear distinction between these descriptions, a vertical line ('|') is used to separate them. Considering that a workflow instance begins in *Start*, then only the two transitions that are at the end of an outgoing arc of *Start* can be triggered.

When for example *Approve Manager|Accept* is fired, the state of the workflow instance is moved to the place labeled as *Manager Accepted*. Then, the task *Approve Admin* is enabled, which has as actions *Accept* and *Reject*. Note that the combinations of this task and its actions are translated into transitions. When the manager or the administrator does not agree with the request, then the state gets changed to place *Rejected*. From this state, the only option is to adjust the request and resend, meaning that the whole process of approval by the manager and the administrator gets repeated. Finally - when both agree - the workflow instance is finished.

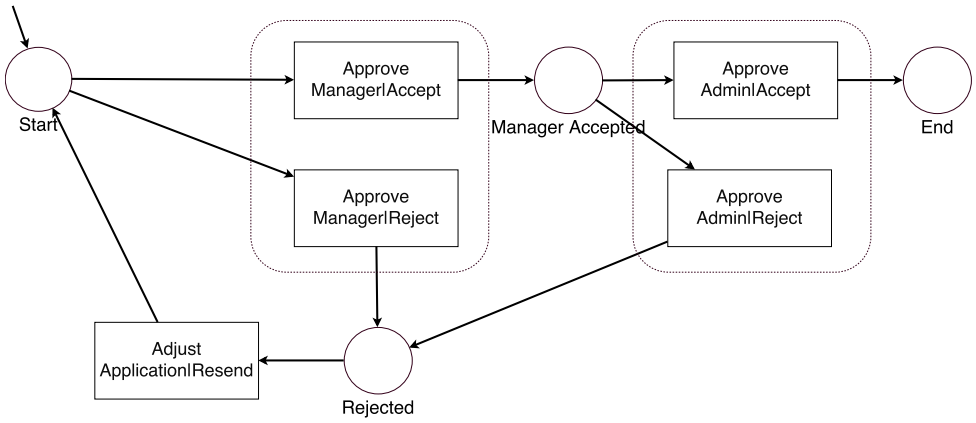

FIGURE 1.1: Example workflow, representing the process of buying additional leave. The dotted areas show occurrences of the approval pattern.

Besides tasks and actions, there are workflow patterns. According to (Riehle and Züllighoven, 1996), a pattern *"is the abstraction from a concrete form which keeps recurring in specific nonarbitrary contexts"*. In other words, a pattern is an abstraction of a concrete set of tasks and actions and is independent of the workflow language used. An example is the OR-split workflow pattern, which is visible in Figure 1.1. Since the approve-tasks have two actions and you may only choose one, you can see these structures as OR-splits. However, a workflow pattern like the OR-split ignores the actual meaning of events and only considers the control-flow.

The problem we try to solve is to find a given workflow pattern within a workflow model. However, we look beyond the purely syntactic scope of patterns and include semantics or contextual meaning of a pattern (i.e. semantical workflow patterns). To give an example, we can say that the workflow in Figure 1.1 contains a pattern which we call the *approval pattern*. This pattern represents the decision

whether some change in the system gets accepted or not. As you can see, such a decision occurs two times in the workflow example: the manager needs to approve the request and also the administrator needs to approve this (indicated as the dotted areas in Figure 1.1). So when we try to find occurrences of the approval pattern, we would like to see these two parts of the workflow in the result of our query.

1.2 Purpose Of This Study

As a business company using ERP-software, you can use workflows provided with your ERP-software package or create your own workflow. Provided workflow templates are general workflows which represent very common business processes most companies are likely to use. When you would like to have a workflow for a very specific process, you might have to assemble your own workflow structure. However, setting up such a workflow can become quite complex which can easily lead to errors in the model defined. Especially when you have to consider multiple different use cases.
To remove the burden of giving the user the option to define a workflow on such a low level by itself, we want to study workflow patterns on a higher level. This way, we abstract from a low-level representation of a workflow and use these high-level patterns to describe processes in an organization.

In the eyes of an IT-company which develops and sells ERP-software, it is important to know which processes are most common in a company and what kind of steps are most important in workflows. Of course, having a good insight into the usage of workflows will be very helpful when you want to deliver a workflow management system. The goal of this research project is getting a better understanding of the high-level workflow patterns[1].

From the literature study (Chapter 5) can be concluded that many others studied workflow mining and pattern matching. Where most of them focus on one issue, we tackle a combination of problems: (1) the discovery of workflow models, given a collection of workflow logs and (2) the discovery of such patterns in workflow models (see Figure 1.2).

[1] From now on, we use the term *workflow pattern* in the context of semantical workflow patterns, unless stated otherwise.

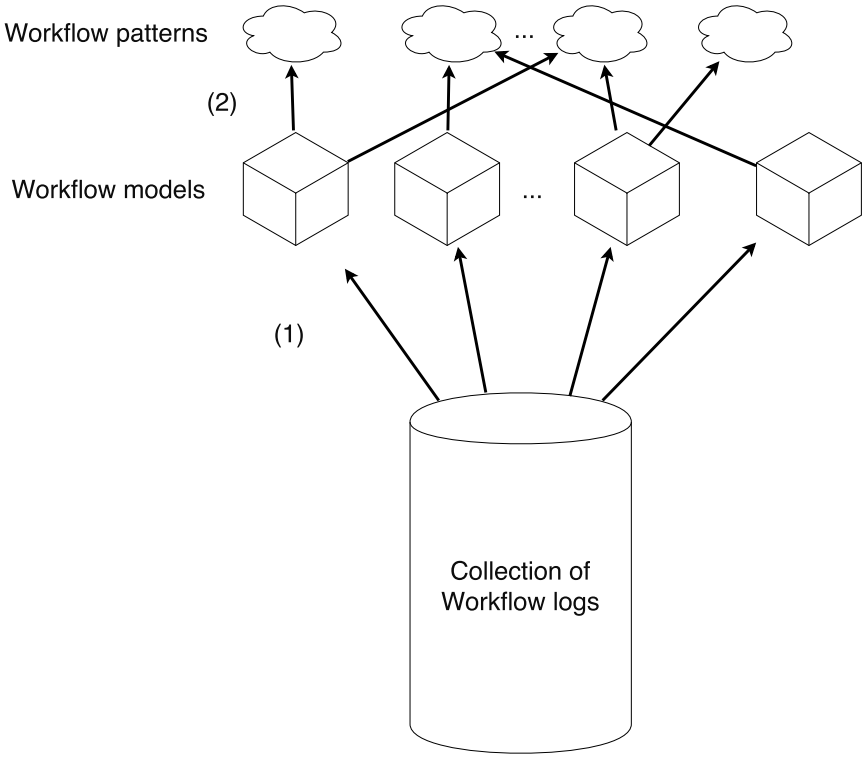

FIGURE 1.2: High-level overview of our problem. (1) A collection of workflow logs need to be translated into workflow models, (2) these workflow models are then searched for occurrences of workflow patterns.

Chapter 2

Workflow Data from Profit

This chapter mainly describes several qualitative properties of workflow logging data from Profit. The workflow log's structure is discussed in Chapter 7.

2.1 Data Quality

Unfortunately, we do not have access to the workflow models used by customers of Profit. Profit itself only consists of templates and basic workflows, but many processes use a workflow which has been customized. Customers can change a standard workflow or design a completely new workflow model which suits their needs. Since the configuration of workflows can be done by the customer, these versions are not stored in one central unit, but every environment keeps track of the workflows it uses. Since the event logs are based on these models - and our approach is dependent on these models - we need to translate these logs into models (step (1) in Figure 1.2). Furthermore, the workflow logs of models which are changed over time are quite hard to work with. A workflow log W of workflow model w gets simply extended when an event is executed for w. Suppose that w gets changed into w' at some point in time. In this case, W first has a set of events based on w but also contains events based on w'. Since there is at first no clear distinction which events belong to which workflow model, we try to group events based on the labels they use. One solution to this issue is explained in Chapter 7, where we try to discover when a workflow model is changed by keeping track of event description and their id's. Whenever a description of an event is changed, but their id is known already, then we can conclude that this is a change in the model. However, this approach does not discover changes in the model when new tasks or actions get added or removed. These changes only add or remove an event to the model, but does not alter an event. Thus, the ability to discover changes made to a workflow model is limited. Another solution would be to remove workflow logs from the dataset when we discover that the workflow model is changed.

Lastly, we ignore workflow instances which consist of incomplete information. This means that in case that some field is empty for some event in this instance, we remove the instance from the workflow log. We do not want incomplete workflow instances to influence the resulting model.

2.2 Data Size

The dataset we used for this thesis, is a collection of workflow logs of the top 64 customer environments based on the number of workflows they actively used, and consists of 4731 workflow logs. Some of these 64 environments are test environments of customers. A customer can use a test environment to test a new workflow model

before actually adding this model to the customer environment itself. Although these environments could contain more noise in their workflow logs, we included them in our research because we are interested in every type of usage of workflow models.

For each environment, we took every workflow event executed between 1 January 2017 and 31 March 2017. We have chosen this period because of the problem described in Figure 2.1. Since we want to obtain workflow models based on this dataset, it is crucial that this collection of workflow logs consists of complete workflow instances. In other words, we want to base our models on instances from which we know every event executed. Incomplete instances are those from which we do not know every event between its starting - and ending event. This last group of instances is seen as unwanted since they are an incorrect representation of the workflow model. For this reason, we tried to filter out instances which are started before 1 January 2017 or were still executing events after 31 March 2017. Note that the last case is not an airtight solution. We assume that most workflow instances are finished at this point in time (April 2018), but we cannot be 100% sure that there are no instances which are still alive. Unfortunately, there is no easy way to confirm this automatically. Thus, although we are able to filter out most of the incomplete instances within the set time period, there is still a small possibility that incomplete instances are included in our dataset. Still, we expect that this group of instances is only a small portion of the total collection and has little influence on the resulting workflow models.

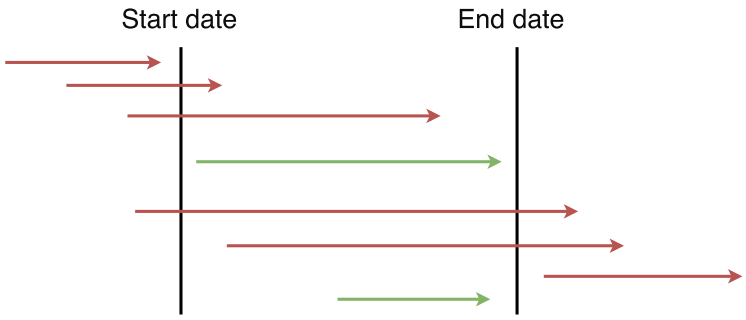

FIGURE 2.1: Overview of possible lifetime of a workflow instance. Red arrows represent instances which are not completely contained within the time period. Green arrows represent instances which are fully contained within the time period.

Due to privacy reasons, the dataset we use is not publicly available.

Chapter 3

Problem and Goal

The challenge which we undertake in this thesis is to reason about behavior in workflows. We try to achieve this by studying so-called higher level workflow patterns and the way such patterns are being used. In short, we will try to answer the following research questions:

1. How can we find variations of assumed[1] workflow patterns, given a collection of workflow models?
   (a) How do we define a workflow pattern and what semantics should be included?
   (b) How can we determine which workflow model contains a workflow pattern?
   (c) What variations of a workflow pattern exists?

When we can find occurrences of a given workflow pattern, then it is also possible to find workflow patterns based on their frequency.

As a case study, we use data provided by AFAS Software BV (Chapter 2). AFAS is interested in the way customers use workflows and with this thesis we have created a tool which they can use for studying workflow patterns. Note that we want to look further than the syntactic value of a pattern (**1a**). We propose a method that is able to discover pattern occurrences based on their syntax and semantics.

We try to capture behavioral properties of a workflow pattern from which AFAS is confident that these are part of their workflow system. However, how these patterns are actually used is often unknown. Recall the approval pattern described in Chapter 1. AFAS is aware of the existence of this pattern, but they are interested in behavioral properties such as the number of times the pattern occurs repetitively within a workflow. This pattern is mainly used and is studied in more detail in Chapter 8.

Creating a tool which is able to detect occurrences of a given workflow pattern can help AFAS to improve their workflow engine in their upcoming product, called NEXT. To be able to discover workflow patterns, we need to look at situations where this pattern is present (**1b**). In other words, we want to know what variations of a workflow pattern exists in a given set of workflow models (**1c**).

[1] AFAS would like to study a specific set of patterns present in the workflow models used in Profit.

Chapter 4

Preliminaries

Let us state some basic definitions which we use throughout this thesis.

Definition 1 (Workflow (Georgakopoulos et al., 1995))**.** A *workflow* W is an organized set of *tasks* to accomplish some business process.

Definition 2 (Task, actions and events)**.** A *task* is a subprocess of a workflow. Each task t has a set of *actions* $A = [a_1, ..., a_m]$, which is enabled when this task is enabled in the workflow (see Definition 5). When an action a_i of task t is triggered, t is completed and the workflow instance proceeds to the next task (or is finished). The execution of some (t, a_i) is called an *event*.

Definition 3 (Workflow log)**.** A *workflow log* L is a collection of traces t of some workflow W. A trace T consists of a set of events E, where each event $e \in E$ contains information about the execution of some event in the workflow instance i.

Definition 4 (Workflow pattern)**.** A *workflow pattern* p is a connected subgraph of a workflow W, i.e. $p \subset W, a_p \subset A, t_p \subset T$, such that this pattern represents a specific objective O.

Recall the approval pattern from the example workflow in Figure 1.1. In this case, the objective O is the *approval* of something.

Definition 5 (Petri net (Van der Aalst, 1998))**.** A petri net is a triple (P, T, F):

- $P = \{p_1, p_2, \cdots, p_m\}$ is a finite set of places,
- $T = \{t_1, t_2, \cdots, t_n\}$ is a finite set of transitions ($P \cap T = \varnothing$ and $P \cup T = \varnothing$),
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs (*flow relation*).

The marking of a petri net PN gets changed according to a firing rule:

1. A transition t is enabled if each input place p of t contains at least one token.
2. Firing an enabled transition t consumes one token in each input place p of t and adds one token to each output place p' of t.

Definition 6 (Workflow net)**.** A petri net representing a workflow is called a *workflow net* (WF-net), which is a petri net having just one start place and one end place (like a workflow should have). When a workflow net consists of blocks which also have the property of starting and ending in one place, this workflow net is called *block-structured*.

Definition 7 (Sound workflow net (Van der Aalst, 1998))**.** Let $N = (P, T, F)$ be a WF-net with input place i and output place o. N is *sound* iff:

1. Every state M from reachable state i, the final state o can get reached by a firing sequence:

$$\forall_M(i \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} o) \tag{4.1}$$

2. State o is the only state reachable from state i with at least one token in place o:

$$\forall_M(i \xrightarrow{*} M \wedge M \geq o) \Rightarrow (M = o) \tag{4.2}$$

3. There are no dead transitions in (N, i):

$$\forall_{t \in T} \exists_{M,M'} i \xrightarrow{*} M \xrightarrow{t} M' \tag{4.3}$$

Definition 8 (Graphs and trees)**.** A *graph* G consists of a set of vertices V and edges E, connecting vertices with each other (i.e. $G = (V, E)$). A graph can be *labeled*, meaning that every vertex $v \in V$ has a label, like an id or name. A graph can contain *cycles*, which are paths where the start-vertex and end-vertex of the path are the same. When no cycles are possible, we also speak of *trees*.

A *rooted tree* is a tree where one vertex v_r is seen as the root or starting point. From this root, a path to every other vertex is possible $(v_r, v_1 \ldots v_x)$. In other words, every *node* (another word for vertex) is said to be connected with the root. Nodes v_a that lie on the path of a node v_x to the root node are said to be ancestors of v_x and v_x is called a descendant of the ancestor nodes v_a. A node v_p is a *parent* node of v_x if v_x is a directly descendant of v_p. In the same situation, v_x is said to be a *child* node of v_p. Nodes in a tree are often referred to as either leaf nodes or internal nodes. Leaf nodes do not have any child nodes, whereas internal nodes do have children. Nodes having the same parent node are called siblings. A tree consists of so-called *subtrees*. We focus on two types of subtrees: *induced subtrees* and *embedded subtrees*.

Definition 9 (Induced subtrees)**.** Assuming that trees P and T are connected, tree $P = (V', E')$ is an induced subtree of tree $T = (V, E)$ if and only if the following constraints are met:

1. $V' \subseteq V$,
2. $E' \subseteq E$,
3. sets of siblings in T should remain in the same order as in P,
4. parent-child relations in P should hold in T. Considering nodes $x, y \in P$ and their matching nodes $x', y' \in T$, then x' is the parent of y' if and only if x is the parent of y.

Definition 10 (Embedded subtrees)**.** Assuming that trees P and T are connected, tree $P = (V', E')$ is an embedded subtree of tree $T = (V, E)$ if and only if the following constraints are met:

1. $V' \subseteq V$,
2. $E' \subseteq E$,
3. ancestor-descendant relations in P should hold in T. Considering nodes $x, y \in P$ and their matching nodes $x', y' \in T$, then x' is an ancestor of y' if and only if x is a ancestor of y.

This having said, we can state that induced subtrees are a proper subset of embedded subtrees. Having defined both types of subtrees, every induced subtree of P in T is also an embedded subtree. However, not every embedded subtree is also an induced subtree.

Chapter 5

Literature Study

This chapter discusses previous works, presenting theories and techniques around workflows, workflow patterns and the problem of mining such structures from workflow models.

5.1 Workflow Paradigm

Business process re-engineering was first proposed by (Hammer, 1990) as an approach to tackle the problem of improving the quality of business processes while reducing their cost. One of the earlier works presenting workflow management systems are (Ellis and Bernal, 1982; Engel et al., 1979). The term *information control net model* is introduced by (Ellis and Bernal, 1982), which can be seen as one of the earlier variants of workflow models.

There are no real standards when it comes to the workflow paradigm (van der Aalst, 1997). This causes management systems to use different modeling languages, but a more important problem is that the ability to verify and analyze workflows is often not available in such tools (van der Aalst, 1997). Because of this reason, (van der Aalst, 1997) shows that a class of *petri nets* can be used as a way to represent workflows, which they call *WF-nets* (workflow nets). Also, the correctness of WF-nets can be analyzed using the petri net theory.

5.2 Workflow Patterns

The most common way to verify whether a workflow language is a good representation is by checking which *workflow patterns* are covered by this language. A workflow pattern is described by (Riehle and Züllighoven, 1996) as *"the abstraction from a concrete from which keeps recurring in specific nonarbitrary contexts"*. In other words, a pattern represents a certain functionality in a workflow while being dependent on the modeling language. Many (Dijkstra et al., 2003; Russell et al., 2004, 2006, 2005; Van der Aalst et al., 2003a) describe workflow patterns providing functional requirements for workflow model languages. This set of requirements is defined as (1) a set of conditions which must be satisfied to be applicable, (2) examples of business situations where this pattern should be applied, (3) a problem description stating why applying this pattern is not trivial and (4) implementation solutions. However, (Dijkstra et al., 2003; Russell et al., 2004, 2006, 2005; Van der Aalst et al., 2003a) focus solely on the syntax of workflow patterns, whereas we are also interested in their semantics.

5.3 Workflow Mining

When in fact the notion of process mining emerged within the last two decades (Van der Aalst et al., 2003b), the concept of workflow mining is first introduced by (Agrawal et al., 1998), presenting an algorithm which has a set of unstructured executions of a business process as input and outputs a minimal dependency graph representing the control flow of this business process. Besides, (Agrawal et al., 1998) discusses ways to cope with cycles in the graph and noise in event logs (missing executions of tasks or wrongly inserted tasks in a log). Other heuristic approaches for noise and incomplete logs are presented in (Măruşter et al., 2002; Weijters, 2001; Weijters and van der Aalst, 2001). The heuristic approach (Weijters, 2001; Weijters and van der Aalst, 2001) is a combination of the following steps: (1) constructing a dependency/frequency table, (2) mining basic dependency relations out of this table and (3) creating a workflow based on the relations found. Besides using these steps, (Măruşter et al., 2002) presents a logistic regression model to learn when two events are direct dependencies of one another.

(Van Der Aalst et al., 2002) states that it is impossible to mine every possible WF-net. They name this challenge the *rediscovery problem*: *"Find a mining algorithm able to rediscover a large class of sound WF-nets on the basis of complete workflow logs."*(Van Der Aalst et al., 2002). An algorithm which can successfully mine a large class of WF-nets is the α-algorithm (van der Aalst and van Dongen, 2002; Van Der Aalst et al., 2002), which assumes that a given workflow log is complete (this means that every possible execution path of the business process at hand must be present in the log). Although this completeness requirement is easy to satisfy for a simple workflow, larger workflows will have more trouble. For instance, a workflow having 10 tasks which can be executed in parallel results in 10! = 3628800 possible execution sequences. As you can tell, there is a rather small chance that every possible sequence is covered in a corresponding workflow log. Besides, the α-algorithm did have issues with loops and self-loops. Multiple extensions on the α-algorithm have been developed. Examples are the $\alpha+$ variant (De Medeiros et al., 2004) which can deal with such short loops, and (van der Aalst and van Dongen, 2002) which applies the algorithm on timed logs. Other approaches are the Heuristics miner, which can deal with noise (Weijters et al., 2006), and the Inductive Miner (Leemans et al., 2013) that finds a sound and fitting process model in polynomial time. Others made an extension on the Inductive Miner, called *Inductive Miner - infrequent* (IMi), which filters out infrequent behavior (Leemans et al., 2014b). IMi creates so-called 80% models, which is based on the Pareto principle. This principle states that "80% of the observed behavior can be explained by a model that is only 20% of the model required to describe all behavior"(Leemans et al., 2014b). This variant is also implemented in ProM.

5.4 Frequent Pattern Mining

Another interesting research topic is obtaining workflow patterns through frequent pattern mining. In this case, the problem of finding frequent patterns in process trees can be defined as follows.

Definition 11 (Frequent Tree Discovery Problem)**.** Given a set of labels L, a set of trees T using L and a frequency threshold $0 \leq k \leq 1$, find all k-frequent trees $t \in T$. Such trees have a frequency of k, meaning that they are present in k% of all trees in T.

Work related to this problem is (Tax et al., 2016), where local process models are mined from a process tree and a corresponding event log. This work formulates multiple pruning techniques for expanding process trees since the structure of process trees are different from others. Since (Tax et al., 2016) only focuses on finding frequent local process models from one process model, this method can be used in order to determine when a pattern is a substantial part of a process model, i.e. when a pattern appears frequently in a model. However, (Chapela-Campa et al., 2017) states that (Tax et al., 2016) fails to measure the frequency of patterns correctly in some cases and proposes the algorithm WoMine, a generic algorithm that is capable to determine the frequency of all types of patterns.

5.5 Meaning of Words

Determining the similarity between workflow models, where we include semantics like labels of task- and actions descriptions is another problem. See for example the workflow nets that are shown in Figure 5.1. Although we may see that these models have similar meaning, explaining to a computer why these models are similar is quite a challenge since a computer detects that the labels consist of different words.

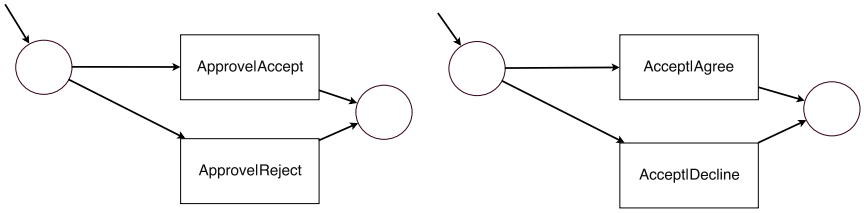

FIGURE 5.1: Workflow nets which seem very similar, but have different labels.

When we search for research done on determining the semantic meanings of words, we see that this field has attracted a lot of attention in the last few years. Especially the Word2Vec model (Mikolov et al., 2013a,b) is a popular technique. This technique results in a vector space representation of words which carries semantic meanings and is very useful in natural language processing tasks. Many variants on Word2Vec have been proposed, like Sent2Vec (Pagliardini et al., 2017). Sent2Vec is particularly useful for our problem since descriptions can contain more than one word. The general idea behind Sent2Vec is computing the average vector of a sentence by looking the vectors up for each word through Word2Vec. More specifically, comparing process models and determining activity similarity has also gained a lot of attention lately (David et al., 2017; Klinkmüller et al., 2013). Where (Klinkmüller et al., 2013) determines activity similarity by applying bag-of-words techniques, (David et al., 2017) uses a Word2Vec approach to cluster such activities. They introduce *Event Variability Reduction (EVR)*, which stands for the projected event log where every event is mapped to its image event (clustered event). Through Word2Vec, events get grouped together, leading to less variability. Since our problem looks very similar to theirs, we include Word2Vec in our approach. Applying techniques like Word2Vec are important for our study since we want to discover pattern occurrences based on the context of their events.

5.6 Workflow Mining Tools

A widely used tool for process mining is ProM (Verbeek et al., 2010). This open-source, extensible framework has over 600 different plug-ins, including process mining algorithms like the α-algorithm, that add their own functionality to the complete software package. ProM gives the user lots of possibilities when it comes down to mining event logs and the user interface helps with getting a good understanding of the structure of process models. Another plug-in, called the “Inductive Miner" (Leemans et al., 2013), is an implementation of the inductive mining algorithm on event logs. Besides using the user interface of ProM, we mainly use its command-line interface to be able to execute plug-ins automatically[1].

[1] https://dirksmetric.wordpress.com/2015/03/11/tutorial-automating-process-mining-with-proms-command-line-interface/

Chapter 6

Theoretical Background

This chapter introduces the reader to the techniques used in this thesis. The Word2Vec model, Inductive Miner algorithm and the quality of process models are subjects which get described below.

6.1 The Word2Vec model

Word2Vec (Mikolov et al., 2013a,b) is a technique for creating word embeddings developed by Google. This technique has two variations, the continuous-bag-of-words model (CBOW) (Mikolov et al., 2013a) and the skip-gram model (Mikolov et al., 2013b). In short, the CBOW model uses multiple context words as input and tries to predict a single so-called 'target' word, whereas the skip-gram model uses a single word as input and predicts context words. The basic idea of these models is shown in Figure 6.1, $w(t-1)$ stands for the word which stands before the so-called target word $w(t)$ etc. Notice that the skip-gram model is the inverse of CBOW when it comes down to the format of input and output. The models use a neural network with one hidden layer. However, this learning process focuses on the weights of the hidden layer. The weights of this hidden layer eventually represent word vectors, which are exactly what we want to know.

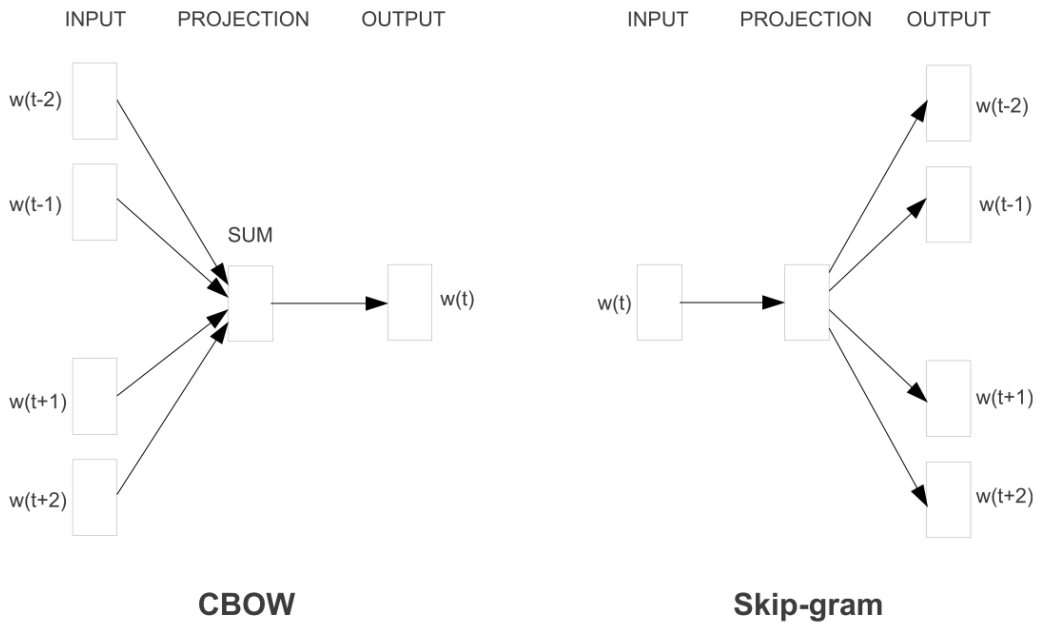

FIGURE 6.1: CBOW and skip-gram model. Models adopted from (Mikolov et al., 2013a,b).

6.1.1 Training a model

Before these word vectors can be used to determine the similarity between words, the neural network needs to be trained. This is done by defining a set of training documents. A document, for example a newspaper article, can be seen as a set of sentences. The neural network is trained by feeding word pairs from these sentences, where a word pair consists of a target word and a context word. To determine the number of context words, a window size is set. In this case, the example models use a window size of 2. That size defines the number of words before and after the target word are seen as context. Figure 6.2 shows what training samples are generated for the skip-gram model from the sentence "The quick brown fox jumps over the lazy dog". By feeding the network these samples, it learns how likely words are used in the same context. For example, assuming that the word "New" always comes before the word "York" in a given set of training documents, the neural network learns that, given the word "York", as context word "New" is very likely (100%).

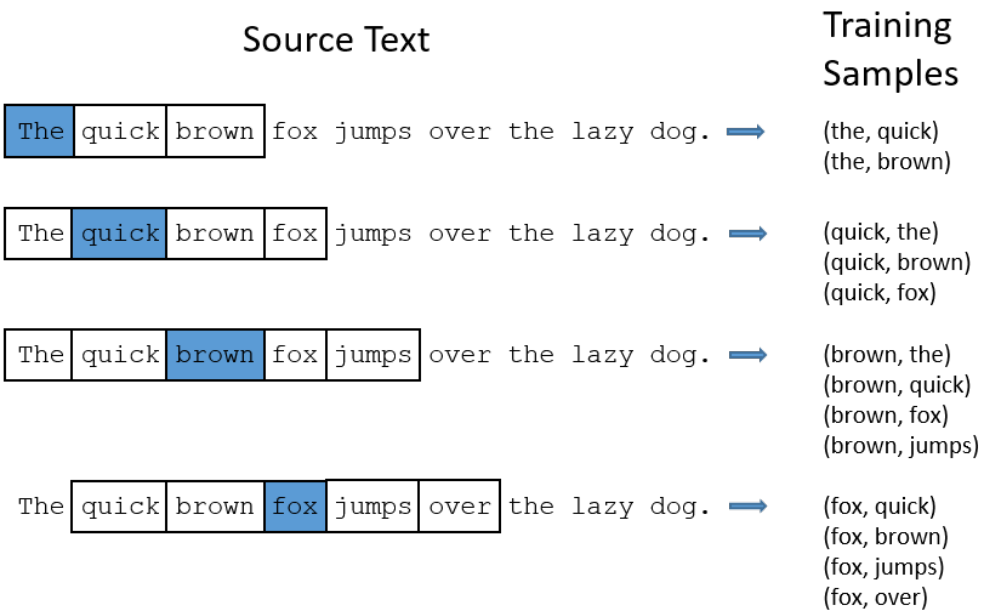

FIGURE 6.2: Example to show how the window size influences the training samples created for training a skip-gram model. The window size is set to 2, where the blue word is the target word and the white words are context words. Figure adopted from (McCormick, 2016).

The way a word gets input in the neural network is by translating it into a so-called one-hot vector (a vector consisting of one '1' and '0' on the remaining positions). The dimension of this vector is the number of distinct words used in the set of training documents. Assuming that the training documents contain 10.000 different words, then the input vector has 10.000 positions. While training the model, the input vector is a one-hot vector and the output is also a one-hot vector. This output vector represents the vector of the output word.

6.1.2 Evaluating a trained model

While evaluating the model, the output layer is actually a probability distribution, where every output value stands for the probability that a certain word is a context

word, given the target word. To make this idea clear, see Figure 6.3. The example target word here is “ants" and the output layer represents the likelihood that words as “abandon" and “ability” appear in the same context as “ants”. The output layer gets ultimately normalized through something named *softmax*. This means that the output layer should sum up to 1. Simply put, the relation between context words and target words can be described as a formula. Given a target word t, the probability of a context word c is defined as:

$$P(c|t) = \frac{\exp(v_t^T u_c)}{\sum_{w=1}^{|V|} \exp(v_t^T u_w)}, \tag{6.1}$$

where v_t stands for the target vector, u_c stands for the context vector, V is the vocabulary size. This function basically applies the dot product to determine the similarity between these vectors. Note that $v_t \cdot u_c$ becomes higher whenever v_t are more similar. For normalization, the dot product gets divided by the total sum over all possible words in the vocabulary V.

Another way to represent the model is in terms of neural networks. Word2vec trains on a given set of training documents by splitting documents up in word pairs. Training the model changes the weights in the neural network. Eventually, the hidden layer represents word vectors for every word present in the dictionary. Finally, evaluating the model using a target word as input results in a probability distribution over the dictionary, where words occurring more often in the context of that target word during training will have a higher probability than non-relating words.

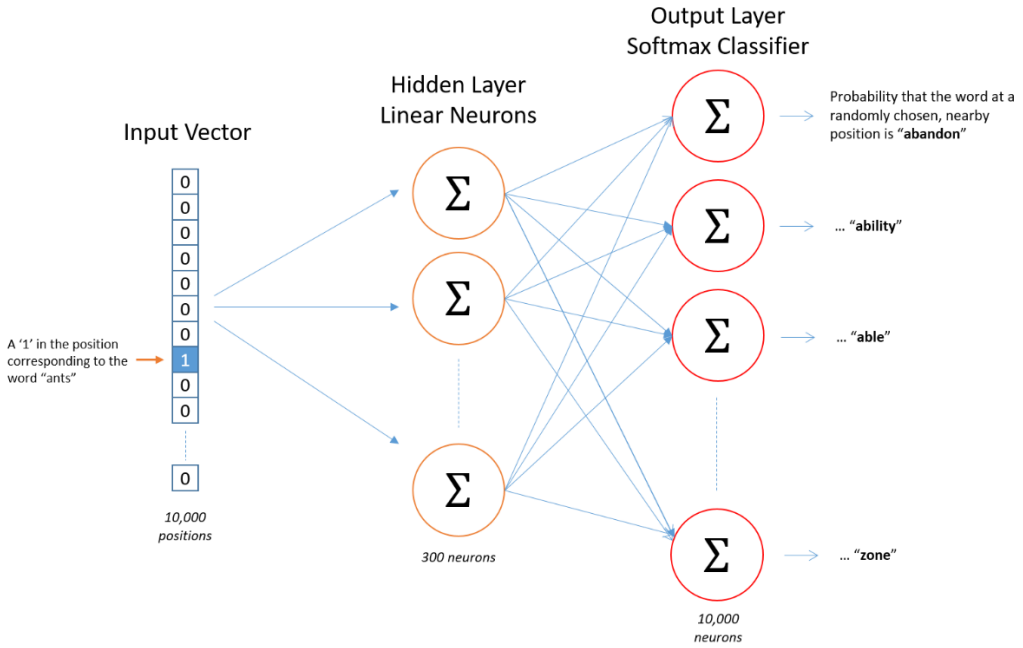

FIGURE 6.3: Description of Word2Vec as a neural network. Figure adopted from (McCormick, 2016).

6.1.3 Negative Sampling

A serious issue is the fact that all neuron weights get tweaked during each training sample. When considering millions of training samples, updating all weights becomes computationally very expensive. To address this problem, *negative sampling* has been introduced in (Mikolov et al., 2013b). With negative sampling, only a small group of the weights gets updated. Take for example the word pair ("fox", "quick"), where "fox" is the input word and "quick" is the positive context word. Ideally, the network would output a one-hot vector. The output neuron of "quick" would lead to the 1 and all the other output neurons would be set to 0. However, negative sampling considers only a few (5-20) output neurons of "negative" words. When considering Figure 6.3, the number of weights are $300 \times 10000 = 3$ million. Given that we only update the weights of the positive output neuron ("quick") and 5 negative output neurons, only $6 \times 300 = 1800$ weights get updated. This shows that negative sampling solves the issue of updating the model.

6.2 The Inductive Miner

As stated in Chapter 5, there are many techniques for mining the process model based on a given workflow log. One of them is the Inductive Miner (Leemans et al., 2013), IM for short, which always finds a sound workflow net (see Definition 7) in a finite time. Therefore, we use this algorithm in the phase of obtaining workflow models. IM is implemented as a plug-in of the ProM framework (Leemans et al., 2014a). Let me briefly explain how this algorithm works.

The Inductive Miner uses a workflow log as input. Then, it determines the most likely split between the different events in the log in a divide-and-conquer approach. As an example, look at Table 6.1. The first split is applied between a and the rest since it observes that every case starts with event a. The same can be said for g, which is the final event for every case. Then, the inner part of every case either ends with e or f, which implies that there is an exclusive choice between e and f. Lastly, the part that remains is a parallel relation between b, c and d since these events always occur, but their order is not fixed.

Based on these splits the algorithm sets up a so-called *process tree* (Buijs et al., 2012b), which is a compact abstract representation of a workflow net (recall that a workflow net is a petri net having one start place and one end place, Definition 6). A process tree is a rooted tree where nodes represent operators and leaves represent events. Leaves with the same parent share a certain relation, which is defined in their parent. See Figure 6.4 for a concrete example.

Case	Events
1.	$< a, b, c, d, e, g >$
2.	$< a, b, c, d, f, g >$
3.	$< a, c, d, b, f, g >$
4.	$< a, b, d, c, e, g >$
5.	$< a, d, c, b, f, g >$

Split	string representation
1.	$\rightarrow (a)$
2.	$\rightarrow (a, g)$
3.	$\rightarrow (a, \times(e, f), g)$
4.	$\rightarrow (a, +(b, c, d), \times(e, f), g)$

TABLE 6.1: Example log, shown in the left table. On the right, you see the string representation of the resulting process tree after a certain split.

IM uses a set $\bigoplus$ which consists of the following operators:

- $\rightarrow$ means a sequential relation between its events. Given $\rightarrow(a, b)$, then event b is executed after the completion of event a.
- $\times$ represents an exclusive choice between its events. Given $\times(a, b)$, either event a or event b is executed. Only one of them are executed, but not both.
- $+$ means a parallel execution between its events. Given $+(a, b)$, event a and b are executed in no particular order.
- $\circlearrowleft$ is the loop-operator. Given $\circlearrowleft(a, b)$, then a correct trace would start with a and end with a. In other words, a is the *do*-part and b is the *redo*-part. Whenever b occurs, then a should occur after.

These operators are used in the string representation of the process tree. On the right subtable of Table 6.1, these representations are shown after each split discussed. Eventually, we end with the process tree $\rightarrow(a, +(b, c, d), \times(e, f), g)$. Note that Figure 6.4 describes the same relations. You should read this tree like: (1) first event a is executed, (2) then b, c and d are executed in parallel, (3) then e or f is executed but not both, and finally (4) g is executed.

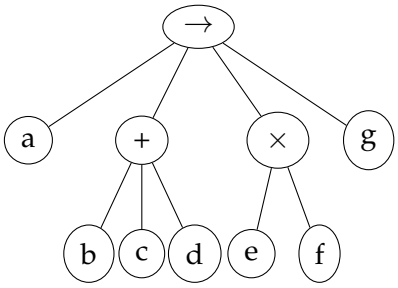

FIGURE 6.4: A possible process tree from splitting the log shown in Table 6.1.

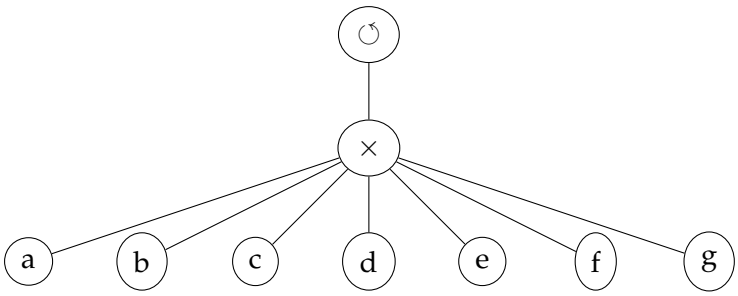

FIGURE 6.5: A typical flower model, based on the log shown in Table 6.1. Such a model allows all possible variations from its children.

You might wonder why process trees are so important, whereas so much research (Van der Aalst et al., 2003b) is done on other modeling languages (petri nets, BPMN, YAWL etc.). However, these standard business process model languages allow the occurrence of anomalies such as deadlocks, live-locks, and improper termination (Buijs et al., 2012b). Since such properties can only be checked afterward, it is quite hard to ensure the soundness property when such a model gets constructed. Process trees do not have this drawback, because of their block-structure (Kopp et al., 2009). This means that every possible process tree is a sound model. The soundness property is not guaranteed by petri nets or other types of models.

6.2.1 Algorithmic Idea

The general idea of the Inductive Miner framework is to compute a log split directly based on the ordering of activities in the given workflow log L. Such splits can be described in terms of *directly-follows graphs*. Such graphs describe which event is directly followed by another event. In Figure 6.6 some directly-follows graphs are shown for an example log $L = \{\langle a, b, c\rangle, \langle a, c, b\rangle, \langle a, d, e\rangle \langle a, d, e, f, d, e\rangle\}$. Nodes in a graph represent activities. An edge (x, y) exists if and only if L contains a trace where x is directly followed by y, i.e. $\langle \ldots, x, y, \ldots\rangle$. The mapping from L to its directly-follows graph is presented in Figure 6.6a, written as $G(L)$. Furthermore, we

denote $Start(G)$ as the set of start nodes and $End(G)$ as the set of end nodes of a given $G(L)$. When looking at Figure 6.6a, $Start(G) = \langle a \rangle$ and $End(G) = \langle b, c, e \rangle$.

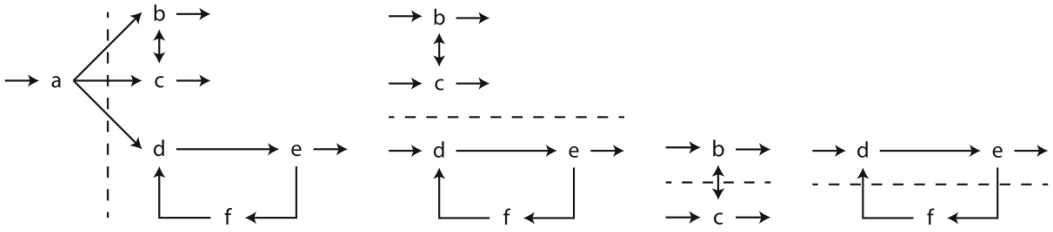

FIGURE 6.6: Example of directly-follows graphs. Arrows represent a directly-follow relation between events. Splits are denoted by dashed lines. This Figure is extracted from (Leemans et al., 2013).

IM tries to find $G(L)$ structures that indicate the most dominant operator describing the behavior of a given event log L. An event log L consists of traces $t \in L$ and traces consist of events $e \in t$.

Each of the four operators $(\rightarrow, \times, +, \circlearrowleft)$ can be described as a specific pattern in $G(L)$ that can be discovered by partitioning the nodes in $G(L)$ into n disjoint sets $\sum_1, \ldots, \sum_n$ in a certain way. The formal definitions of these partitioning are described below. An n-ary split of G is said to be *maximal* if a split of size $> n$ does not exist for G. A split c of size n is said to be *nontrivial* if $n > 1$.

Given a log L, an exclusive split divides the nodes in $G(L)$ into sets $\sum_1, \ldots, \sum_n$ where nodes $a_i \in \sum_i$ and $a_j \in \sum_j, i \neq j$ do not directly follow each other. In other words, $G(L)$ cannot contain an edge (a_i, a_j) when a_i is never directly followed by a_j in any trace in L.

Definition 12 (Exclusive Choice Split, (Leemans et al., 2013))**.** An exclusive choice split is a partitioning $\sum_1, \ldots, \sum_n$ of a directly-follows graph G, where

1. $\forall i \neq j \wedge a_i \in \sum_i \wedge a_j \in \sum_j : (a_i, a_j) \notin G$

A sequence split results in an ordered partitioning $\sum_1, \ldots, \sum_n$ such that for any two nodes $a_i \in \sum_i, a_j \in \sum_j, i < j$, there exists a path from a_i to a_j in $G(L)$, but not vice versa. A path from a_i to a_j is denoted by $a_i \rightsquigarrow a_j$.

Definition 13 (Sequential Split, (Leemans et al., 2013))**.** A sequential split partitions the nodes in a directly-follows graph G into $\sum_1, \ldots, \sum_n$ in such an ordered way that

1. $\forall 1 \leq i < j \leq n \wedge a_i \in \sum_i \wedge a_j \in \sum_j : a_i \rightsquigarrow a_j \in G$
2. $\forall 1 \leq i < j \leq n \wedge a_i \in \sum_i \wedge a_j \in \sum_j : a_j \rightsquigarrow a_i \notin G$

A parallel split leads to a partitioning $\sum_1, \ldots, \sum_n$ where each set can be executed in parallel. This means that any two nodes $a_i \in \sum_i$, $a_j \in \sum_j$, $i \neq j$ are directly followed by each other. This relation $(a_i, a_j), (a_j, a_i) \in G$.

Definition 14 (Parallel Split, (Leemans et al., 2013))**.** A parallel split of a directly-follows graph G leads to $\sum_1, \ldots, \sum_n$, such that

1. $\forall i : \sum_i \bigcap Start(G) \neq \varnothing \wedge \sum_i \bigcap End(G) \neq \varnothing$
2. $\forall i \neq j \wedge a_i \in \sum_i \wedge a_j \in \sum_j : (a_i, a_j) \in G \wedge (a_j, a_i) \in G$

A loop split on a directly-follows graph G results in a partially ordered partitioning $\sum_1, \ldots, \sum_n$, where $\sum_1$ consists of the start and end nodes of G, where no edge (a_i, a_j) exists between any two nodes $a_i \in \sum_{i>1}, a_j \in \sum_{j>1}, i \neq j$. Note that there can be edges from a node in $\sum_1$ to any other $\sum_{i>1}$ and vice versa. Such edges either leave an end node of G or enter a start node of G.

Definition 15 (Loop Split, (Leemans et al., 2013))**.** A loop split of a directly-follows graph G results in a partially ordered split $\sum_1, \ldots, \sum_n$, such that

1. $Start(G) \cup End(G) \subset \sum_1$
2. $\forall i \neq 1 \wedge a_i \in \sum_i \wedge a_1 \in \sum_1 : (a_1, a_i) \in G \Rightarrow a_1 \in End(G)$
3. $\forall i \neq 1 \wedge a_i \in \sum_i \wedge a_1 \in \sum_1 : (a_i, a_1) \in G \Rightarrow a_1 \in Start(G)$
4. $\forall i \neq j \neq 1 \wedge a_i \in \sum_i \wedge a_j \in \sum_j : (a_i, a_j) \notin G$
5. $\forall i \neq 1 \wedge a_i \in \sum_i \wedge a_1 \in Start(G) : (\exists a_1^{'} \in \sum_1 : (a_i, a_1^{'}) \in G) \Leftrightarrow (a_i, a_1) \in G$
6. $\forall i \neq 1 \wedge a_i \in \sum_i \wedge a_1 \in End(G) : (\exists a_1^{'} \in \sum_1 : (a_1^{'}, a_i) \in G) \Leftrightarrow (a_1, a_j) \in G$

Now that the pattern for each operator is defined in terms of a directly-follows graph G, we can work out an example by looking at the corresponding pseudo-code (Leemans et al., 2013) of the Inductive Miner.

The Inductive Miner basically consists of a framework - called B in (Leemans et al., 2013) - and a *select* function. B is described in Function 1 and *select* is described in Function 2. The framework works as a divide-and-conquer approach. It starts with a log L and searches for possible splits into smaller parts $L_1, \ldots, L_n$. Combining $L_1, \ldots, L_n$ with a split operator from $\bigoplus$ should produce L again. As notation, $\sum$ is a finite alphabet of activities, τ represents a silent activity, M_i is a process tree describing L_i and ϵ represents an empty activity. Together with an operator $\bigoplus$,

$\bigoplus(M_1, \ldots, M_n)$ represents the process tree of log L.

Function 1: function B_{select}, (Leemans et al., 2013).

framework B (L, ϕ);
Input : A log L and counter parameter ϕ.
Output: A process tree string.
if $L = \{\epsilon\}$ **then**
  $base \leftarrow \{\tau\}$
else if $\exists a \in \sum : L = \{\langle a \rangle\}$ **then**
  $base \leftarrow \{a\}$
else
  $base \leftarrow \varnothing$
end
$P \leftarrow select(L)$
if $|P| = 0$ **then**
  if $base = \varnothing$ **then**
    return $\{\circlearrowleft(\tau, a_1, \ldots, a_m)$ where $\{a_1, \ldots, a_m\} = \sum(L)\}$
  else
    return $base$
  end
return
$\{\bigoplus(M_1, \ldots, M_n) | (\bigoplus, ((L_1, \phi_1), \ldots, (L_n, \phi_n))) \in P \wedge \forall i : M_i \in B(L_i, \phi_i)\} \cup base$

The framework deals with recursion, whereas the actual log split is done within the *select* function.

Function 2: The Inductive Miner *select* function, (Leemans et al., 2013).

function select (L)
Input : A log L.
Output: A single log division.
if $\epsilon \in L \vee \exists a \in \sum(L) : L = \{\langle a \rangle\}$ **then**
  return $\varnothing$
else if $c \leftarrow$ *a nontrivial maximal exclusive choice cut c of* $G(L)$ **then**
  $\sum_1, \ldots, \sum_n \leftarrow c$;
  $L_1, \ldots, L_n \leftarrow ExclusiveChoiceSplit(L, (\sum_1, \ldots, \sum_n))$;
  return $\{(\times, ((L_1, 0), \ldots, (L_n, 0)))\}$;
else if $c \leftarrow$ *a nontrivial maximal sequence split c of* $G(L)$ **then**
  $\sum_1, \ldots, \sum_n \leftarrow c$;
  $L_1, \ldots, L_n \leftarrow SequenceSplit(L, (\sum_1, \ldots, \sum_n))$;
  return $\{(\rightarrow, ((L_1, 0), \ldots, (L_n, 0)))\}$;
else if $c \leftarrow$ *a nontrivial maximal parallel split c of* $G(L)$ **then**
  $\sum_1, \ldots, \sum_n \leftarrow c$;
  $L_1, \ldots, L_n \leftarrow ParallelSplit(L, (\sum_1, \ldots, \sum_n))$;
  return $\{(+, ((L_1, 0), \ldots, (L_n, 0)))\}$;
else if $c \leftarrow$ *a nontrivial loop split c of* $G(L)$ **then**
  $\sum_1, \ldots, \sum_n \leftarrow c$;
  $L_1, \ldots, L_n \leftarrow LoopSplit(L, (\sum_1, \ldots, \sum_n))$;
  return $\{(\circlearrowleft, ((L_1, 0), \ldots, (L_n, 0)))\}$;
else
  return $\varnothing$;
end

Let us work out an example that leads to the splits in Figure 6.6. Each Function

that represents a split is based on (Leemans et al., 2013).

Given the event log $L = \{\langle a,b,c\rangle, \langle a,c,b\rangle, \langle a,d,e\rangle\ \langle a,d,e,f,d,e\rangle\}$, $G(L)$ is shown in Figure 6.6a where the sequence split $\{a\}, \{b,c,d,e,f\}$ is applied. The log L gets split up by projecting every trace t in L on the split. In other terms, $SequenceSplit(L, (\{a\}, \{b,c,d,e,f\}))$ results into $\{\langle a\rangle\}, \{\langle b,c\rangle, \langle c,b\rangle, \langle d,e\rangle, \langle d,e,f,d,e\rangle\}$. We define the second log as L_2 and $G(L_2)$ is shown in Figure 6.6b.

---

1 function SequenceSplit $(L, (\sum_1, \ldots, \sum_n))$
2 $\forall j : L_j \leftarrow \{t_j | t_1 \cdot t_2 \cdots t_n \in L \wedge \forall i \leq n \wedge e \in t_i : e \in \sum_i\}$
3 return $L_1, \ldots, L_n$

The next split, as shown in Figure 6.6b, is an exclusive choice split $\{b,c\}, \{d,e,f\}$. This split divides every trace t from L_2 into two logs; one log where each trace contains {b,c} and one log where each trace contains {d,e,f}. $ExclusiveChoiceSplit(L_2, (\{b,c\}, \{d,e,f\}))$ results in $\{\langle b,c\rangle, \langle c,b\rangle\}, \{\langle d,e\rangle, \langle d,e,f,d,e\rangle\}$. We define the first log as L_3 (Figure 6.6c) and the second log as L_4 (Figure 6.6d).

---

1 function ExclusiveChoiceSplit $(L, (\sum_1, \ldots, \sum_n))$
2 $\forall i : L_i \leftarrow \{t | t \in L \wedge e \in t : e \in \sum_i\}$
3 return $L_1, \ldots, L_n$

Continuing with L_3, a parallel split between b and c is possible as you can see in Figure 6.6c. $ParallelSplit(L_3, (\{b\}, \{c\}))$ leads to $L_5 = \{\langle b\rangle\}, L_6 = \{\langle c\rangle\}$. t_{E_j} stands for the projection of trace t onto the activity set of E_j, such that all remaining activities in t_{E_j} are included in E_j.

---

1 function ParallelSplit $(L, (\sum_1, \ldots, \sum_n))$
2 $\forall i : L_i \leftarrow \{t_{\sum_j} | t \in L\}$
3 return $L_1, \ldots, L_n$

The last split is done on L_4 and shown in Figure 6.6d. This is the loop split $\{d,e\}$, $\{f\}$. A loop split divides each trace $t \in L$ into subtraces of the loop body and the loop return condition. In our case, $LoopSplit(L_4, (\{d,e\}, \{f\}))$ results in $L_7 = \{\langle d,e\rangle\}$, $L_8 = \{\langle f\rangle\}$.

---

1 function LoopSplit $(L, (\sum_1, \ldots, \sum_n))$
2

$$\forall i : L_i \leftarrow \{t_2 | t_1 \cdot t_2 \cdot t_3 \in L \wedge \sum(\{t_2\}) \subseteq \sum_i \wedge (t_1 = \epsilon \vee (t_1 = \langle \cdots, a_1\rangle \wedge a_1 \notin \sum_i))) \wedge (t_3 = \epsilon \vee (t_3 = \langle \cdots, a_3\rangle \wedge a_3 \notin \sum_i)))\}$$

3 return $L_1, \ldots, L_n$

Combining framework B and the *select* function as shown in Function 2 returns a proper process model. At first, we start of with L, for which no nontrivial exclusive split exists. As we showed earlier, L can be properly split by $SequenceSplit$, which returns $\{\rightarrow, (L_1, L_2)\}$ and B converts this result in the partial model $M = \rightarrow (B(L_1), B(L_2))$.
When processing log L_1, $B(L_1)$ defines *base* as $\{a\}$ and the *select* function returns $\varnothing$.

This leads to the partial model $M = \rightarrow (a, B(L_2))$.
When processing log L_2, the log gets split exclusively into $L_3 = \{\langle b, c\rangle, \langle c, b\rangle, \langle c, b\rangle\}$ and $L_4 = \{\langle d, e\rangle, \langle d, e, f, d, e\rangle\}$. M becomes $\rightarrow (a, \times(B(L_3), B(L_4))$. For L_3, we use a parallel split which results in $L_5 = \{\langle b\rangle\}$ and $L_6 = \{\langle c\rangle\}$. This eventually leads to $M = \rightarrow (a, \times(+(b, c), B(L_4))$, since $B(L_5)$ and $B(L_6)$ are simple base cases and similar to the processing of L_1.
At last, a loop split is applied on L_4, resulting into $L_7 = \{\langle d, e\rangle\}$ and $L_8 = \{\langle f\rangle\}$. Using $SequenceSplit$, L_7 leads to $\rightarrow (d, e)$. $B(L_8)$ is also a base case, leading to f.
The final model discovered by the framework B, using the *select* function as stated above, is $M = \rightarrow (a, \times(+(b, c), \circlearrowleft (\rightarrow (d, e), f))$.

For further information and proofs on the correctness of the Inductive Miner, we refer you to read (Leemans et al., 2013).

6.2.2 Inductive Miner infrequent

Besides the Inductive Miner, we also make use of the Inductive Miner infrequent (IMi) (Leemans et al., 2014b). This variant makes use of a noise threshold ϵ, which is used to filter out infrequent events on each local step in the Inductive Miner algorithm; during the selection of an operator and cut, while splitting a log or during the base cases of the recursion. For instance, given ϵ, IMi considers events to be ϵ-infrequent when they occur less then ϵ times the frequency of the most frequent event y occurs at some point in a trace.

IMi is advantageous over IM in terms of runtime and still results in a sound model. However, models discovered with IMi often have a lower fitness than models obtained by IM (Leemans et al., 2014a) since IMi filters away infrequent events. Because of this filtering, we cannot guarantee that every trace in a workflow log can be replayed in its corresponding workflow model when it is obtained by IMi.

6.3 Measuring Quality of Process Models

A lot of research (Van der Aalst et al., 2003b) has been done on other modeling languages (petri nets, BPMN, YAWL etc.). However, these standard business process model languages allow the occurrence of anomalies such as deadlocks, live-locks, and improper termination (Buijs et al., 2012b). Since such properties can only be checked afterward, it is quite hard to ensure the soundness property when a model gets constructed. Process trees do not have this drawback, because of their block-structure (Kopp et al., 2009). This means that every possible process tree is a sound model, which is a property that petri nets or other types of models cannot guarantee.

Furthermore, there are four quality dimensions that play a role when it comes down to the quality of a process tree (Buijs et al., 2012b). The values of these dimensions lie within a 0 to 1 range, where 1 is the optimal value. Although we do not these quality operators during our study, we present them to the reader to be aware that a workflow model can be checked on quality.

1. The *(replay) fitness* quality dimension stands for the ability to allow all observed behavior in the given log. The way this dimension actually works is that the traces from the event log gets aligned with the process tree. If an event cannot be aligned, events are skipped or inserted in the trace such that this trace is still *replayable*. Skipping or inserting events for this purpose result in a certain

penalty. The replay fitness is defined as:

$$Q_{rf} = 1 - \frac{\text{Total cost for aligning model and event log}}{\text{Minimal cost to align arbitrary event log on model and vice versa}} \tag{6.2}$$

where the denominator is meant to normalize the final outcome.

2. The *simplicity* dimension prefers simpler models that can explain the observed behavior. This property is based on Occam's Razor. The process tree in Figure 6.5 might be considered as better than the tree in Figure 6.4 in this case. Simplicity gets computed as follows:

$$Q_s = 1 - \frac{\#\text{duplicate activities} + \#\text{missing activities}}{\#\text{nodes in process tree} + \#\text{event classes in event log}} \tag{6.3}$$

3. The *precision* quality dimension stands for the inability to allow behavior which is different from the observed behavior. A very generic process tree might allow all possible variations of events, leading to a model which can be seen as imprecise. For this purpose, the precision quality dimension is a good way to check whether your model is under-fitted or not. This dimension can be represented in terms of state space. Recall that a marking is a certain state in a Petri Net. Markings consist of a set of places where each place has a number of incoming and outgoing edges. This dimension uses these numbers in its calculation:

$$Q_p = 1 - \frac{\sum_{\text{visited markings}} \#\text{visits} \times \frac{\#\text{outgoing edges} - \#\text{used edges}}{\#\text{outgoing edges}}}{\#\text{total visited markings overall}} \tag{6.4}$$

In short, a model has optimal precision if and only if all states in the model are actually visited according to the event log. Whenever the model has unused markings, its precision decreases.

4. The *generalization* quality dimension aims to avoid over-fitting. Where the precision dimension aims for models which do not allow other behavior than those seen in the log, the generalization quality increases when a model allows more than just the observed behavior.

$$Q_g = 1 - \frac{\sum_{\text{nodes}} (\sqrt{\#\text{executions}})^{-1}}{\#\text{nodes}} \tag{6.5}$$

Important to know is that reaching a generalization of 1 is impossible since you would need to have infinitely many executions of nodes. The more nodes are visited, the smaller the numerator will get.

A desired trade-off between these quality dimensions depends on the structure of the events log and the purpose of the models. For instance, when your workflow log consists of a lot of variability, the resulting process tree might become rather complicated when you mainly focus on the precision dimension. An overall quality

is computed by normalizing the four quality dimensions. Defining weights for a given dimension X as w_x, the overall quality is defined as:

$$Q = \frac{w_{rf} \cdot Q_{rf} + w_s \cdot Q_s + w_p \cdot Q_p + w_g \cdot Q_g}{w_{rf} + w_s + w_p + w_g} \tag{6.6}$$

Important to note is that the Inductive Miner always results in a perfect replay fitness. However, there is no guarantee over the other three quality measures.

Although a user cannot influence these quality weights when applying the Inductive Miner, there is an alternative which makes this possible. With the Evolutionary Tree Miner (Buijs et al., 2012a) - ETM for short - a user is able to configure the quality weights. This way the user can let the algorithm prefer certain quality measures over others. Like IM, ETM is also implemented as a plug-in in ProM.

Unfortunately for us, running ETM on a single workflow log takes a couple of minutes on the machine that we use for this study. This means that processing all 4000+ workflow logs would take days, which makes ETM impractical. Furthermore, ETM does not always result in higher quality models. It requires a user to experiment with the configuration to obtain the best process models. Because of these issues, we decided to keep IM as our algorithm for process model discovery. Nevertheless, with ETM it is possible to obtain process models with a higher overall quality (Buijs et al., 2012a).

Chapter 7

Our Approach

This chapter presents a method for converting workflow logs into workflow models. In Chapter 7.1, we study the raw logging data of AFAS. In Chapter 7.2 we describe how to convert such workflow logs into workflow models. Chapter 7.3 discusses a way to define workflow patterns in terms of a process tree and presents an algorithm which finds occurrences of a given pattern in a given workflow model. In Chapter 7.5 we explain how the proposed method is realized in the form of a tool.

7.1 Studying Raw Data

For over 10 years, workflow events are being logged in the software package Profit. These logs can be seen as a table where each record represents an action on a task in a certain workflow instance. Although the logs contain more fields, the most relevant fields are shown in an example log in Table 7.1. The first field is the *Workflow ID* and is a unique reference to a workflow model (combination of task and action like shown in Figure 1.1) from which this event sequence is an instance. Although we did not include more workflows in the example, there are many workflow models used, so this field is not superfluous. By grouping on this field, you collect all event logs of that specific workflow log.

The *Case ID* denotes the event sequence or trace of a given workflow. Another way to refer to an event sequence is by using a tuple <*Workflow ID*, *Case ID*>. The *Task ID* is a key for the current task in the sequence. The *Action ID* refers to a unique action, given a task. Note that this value is not unique when you consider the complete workflow model. Every task within a workflow must have at least one action, and ID's of these actions all start at 1. Of course, it would be better if this ID field was unique within the complete model, but that is not how these workflow logs are set up at AFAS.

Then we also have two fields *Task Description* and *Action Description* which represent the actual meaning or reasoning behind a task or action in the workflow. These descriptions are set by the creator of the workflow. Since we are interested in the context of such tasks or actions, these fields will become crucial for determining this context. Finally, the last two fields *Start Time* and *End Time* contain the time stamps of this action. When you see a workflow as a finite state machine, the moment that the current task is entered is the start time of this state, whereas the moment that the action is triggered is the end time of this state. Using this information, we can derive the order of events in a sequence.

Using this information, we can derive that the log contains two event sequences, shown in Table 7.2.

TABLE 7.1: An example workflow event log.

Workflow ID	Case ID	Task ID	Task Description	Action ID	Action Description	Start Time	End Time
1	82	1	Approve Manager	1	Accept	2-1-2018 09:10	2-1-2018 11:15
1	82	2	Approve Admin	1	Accept	2-1-2018 11:15	2-1-2018 11:42
1	83	1	Approve Manager	2	Reject	2-3-2018 08:36	2-3-2018 08:49
1	83	3	Adjust application	1	Resend	2-3-2018 08:49	2-3-2018 09:11
1	83	1	Approve Manager	1	Accept	2-3-2018 09:11	2-3-2018 09:17
1	83	2	Approve Admin	2	Reject	2-3-2018 09:17	2-3-2018 09:33
1	83	3	Adjust Application	1	Resend	2-3-2018 09:33	2-3-2018 09:53
1	83	1	Approve Manager	1	Accept	2-3-2018 09:53	2-3-2018 10:04
1	83	2	Approve Admin	1	Accept	2-3-2018 10:04	2-3-2018 10:11

TABLE 7.2: Sequences of workflow with ID 1, based on the workflow log in Table 7.1.

Case ID → Step in workflow↓	82	83
1.	Approve Manager - Accept	Approve Manager - Reject
2.	Approve Administrator - Accept	Adjust Application - Resend
3.		Approve Manager - Accept
4.		Approve Admin - Reject
5.		Adjust Application - Resend
6.		Approve Manager - Accept
7.		Approve Admin - Accept

However, we do start searching workflow patterns directly from workflow logs like described above. A translation needs to be made from workflow logs to workflow models. Now that we have seen what kind of data we will be working with, let's take a look at actions which translate this such logging data into workflow models.

7.2 From Workflow Logs to Models

First of all, the workflow logs need to be ordered. This ordering is done in the following way:

$$\text{Workflow ID} \downarrow \quad \text{Case ID} \downarrow \quad \text{Start Time} \downarrow \quad \text{End Time} \downarrow \tag{7.1}$$

Applying this ordering results in a table where the records are ordered based on their workflow- and case ID. At last, events within a workflow instance are chronologically ordered.

One of the problems with the set of workflow logs we use, as mentioned in Chapter 2.1, is that different versions of a workflow can be used throughout time. This means that different descriptions for tasks/actions can be used, or tasks/actions that previously existed in the model may have been removed in later versions. Unfortunately, an event log does not refer to a specific version of a workflow model. This causes to be problematic when mining for relations between tasks and actions. To give an example, recall the example workflow in Figure 1.1 and suppose that, at some point in time, the description of the first task "Approve" gets changed to "Check". The result of this change is that all upcoming instances of the workflow log a task called "Check", whereas we before logged "Approve". How do we determine that instances using "Approve" and instances using "Check" use a different

workflow model? We do this by iterating over all instances of a workflow, keeping track of a dictionary where we store the description of every task as a tuple $<$ Task ID, Task Description $>$. By starting with the most recent instance, we collect the latest workflow model. When an instance uses a different description for a task, given the Task ID, then we conclude that its *workflow version* is different. The same goes for changed actions in a model. This way, we recognize which instances belong to the same workflow version. This recognition is very important since we want to derive the correct model and this can only be done when we know which traces are based on this model. Using logs which use a different model will, of course, influence our mined workflow model in a negative way, which is something we try to avoid by applying the steps explained above.

As explained in Chapter 2, we do not know the corresponding workflow model of a log beforehand. Although the example log from Table 7.1 is based on the model in Figure 1.1, we do not have access to the corresponding model. Because of this, we need to convert a workflow log into a workflow model before being able to search for useful patterns in this model. The model discovery is done by applying the Inductive Miner, or the Inductive Miner infrequent. Both algorithms are discussed in Chapter 6.2.

Since 2011, the standard event log format is XES Buijs (2010), which stands for *eXtensible Event Stream*. This format is XML-based and many tools, like ProM 6+ Verbeek et al. (2011), and algorithms, like the Inductive Miner Leemans et al. (2013), are based on XES files as their input. The logs like Table 7.2 can easily be converted to this format.

7.3 Workflow Pattern Matching

Before going into the algorithm used for our research, we need to define when a given pattern P is said to be matched by a given workflow T.

Definition 16 (Pattern matching rule)**.** We say that a given workflow pattern P is contained in a given workflow model T (both described as process trees) iff:

1. every event $p \in P$ is similar to an event $t \in T$ according to Equation 7.3, and
2. the structure of P found in T should still meet the requirements of an induced/embedded subtree as stated in Definitions 9 and 10.

Given a process tree T and a workflow pattern P, the algorithm tries to find occurrences of P in T.

7.3.1 Applying Word2Vec

During workflow pattern matching, we make use of Word2Vec (see Chapter 6.1 for more details). To elaborate further on this technique, see Figure 7.1. For instance, *juice* and *apple* are quite close to each other. This makes sense because they are both fruits. However, the distance between *apple* and *car* is much bigger. From this, we learn that these words are unrelated.

To put Word2Vec in our perspective, terms used in events of workflow models we study should behave the same way. For instance, we expect that 'approval' and 'agreement' will lie pretty close to each other in the vector space. Also note that

the words *accept* and *reject* are closely positioned in the Word2Vec model of Figure 7.1. You might want to set similarity between antonyms like these to 0 because their meaning is the opposite of each other. However, since antonyms often occur in the same context, Word2Vec 'learns' that these terms are quite similar. Our solution for this problem is explained in Chapter 7.3.2.

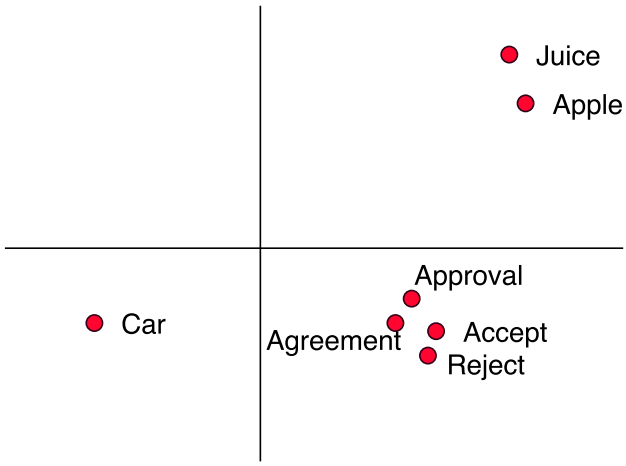

FIGURE 7.1: An example vector space.

A variation of Word2Vec is called *Sent2Vec* (Pagliardini et al., 2017). Sent2Vec is a way to train distributed representations of sentences. In short, Sent2Vec uses an aggregate function to determine the vector of a sequence of words. For example, you can define the final vector as the average vector when applying Word2Vec on each word in the sentence. So when looking at our example sentence s ="Most of the time, apples are green", you could for instance define the position of s as

$$v(s) = \frac{1}{7} * [v(\text{most}) + v(\text{of}) + v(\text{the}) + v(\text{time}) + v(\text{apples}) + v(\text{are}) + v(\text{green})]. \quad (7.2)$$

However, since task and event descriptions in our dataset mostly consist of a few keywords instead of a sentence, we chose to apply Word2Vec instead of Sent2Vec.

7.3.2 Normalization of Terms And Their Similarity

For normalization purposes, we apply the following steps to pre-process each sentence s:

1. Convert s to lower-case.
2. Strip s of punctuation characters and numerical characters.
3. Remove dutch stopwords in s, using Natural Language Toolkit (NLTK) (Loper and Bird, 2002) in Python.

We could use a simple formula to determine whether a given workflow pattern P is contained in a given process tree P. Given a sentence representing event $p \in P$ and given a sentence representing event $t \in T$:

$$AreSimilar(s_p, s_t) = \max[\sum_{w_p \in s_p} \sum_{w_t \in s_t} W2V(w_p, w_t)] \geq \pi \quad (7.3)$$

Where $W2V(w_p, w_t)$ stands for the cosine similarity function of Word2Vec and π is a given minimum similarity threshold, where $0 \leq \pi \leq 1$. When this threshold value is met, then we conclude that the given events are similar. Although Equation 7.3 is easy to understand, we want to extend this function somewhat further.

We have noticed that Word2Vec does not make a clear distinction between antonyms. For example, using the *sonar-320* corpus, the term *afkeuren* is placed within the top 10 most similar terms for *goedkeuren*. Since these terms are often placed in the same context, Word2Vec sees the terms as quite similar.

To fix this problem, we use a Dutch website called mijnwoordenboek.nl. This website can be used to extract synonyms and antonyms of a given word[1]. We used Python packages *urllib2* and *Beautiful Soup 4* to be able to query and translate XML pages from *mijnwoordenboek.nl*. We use this website for two reasons:

- We expand the set of similar terms of a given word w_p with the synonyms retrieved, denoted by $Synonyms(w_p)$. When matching w_p to a synonym w_t, we set their similarity score to 1. This is because we assume that this set of synonyms consists of similar words.
- We also collect a set of antonyms of w_p, denoted $Antonyms(w_p)$. Since the list of antonyms retrieved from *mijnwoordenboek.nl* often has a smaller size compared to synonyms, we extend this list by obtaining all synonyms of the antonyms found. This makes $Antonyms(w_p)$ ultimately the set of $a_i \in antonyms(w_p) \cup synonyms(a_i)$. Whenever we discover that s_t contains an antonym of any $w_p \in s_p$, we assume that the two sentences s_p and s_t cannot be similar.

When computing the similarity between two sentences w_p and w_t, we now use sets of synonyms and antonyms, along with their purpose as explained above. The extended version of the similarity function is given by Functions 3 and 4. We simply use Function 3 to check whether the similarity between two sentences s_p and s_t is acceptable, but the actual similarity score gets determined in Function 4.

Function 3: Function which checks the similarity between two sentences.

```
1  function AreSimilar (s_p, s_t)
   Input  : A pattern sentence s_p, a tree sentence s_t.
   Output: A boolean value.
2  score = Similarity(s_p, s_t);
3  if score ≥ π then
4  |   return True;
5  end
6  return False;
```

In general, we compute the similarity between each combination of $w_p \in s_p, w_t \in s_t$ and keep track of the highest score (lines 3-18) and finally check whether this score $\geq \pi$ (lines 15-17). Note that the approach up to this point is similar to Equation 7.3. In lines 5-17, we check the similarity between a given pattern word w_p and w_t. At line 6, we check whether w_t is an synonym of w_p. In this case, we set the maximum score to 1. Although this is the highest possible score, we continue searching. The reason why is given by lines 9-10. Whenever we encounter an antonym of w_p, we immediately return 0 as the similarity score. The reason why is that we assume that the two sentences s_p and s_t cannot be similar whenever they contain one or more

[1]The website returns synonyms and puzzle variations of a given term. We only use the synonyms and ignore the list of puzzle words.

antonyms of each other.

Function 4: Function which checks the similarity between two sentences.

```
1  function Similarity (s_p, s_t)
   Input  : A tree sentence s_t, a pattern sentence s_p.
   Output: A boolean value.
2  maxScore = 0;
3  for w_p ∈ s_p do
4      for w_t ∈ s_t do
5          score = 0;
6          if w_t ∈ Synonyms(w_p) then
7              score = 1;
8          end
9          if w_t ∈ Antonyms(w_p) then
10             return 0;
11         end
12         else
13             score = W2V(w_p, w_t);
14         end
15         if score > maxScore then
16             maxScore = score;
17         end
18     end
19 end
20 return maxScore;
```

7.3.3 Finding Pattern Matches: A Graphical Example

The general idea is to 'walk' over all nodes $t \in T$. This is done in post-order[2] for matching with the root node $p_1 \in P$. However, for the purpose of explaining how patterns are being matched, we ignore all matching steps executed before the root note of P is matched, making the explanation shorter and easier to understand. The order of matching the remaining nodes $p_{>}1 \in P$ is done in a depth-first approach. Thus, when $p_1 \in P$ is matched, we search whether its child nodes are also matched etc. When some descendant of $p_1 \in P$ cannot be matched, we move on with matching p_1.

So we start with matching $p_1 \in P$, which is the root node of P. When p_1 is matched, we continue with searching for p_1's child nodes in P etc. To visualize this approach, Figure 7.2 shows a pattern considering a start- and finish event. Furthermore, pattern matching is done differently when we allow induced subtrees or allow embedded subtrees to be proper occurrences of P. Figures 7.3 and 7.4 present process trees, where the former shows an induced match and the latter shows an embedded match. For reference purposes, we refer to a node from T as t and a node from P as p. We have added a number as the identifier to every node in the process trees, and refer a node t with id i as t_i.

[2]We go into more depth on this ordering in Chapter 7.3.4.

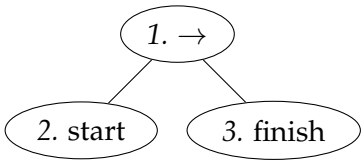

FIGURE 7.2: A sequence pattern.

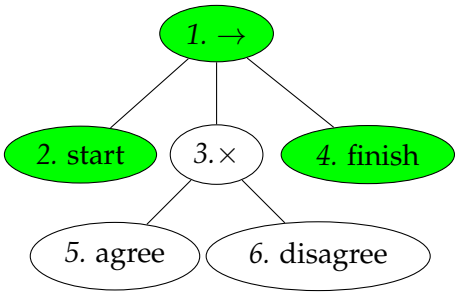

FIGURE 7.3: A process tree T_1 containing the pattern from Figure 7.2 as an induced subtree.

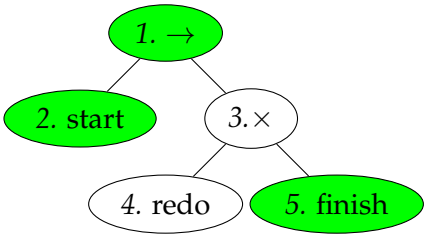

FIGURE 7.4: A process tree T_2 containing the pattern from Figure 7.2 as an embedded subtree.

Step	P node	T_1 node	Match
1.	p_1	t_1	✓
2.	p_2	t_2	✓
3.	p_3	t_3	✗
4.	p_3	t_4	✓

TABLE 7.3: Steps of searching in tree T_1 given in Figure 7.3 for an induced occurrence of Figure 7.2.

Step	P node	T_2 node	Match
1.	p_1	t_1	✓
2.	p_2	t_2	✓
3.	p_3	t_3	✗
4.	p_3	t_4	✗
5.	p_3	t_5	✓

TABLE 7.4: Steps of searching in tree T_2 given in Figure 7.4 for a embedded occurrence of Figure 7.2.

The root node p_1 of the example pattern P is $\rightarrow$. We start by searching for a node $t \in T_1$ that matches p_1, where T_1 is given in Figure 7.3. Also, we start off with finding induced matches. Recall that P is an induced subtree of T whenever the child-parent relations of P are present in T. The matching function is easy to understand; whenever a node is an operator, it only matches with nodes that are the same operator. When node p is not an operator - thus represents an event - we conclude whether it matches with t or not using Word2Vec (see the similarity function in Function 3). However, for the examples explained below we only consider exact matches of terms. The according steps for pattern matching are shown in Tables 7.3 and 7.4.

The first step is searching for a node that matches p_1. As you can see, t_1 also is a $\rightarrow$ operator. Thus, t_1 matches p_1 and we move on to the next node in P, which is p_2. Like p_2 does t_2 represent the event 'start'. Now we continue with p_3. Node t_3 is an operator so does not match p_3. t_4 finally does match p_3. Since there are no more nodes in P which we have not considered, we are done and found induced subtree $\{t_1, t_2, t_4\} \in T_1$.

Considering T_2 in Figure 7.4, we find no induced matches for P. The reason why there are no induced matches in T_2 is because the parent-child relation of p_1 and p_3 cannot be met in T_2. Although t_1 matches p_1 and t_5 would match p_3, the parent of

t_5 is not t_1, but t_3. Recall that the parent-child relation is relaxed in the definition of embedded subtrees. This means that $\{t_1, t_2, t_5\} \in T_2$ is an embedded match of P.

7.3.4 Finding Pattern Matches: A Basic Framework

To go into more detail, we have written this algorithm into pseudo code. The general framework is shown in Figure 7.5. This Figure represents the relationships between the different Functions. Function 5 is the main Function and makes direct use of Functions 6 and 9 etc. Note that the tree splits into two subtrees. This is because the algorithm distinguishes induced matching (Functions 6, 7, 8) and embedded matching (Function 9). While reading the remainder of this Chapter, it is recommended to use Figure 7.5 as a way to keep track of the location of a Function in the framework.

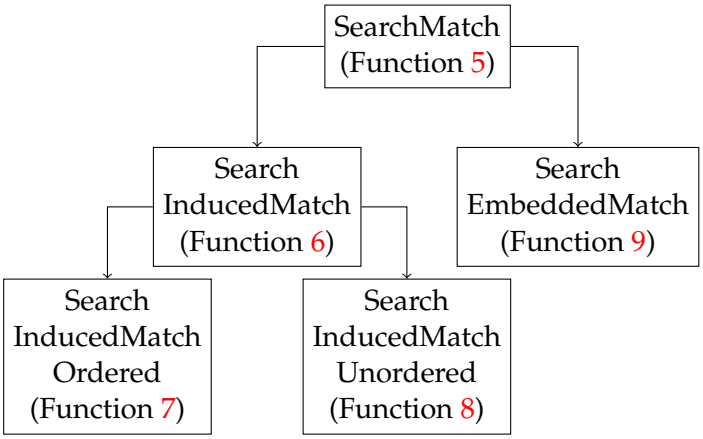

FIGURE 7.5: Pattern Matching Framework tree.

The algorithm starts at Function 5. In this function, we try to find a node t in T that is similar to the root node p of P (line 7). If that is the case, we need to consider the subtrees of p and t and check whether they also match. Matching the subtrees of t and p is done in Functions 6 and 9 (line 9, 12). Eventually, using post-order traversal, each node in T gets visited as long as no match of P is found so far (see line 3). Post order traversal starts at leaf nodes and works its way up to internal nodes when all leaf nodes are explored etc. Using Figure 7.6, the post order of its nodes is $\langle t_3, t_4, t_2, t_1 \rangle$. The reason for using post order traversal is that we prefer to find embedded matches that uses as few as possible nodes. Taking Figure 7.6 as an example and re-using Figure 7.2 as our pattern, then we would like to discover match $M_1 = < p_1, t_2 >, < p_2, t_3 >, < p_3, t_4 >$. Note that using breadth first traversal (or depth first traversal), match $M_2 = < p_1, t_1 >, < p_2, t_3 >, < p_3, t_4 >$ would be discovered, which is less convenient in our opinion since we prefer a directly connected match over an indirectly connected match. M_1 is a directly connected match because all t nodes in M_1 are directly connected with each other. Note that this is not the case for M_2.

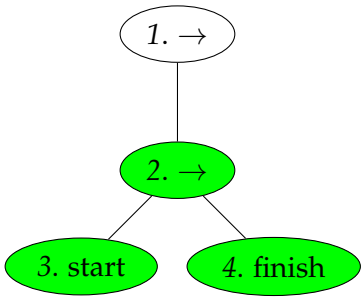

FIGURE 7.6: A process tree which shows why using post order traversal in Function 5 is a good approach.

We make a distinction between the case whether we want to find induced subtrees or embedded subtrees (using boolean variable *isInduced*) in lines 8-13. Although both cases have a somewhat similar code, there are some differences which are important to note.

Function 5: Main function that returns an induced or embedded subtree from a given process tree that matches with a given pattern.

```
1  function SearchMatch (T, P, isInduced)
     Input  : A process tree T, a process tree P, boolean value induced stating
              whether we must only consider induced subtrees as matches or
              consider embedded subtrees.
     Output: A pattern match from P in T if possible. This match is an induced or
              embedded subtree, depending on the value of isInduced.
2  p = P.root;
3  nodeList = T.GetNodesInPostOrder();
4  while |nodelist| > 0 do
5      t = nodelist.pop;
6      result = null;
7      if AreSimilar(t, p) then
8          if isInduced then
9              result = SearchInducedMatch(t, p);
10         end
11         else
12             result = SearchEmbeddedMatch(t, p, []);
13         end
14         if result ≠ null then
15             return result;
16         end
17     end
18 end
19 return null;
```

Induced Matching

We like to say that induced matching is the stricter variant of the two matching procedures that we apply in our study since all induced matches are a proper subset of embedded matches (see Definition 9 and 10). Function 6 determines whether or not the given pattern node p has a specific ordering in its children. Recall that the ordering in child nodes is only relevant whenever p is a sequence operator ($\rightarrow$) or a

loop operator (↻). Eventually, a call to Function 7 or 8 is made.

Function 6: Function that checks whether the pattern is ordered or unordered. This property depends on the type of the current root node.

```
1 function SearchInducedMatch (t, p)
  Input  : A process tree node t, a pattern node p.
  Output: An induced pattern match of p in t if there exists any.
2 if p ∈ [→, ↻] then
3 |   return SearchInducedMatchOrdered(t, p);
4 end
5 else
6 |   return SearchInducedMatchUnordered(t, p);
7 end
```

Function 7 processes patterns where the given p has ordered children and Function 8 processes patterns where the given p has unordered children. Function 7 follows these steps:

1. Return when p has no child nodes (line 2-4).
2. For every child node p_c of p, search a matching child node t_c of t (lines 7-19). Keep in mind that the order of the child nodes needs to be maintained during this matching process. Variable *startIndex* keeps track of child nodes which are still open for further matching (see line 8). Suppose that t has 8 child nodes $\{t_1, \ldots, t_8\}$, p has 3 child nodes $\{p_1, p_2, p_3\}$ and p_1 matches t_3, then only t_4 up to t_8 may be used for matching p_2.
3. Besides matching child nodes t_c and p_c, we also match their corresponding descendants. This results in a recursive call at line 11. When the recursive call returns nothing, we know that the corresponding descendants of t_c and p_c do not match (lines 11-16).
4. When we have found a match of p and its descendants, we return all matching nodes (line 20-22).
5. return *null* when no match of p in t has been found (line 23).

Function 7: Function that tries to find a given induced pattern where the order of p's children matters.

```
1  function SearchInducedMatchOrdered (t, p)
   Input  : A process tree node t, a pattern node p.
   Output: An induced match of p in t if there exists any.
2  if |p.children|== 0 then
3  |  return [⟨t, p⟩];
4  end
5  matches[];
6  startIndex = 0;
7  for p_c ∈ p.children do
8  |  for index ∈ range(startingIndex, |t.children|) do
9  |  |  t_c = t.children[index];
10 |  |  if AreSimilar(t_c, p_c) then
11 |  |  |  subtreeMatch = SearchInducedMatch(t_c, p_c);
12 |  |  |  if subtreeMatch ≠ null then
13 |  |  |  |  matches.add(subtreeMatch);
14 |  |  |  |  startIndex = index + 1;
15 |  |  |  |  break;
16 |  |  |  end
17 |  |  end
18 |  end
19 end
20 if p.descendants ∈ matches then
21 |  return [⟨t, p⟩, matches];
22 end
23 return null;
```

Function 8 shows how we can find induced matches where the given p node has unordered children. In general, this Function looks quite similar to Function 7. The only difference is that p's children are unordered, which makes the matching process of child nodes easier. By simply iterating over the children of p and trying to match a child from t, we search for a pattern match (lines 6-16). Note that we need to check whether t_c does not already match a previous child node of p (condition in line 8), to prevent that we match on some t_c multiple times. Whenever a match is successful, we stop searching for the current p_c and continue with the next (line 10-13).

Function 8: Function that tries to find a given induced pattern where the order of p's children matters.

```
1  function SearchInducedMatchUnordered (t, p)
   Input  : A process tree node t, a pattern node p.
   Output: An induced match of p in t if there exists any.
2  if |pChildren|== 0 then
3  |  return [⟨t, p⟩];
4  end
5  matches = [];
6  for p_c ∈ p.children do
7  |  for t_c ∈ t.children do
8  |  |  if t_c ∉ matches ∧ AreSimilar(t_c, p_c) then
9  |  |  |  subtreeMatch = SearchInducedMatch(t_c, p_c);
10 |  |  |  if subtreeMatch ≠ null then
11 |  |  |  |  matches.add(subtreeMatch);
12 |  |  |  |  break;
13 |  |  |  end
14 |  |  end
15 |  end
16 end
17 if p.descendants ∈ matches then
18 |  return [⟨t, p⟩, matches];
19 end
20 return null;
```

Embedded Matching

The second matching procedure is called embedded matching. This type of matching is less strict than induced matching and uses a wider range of possible matches. As Figure 7.5 shows, embedded matching is described in just one Function 9. Recall that we do not take the ordering of sibling nodes into account for embedded matching, unlike induced matching (see Definitions 9 and 10). Embedded matching can be seen as a relaxed variant of induced matching.

The main difference between induced matching and embedded matching is the set of nodes which is eligible for matching a certain pattern node p of a given pattern tree P. For induced matching, matches of child nodes of p must be direct child nodes of t. For embedded matching, matches of child nodes of p must only have to be descendants of t. Since the parent-child relation constraint for induced matching is relaxed into an ancestor-descendant relation constraint for embedded matching, this extends the search for possible matches of a given pattern node p. This extension leads to lines 22-31 in Function 9. This part of the Function enables the algorithm to continue searching for a match for t whenever p and t itself do not match. This possibly results in indirect matches, meaning that nodes $t \in T$ which are part of an embedded match of a given P are not always directly linked to each other. Recall that the embedded match of Figure 7.2 in Figure 7.4 gives an example of such an indirect match.

The problem of finding embedded matches comes with the following two subproblems:

1. Keeping track of all matched nodes t is more complicated than for induced matching. Since we do not want different $p_1, p_2 \in P$ to be matched to the same node $t \in T$, we need to keep track of all nodes $t \in T$ which have already been matched to some $p \in P$. Like we have seen with induced matching, the list *matches* does this administration. However, the *matches* variable does not keep track of any previous matches that are discovered earlier during the matching procedure. With induced matching, this is no problem since a match is an induced match if and only if it results in a direct match. Though for embedded matches this is not the case and thus, we need to know any previously discovered matched nodes $t \in T$ to avoid using a node $t \in T$ multiple times. For this reason, we added a third parameter to Function 9, called *previousMatches*, which is a list of nodes $t \in T$ that are already used for the embedded match. Along with the variable *newMatches* we know exactly which set of $t \in T$ is currently used for the embedded match.

2. Siblings $p_1, p_2 \in P$ can be matched to nodes $t_1, t_2 \in T$ where t_1 and t_2 are descendants of each other becomes hard. This matching behavior is not correct, according to Definition 16. To avoid this problem, we check whether a given $t \in T$ is an ancestor of another node $t' \in T$ that is already used for matching (line 2). When this is the case, we may not use this node t for matching.

Function 9: Function that tries to find a embedded pattern of p and its descendants.

```
1  function SearchEmbeddedMatch (t, p, previousMatches)
   Input  : A process tree node t, a pattern node p, a list of other matching nodes
            of P and T.
   Output: A embedded pattern match of p in t if there exists any.
2  if AreSimilar(t, p) ∧ ¬IsAncestor(t, previousMatches) then
3      if |p.children|== 0 then
4          return [⟨t, p⟩];
5      end
6      newMatches = [];
7      for p_c ∈ p.children do
8          for t_c ∈ t.children do
9              if t_c ∉ (newMatches ∪ previousMatches) ∧ AreSimilar(t_c, p_c) then
10                 subtreeMatch =
                     SearchEmbeddedMatch(t_c, p_c, newMatches ∪ previousMatches);
11                 if subtreeMatch ≠ null then
12                     newMatches.add(subtreeMatch);
13                     break;
14                 end
15             end
16         end
17     end
18     if p.descendants ∈ newMatches then
19         return [⟨t, p⟩, newMatches];
20     end
21 end
22 for t_c ∈ t.children do
23     if t_c ∉ previousMatches then
24         match = SearchEmbeddedMatch(t_c, p, previousMatches);
25         if match ≠ null then
26             return match;
27         end
28     end
29 end
30 return null;
```

7.4 Extending The Basic Matching Algorithm

The previous section describes a basic algorithm to determine whether a given process tree P is contained by another process tree T. This algorithm only searches one match of P in T if it exists. However, the algorithm can be extended to find multiple matches. Furthermore, taking the similarity scores of other possible matches into account can increase the overall similarity of a match found. Let us study these extensions in more detail.

7.4.1 Finding Multiple Matches

Besides obtaining a valid pattern match, we are interested in all occurrences of the pattern in a workflow model. For this problem, we need to define when two matches are distinct. Recall that leaf nodes in a process tree are nodes that represent workflow events, whereas internal nodes are operators that define the order in which events can be executed. We do not want to use the same event in multiple matches, thus matches M_1 and M_2 are said to be distinct if and only if they have distinct sets of leaf nodes. A more formal definition given below:

Definition 17 (Distinct Matches)**.** Given a pattern P and a tree T, a match M_1 is distinct to match M_2 if and only if leaf nodes $A \in M_1$ and leaf nodes $B \in M_2$ are distinct sets, i.e. $A \cap B = \varnothing$.

Based on Definition 17, you can find all distinct matches of P in T in the following way:

1. Find a match M using the algorithm described above.
2. Remove all leaf nodes $t \in T$ that are used in M, from T. This removal step on T leads to T'.
3. Repeat steps 1 and 2 with T' and P until you find no more matches.

Note that an event node in a process tree cannot have any child nodes. Furthermore, an operator node can have many child nodes, from which many may be matched to a given P. To avoid missing such matches, we only remove leaf nodes (events) from T in step 2.

7.4.2 Finding the Highest Scoring Pattern

Another extension is the retrieval of the match with the highest overall score. We define the overall score of a match M of a given pattern P in tree T:

$$Score(M) = \frac{\sum_{m_i \in M} Similarity(m_i)}{|M|}, \tag{7.4}$$

where a pattern match M consists of node matches $m_i =< p_i, t_i >, p_i \in P, t_i \in T$, $|M|$ stands for the total number of node matches $m \in M$ and *Similarity* is defined in Function 4. In other words, we compute the average score of each node match $m \in M$.

Having defined the overall similarity score of a match, we can define the optimal match as follows:

Definition 18 (Optimal match)**.** Given a set of pattern matches $\mathcal{M}$ of pattern P in tree T, then match M is the optimal match in $\mathcal{M}$ if and only if M has the highest overall score of all $M_i \in \mathcal{M}$.

Note that you would need to consider all possible pattern matches to be able to find the optimal match. However, the algorithm as described in Chapter 7.3.4 does not work this way. Thus, it cannot guarantee to find an optimal pattern match. Although it does not determine all possible matches, here is a heuristic approach that we have added to the algorithm described to find a match that is closer to the optimal match.

Whereas the algorithm as discussed in Chapter 7.3.4 stops searching for a match of a given pattern node p once it has found any acceptable match, it should continue searching for a match with a higher score. Let us explain an approach through an example. We define pattern tree P as Figure 7.8, data tree T as Figure 7.7 and set π to 0.7. The search for an induced match of P in T is given step-by-step in Table 7.5.

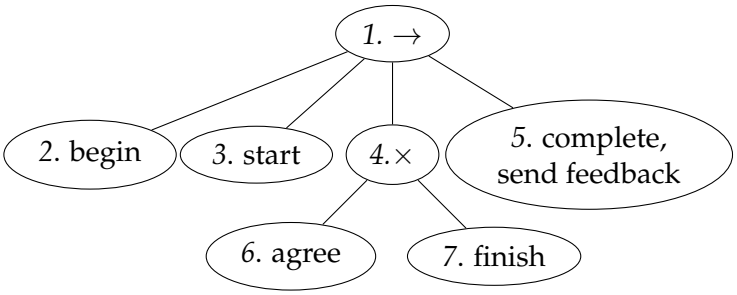

FIGURE 7.7: Another example of a process tree.

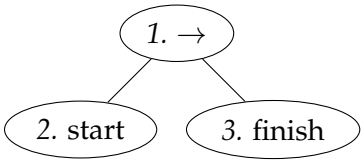

FIGURE 7.8: A simple pattern.

Step	P node	T node	Score	Match
1.	p_1	t_1	1	✓
2.	p_2	t_2	0.8	✓
3.	p_2	t_3	1	✓
4.	p_3	t_4	0	✗
5.	p_3	t_5	0.7	✓

TABLE 7.5: Steps made when searching for an induced match of P in Figure 7.8.

- Recall that a match of the root node $p_1 \in P$ is searched through post order traversal (see Function 5, line 3) in T. This actually leads to a similarity check of p_1 with $t_2, t_3, t_6, t_7, t_4, t_5$ (which do not lead to a match) and finally t_1. Since these steps are somewhat irrelevant for making our point in this case, we have only included matching step $< p_1, t_1 >$ in Tables 7.5 and 7.6.

- Now it is time to search for a match $t_i \in T$ with p_2. Note that we find an acceptable match $< p_2, t_2 >$ (step 2), since its score of 0.8 is higher than 0.7. However, we want to know whether there is a match for p_2 that has an even higher score than 0.8. This is the case for the match $< p_2, t_3 >$ with a score of 1. Because 1 is the highest score possible, we are now finished with p_2 and move on to p_3.

- The search for a match with p_3 starts with t_4 instead of t_2, because P is an ordered tree and t_3 is already matched to p_2. This means that t_2 (and t_3 of course) cannot longer be used for this matching (as is described in Function 7). Since the score of $< p_3, t_4 >$ is lower than π, we move on to $< p_3, t_5 >$. This latter match is acceptable, which leaves us with an induced match $M_{ind} = \{< p_1, t_1 >, < p_2, t_3 >, < p_3, t_5 >\}$. The overall score is $\frac{1+1+0.7}{3} = 0.9$.

Step	P node	T node	Score	Match
1.	p_1	t_1	1	✓
2.	p_2	t_2	0.8	✓
3.	p_2	t_3	1	✓
4.	p_3	t_2	0	✗
5.	p_3	t_4	0.2	✗
6.	p_3	t_6	0.3	✗
7.	p_3	t_7	1	✓

TABLE 7.6: Steps made when searching for an embedded match of P in Figure 7.8.

Table 7.6 shows the matching steps when applying embedded matching. Note that embedded matching leads to more matching steps than induced matching. This is because of two reasons:

1. Whereas induced matching only considers direct child nodes as possible matches for a given pattern node $p \in P$, embedded matching considers all descendants of $t \in T$ (see Definition 16). In this example, $t = t_1$. The child nodes of t_1 are t_2, t_3, t_4, t_5. These are the only set of nodes which is used for induced matching, considering p_2 and p_3. However, embedded matching also considers nodes t_6 and t_7 (shown in steps 6 and 7).

2. Unlike induced matching, embedded matching does not take the order (if an order exists) of nodes in a process tree into account. This is shown in the fact that the embedded matching case contains the matching $< p_3, t_2 >$ (see step 4) but induced matching does not.

The embedded match found is $M_{emb} = \{< p_1, t_1 >, < p_2, t_3 >, < p_3, t_7 >\}$ with an overall score of $\frac{1+1+1}{3} = 1$. Note that M_{emb} is different from M_{ind} and also has a higher score.

We can implement the idea of keeping track of the best match in four Functions, namely Functions 5, 7, 8, 9. For the extended pseudo code, I would like to refer you to Appendix A.

7.5 The Ingredients of Our Tool

Since we want to apply these techniques on a large scale, we developed a program that is able to apply the techniques mentioned - given a collection of workflow logs and workflow pattern - and deliver a set of workflow models containing this pattern. The repository, including an installation- and user guide of the workflow tool can be found on GitHub[3] and mainly consists of three projects. A high-level overview is shown in Figure 7.9. Since this thesis is built on a case study for AFAS, this tool is also part of the result of this thesis.

- A Windows Presentation Foundation Application (C#), called *WorkflowPatternFinder*. This project is mainly created for UI purposes.

- A Windows Console Application (C#), called *WorkflowEventLogProcessor*. This project is mostly used for converting raw workflow logs as explained in Chapter 7.1. For the conversion of CSV files to XES files, we use a Java tool[4]. This

[3] https://github.com/DStekel3/WorkflowPatternFinder
[4] https://github.com/DStekel3/CSV-to-XES

project also takes care of the interaction with other applications, like ProM and Python scripts. ProM is used through its command line approach, which is explained in a blog post[5]. As is needed for this approach, we write a couple of ProM scripts

- A Python project, called *Gensim*. As it is named after the Python Word2Vec package[6], this project mainly consists of scripts where Word2Vec functions need to be applied; finding patterns in process trees or obtaining the most similar terms of a given word.

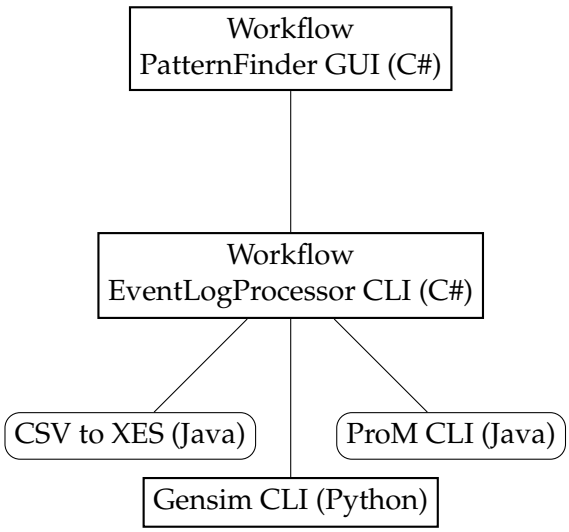

FIGURE 7.9: Overview of different components of the workflow tool[3]. Connected parts are directly interacting with each other. We slightly adjusted the rounded projects from an existing project, whereas we created the squared projects from scratch.

[5] https://dirksmetric.wordpress.com/2015/03/11/tutorial-automating-process-mining-with-proms-command-line-interface/

[6] https://radimrehurek.com/gensim/

Chapter 8

Results

In this chapter, we study variations of the approval pattern. According to AFAS, this pattern plays an important role in workflow models used by customers. For the experiments, we used a dataset from AFAS as described in Chapter 2. For Word2Vec, we used a model which is trained on a Dutch corpus named *SoNaR* (Tulkens et al., 2016), (Oostdijk et al., 2013). This corpus consists of a large set of disparate resources, like newspapers, letters, and articles. Such a wide variety is preferred because it results in a very generic model which captures common Dutch words the best. We study different variations of the approval pattern and describe some notable observations. On the corpus two models are trained, *SoNaR-160* and *SoNaR-320* based on the number of dimensions used. For our experiments, we used *SoNaR-320*, since a model with more dimensions generally outperforms a model with a fewer dimensions (Tulkens et al., 2016). Since the dataset we use consists of workflow logs that use Dutch descriptions, we use some Dutch terms where appropriate and give English translations accordingly. The remainder of this chapter studies occurrences of the approval pattern, as well as some variations on this pattern.

8.1 The Approval Pattern

Let us study the *approval pattern* (recall that we shortly introduced you to this pattern in Chapter 1). The basic structure of this pattern is shown in Figure 8.1. The general idea of this pattern is that some process needs to be approved.

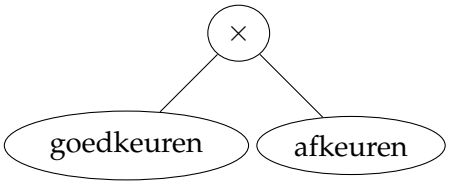

FIGURE 8.1: The approval pattern. Goedkeuren is Dutch for *approve* and afkeuren is Dutch for *refuse*.

Before actually searching for its occurrences, we need to set a similarity threshold π which will be used for classifying whether terms are similar or not (see Function 3).

In general, there are two strategies for setting the value of π:

1. $\pi = 1$. In this case, you only want to consider synonyms of terms used in your pattern and do not want to use any alternatives Word2Vec will give you.
2. $0 \leq \pi < 1$. Besides the use of synonyms, you want to include some room for the algorithm to match the terms used in your pattern with some terms that are suggested by a Word2Vec model (in our case, the *SoNaR-320* model).

Before choosing a value for π, we should study what terms will be used for matching a given word in your pattern. In our case, we can use synonyms given by *mijnwoordenboek.nl* and suggestions made by Word2Vec. Since the set of synonyms or antonyms obtained for a given word can be quite large, we do not present these and only mention somewhere needed. Although Word2Vec can also return a large set of suggestions, we show the most interesting below. These suggestions have a large impact on the choice for π, making it important to study them before setting π. In the resulting list of suggestions, Dutch stop words and antonyms get removed (these steps are described in Chapter 7.3.2).

Using *SoNaR-320*, we retrieve the following terms shown in Table 8.1 when querying the most similar terms to the word *goedkeuren* (Dutch for *approve*). We have classified suggestions that have an italic type as incorrect matches.

Dutch Term	**English Translation**	**Similarity Score**
goedkeurt	approves	0.6849
goedgekeurd	approved	0.6589
instemmen	to agree	0.6340
bekrachtigen	to empower	0.6335
goedkeurde	approved	0.6153
goedkeuring	approval	0.6110
bekrachtigd	enforced	0.5858
budgetrecht	*budgetary laws*	*0.5777*
merstudie	-	*0.5626*
...	...	...

TABLE 8.1: List of similar terms to *goedkeuren*, obtained through Word2Vec.

Table 8.1 shows words that are clearly similar to *goedkeuren*. For *afkeuren*, the resulting terms in Table 8.2 are less obvious. For instance, the word *tolereren* has a different meaning than *afkeuren*. Also the word *goedkeurt* should be filtered out as an antonym, because it is an abbreviation of *goedkeuren*. Since we do not recognize abbreviations of a word, we are not able to recognize that *goedkeurt* is an antonym of *afkeuren*.

Dutch Term	**English Translation**	**Similarity Score**
afwijzen	to decline	0.5791
tolereren	*to tolerate*	*0.5612*
excuserende	*to apologize*	*0.5549*
gedrag	*behavior*	*0.5530*
durft	*dares (to dare)*	*0.5460*
goedkeurt	*approves*	*0.5444*
...	...	...

TABLE 8.2: List of similar terms to *afkeuren*, obtained through Word2Vec.

Still, these tables give us some understanding of what value you want to set as the threshold value π. Since this threshold value is used for every term used in the pattern, you need to be sure to use a threshold value that is high enough for each term. In our case, we see that the similarity scores of the highest scoring

terms of *goedkeuren* (0.61-0.68) starts at a higher range than the scores in the case of highest scoring terms of *afkeuren* (0.54-0.58). In this situation, you should set $\pi \geq 0.57$ to avoid accepting incorrect terms as matches for $afkeuren$, as shown in Figure 8.2. Note that such a value for π also allows the algorithm to choose terms with a similarity score ≥ 0.57 compared with *goedkeuren*.

After applying IM, the total number of matches with the approval pattern in the dataset used is given in Table 8.3.

Type Of Matching	Models	Matches	Distinct Matches	Avg Score
Induced (1)	65	65	47	0.989
Induced (+)		79	57	0.986
Embedded (1)	278	278	159	**0.993**
Embedded (+)		**389**	**220**	0.991

TABLE 8.3: Table showing info about found pattern matches for the approval pattern, using $\pi = 0.57$.

You can immediately see that the number of matches found is much higher for embedded matching and induced matching. Considering embedded matches, the pattern occurs in 278 out of 4731 workflow models. However, only 65 models contain an induced match of the approval pattern. This shows that embedded matching leads to more discovered cases of a given pattern.

Two workflow models that contain the approval pattern are given below. The workflow model in Figure 8.2 has an induced match of the approval pattern, whereas the workflow model in Figure 8.3 has an embedded match. To reduce the amount of space needed for each model, we changed the node style to squared. The underlined words are the terms which lead to this match.

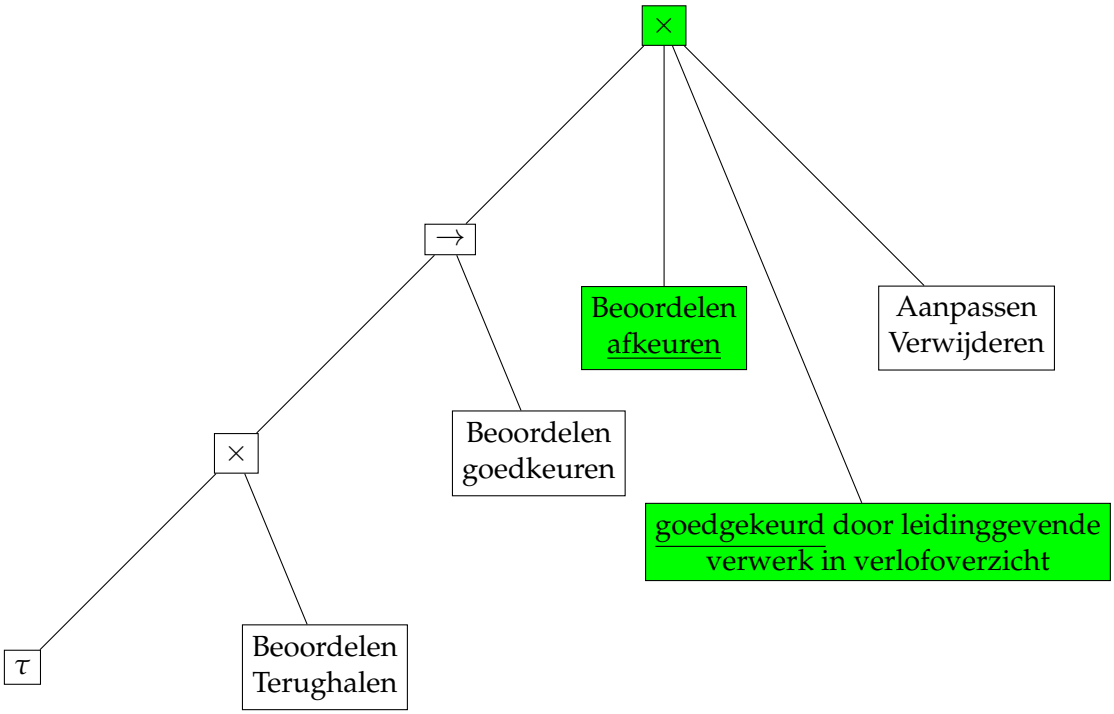

FIGURE 8.2: An induced match of the approval pattern (overall score of 0.87).

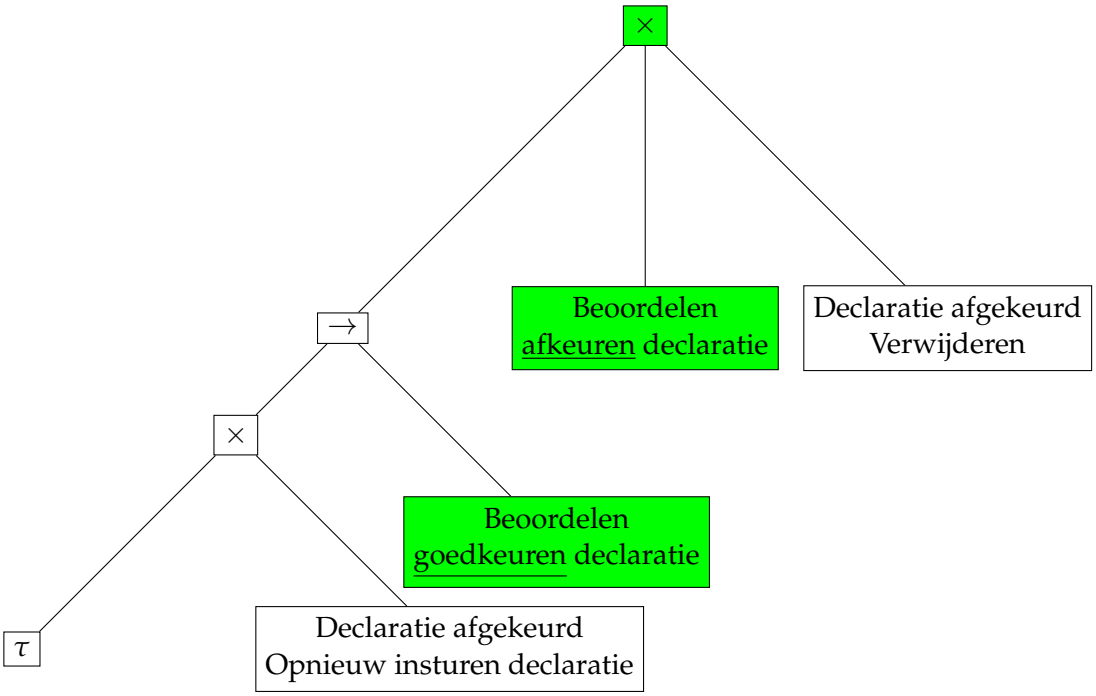

FIGURE 8.3: An embedded match of the approval pattern (overall score of 1).

In fact, almost 90% of the matches found are solely based on the exact words being used (goedkeuren, afkeuren) or one of their synonyms which we obtained from *mijnwoordenboek.nl*. Figure 8.4 shows the number of matches found for each type of matching, based on their overall score. This is something we could expect, considering the number of synonyms used for goedkeuren (21) and afkeuren (5). According to Word2Vec, *SoNaR-320* only has 8 terms which have a similarity score ≥ 0.57 with goedkeuren. Only one word (afwijzen) is acceptable for afkeuren with a score of 0.5791. Furthermore, afwijzen is also retrieved as a synonym, which elevates the similarity score of afwijzen and afkeuren 1. This does not mean that the terms which Word2Vec suggests do not encounter often, but the similarity function as described in Chapter 7.3.2 prefers synonyms over suggestions by Word2Vec.

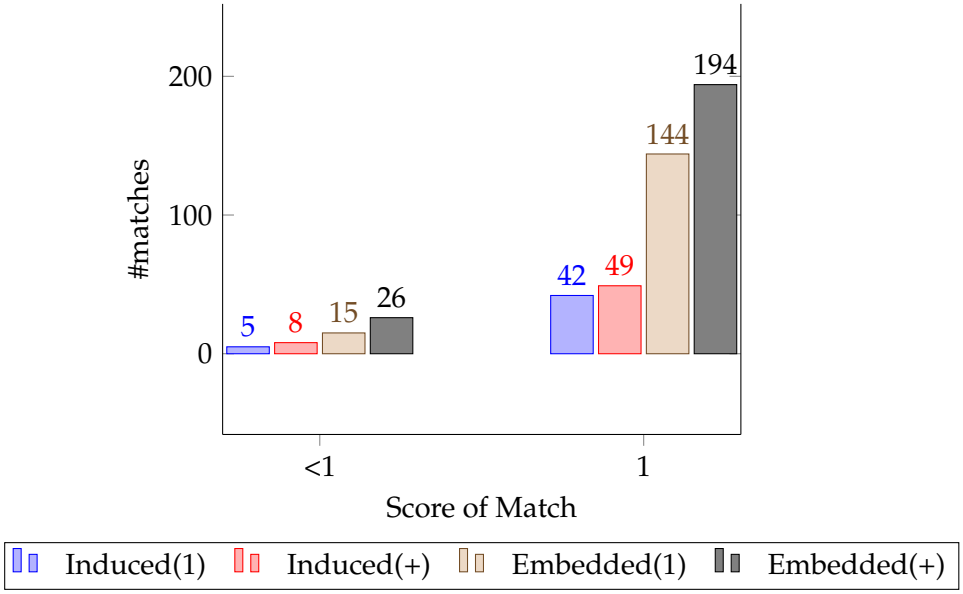

FIGURE 8.4: Bar plot showing the number of distinct matches found, based on their overall score. We distinguish perfect matches (score=1) from imperfect matches (score <1).

To study the behavior of the noise threshold ϵ parameter of IMi, we obtained the following results shown in Table 8.4. This Table includes the same info as Table 8.3, but we experimented with the value of ϵ. The different noise thresholds used are 0.0, 0.2, 0.5 and 0.8. From Table 8.3, we can see that there is only a slight decrease in the number of (distinct) matches when we increase ϵ. This means that the approval pattern occurs often in the dataset because the pattern wouldn't get recognized if it would consist of infrequent events. Table 8.4 also shows that embedded matching can lead to some higher scoring matches, compared to induced matching. However, since all scores are quite similar, this point is quite negligible.

We need to note that the noise threshold would be more effective in filtering infrequent behavior when the number of workflow instances is quite high. However, our dataset also consists of many small workflow logs that only consists of a few instances. We expect that such workflow logs can play a big role in the small decrease in matches instead of a higher decrease.

Type Of Matching	ϵ	Models	Matches	Distinct Matches	Avg Score
Induced (1)	0.0	65	65	47	0.989
Induced (+)			79	57	0.986
Embedded (1)		278	278	159	**0.993**
Embedded (+)			**389**	**220**	0.991
Induced (1)	0.2	40	40	31	0.991
Induced (+)			46	35	0.992
Embedded (1)		274	274	157	**0.994**
Embedded (+)			**384**	**216**	0.992
Induced (1)	0.5	43	43	30	0.994
Induced (+)			49	34	**0.995**
Embedded (1)		272	272	156	0.993
Embedded (+)			**378**	**213**	0.992
Induced (1)	0.8	36	36	28	0.993
Induced (+)			41	31	**0.994**
Embedded (1)		269	269	153	**0.994**
Embedded (+)			**375**	**211**	0.993

TABLE 8.4: Table showing info about found pattern matches for the approval pattern, using π = 0.57. Also, ϵ stands for the noise threshold parameter of the Inductive Miner infrequent variant.

For the rest of the results described in this chapter, we will set the noise threshold ϵ to 0.

When we only consider all distinct embedded(+) matches, we notice that *goedkeuren* is matched to the following terms. They are ordered on the total number of occurrences in a pattern match. It is clear that *goedkeuren* is mostly used. Still, some variants which are suggested by Word2Vec are used as well as some synonyms.

Matching Word	Translation	Type of Match	Score	#Matches
goedkeuren	to approve	Same word	1	175
goedkeuring	approval	Word2Vec	0.6110	20
bevestigen	to confirm	Synonym	1	10
laten	*to let*	Synonym	1	8
goedgekeurd	approved	Word2Vec	0.6589	6
toelaten	to permit	Synonym	1	1

However, we may argue that *laten* does not have the same meaning as *goedkeuren* in the context of approving something in a business process. This shows that our assumption that all synonyms are correct matches is not always correct. In this case, it is possible that the algorithm converts a valid pattern match (using *goedkeuring* or *goedgekeurd*) into an invalid pattern match (using *laten* or *toelaten*) because the score of synonyms is higher than the score of suggested terms by Word2Vec.

Terms used for matching with *afkeuren* are shown in the table below. Only three words are actually used, where no matches use a suggestion from Word2Vec. This should be no surprise since we set π to 0.57 we filtered out many Word2Vec suggestions for *afkeuren*.

Matching Word	Translation	Type of Match	Score	#Matches
afkeuren	to refuse	Same word	1	194
afwijzen	to decline	Synonym	1	20
afbreken	to interrupt	Synonym	1	6

Still, these terms are in general only a part of an event description. Recall that we match events based on their description, which mainly consists of multiple words. The similarity Function (Function 4) matches these descriptions by matching loose words with each other, as discussed in Chapter 7.3.2, but does not consider groups of words as a unit. For instance, a match uses event "bevestigen eerste gesprek door sollicitant niet akkoord" (confirm first interview by applicant do not approve) as a match for *goedkeuren*. This match is based on the fact that *bevestigen* is a synonym of *goedkeuren*. The actual meaning of this event is that some conversation which needs to be approved does not get approved. With this meaning, we may argue that this event has a different purpose than the event *goedkeuren* in the approval pattern and would be more suitable as a match for *afkeuren*.

In the next sections, we study some extended variants of the approval pattern.

8.2 Extending the Approval Pattern (*afkeuren*)

One way to extend the approval pattern defined above is given in Figure 8.5. We can add an event after *afkeuren*. The event *opnieuw insturen* (which means *to resubmit*) is a logical step which should occur after someone has refused something. We also chose the event *opnieuw insturen* because now we can study the behavior of the proposed method while using multiple words for one event.

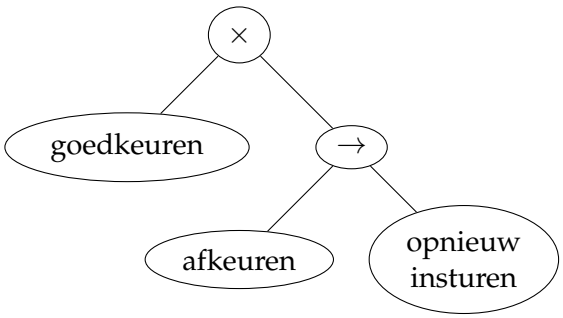

FIGURE 8.5: An extended variant of the basic approval-pattern, shown in Figure 8.1. This variant forces that the event *opnieuw insturen* occurs after event *afkeuren*.

Note that the pattern shown in Figure 8.5 does not include a possible loop. If we would want the approval process to be give the opportunity to repeat events, then we want to define the approval pattern as shown in Figure 8.6. This enables the process to be repeated, meaning that its underlying events can be executed multiple times.

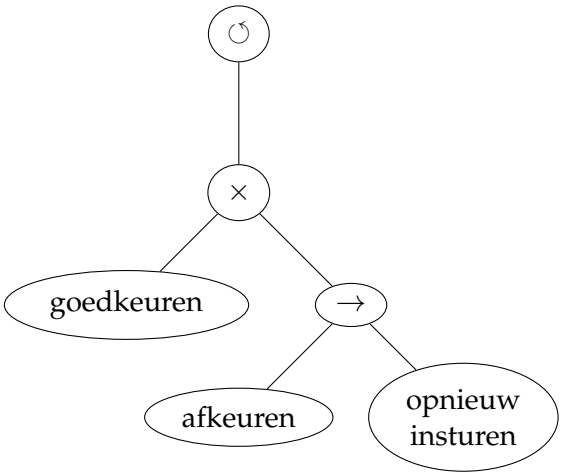

FIGURE 8.6: Loop variant of the extended approval pattern shown in Figure 8.7.

Again, let's find out which terms are suggested by Word2Vec for the words *opnieuw* and *insturen*. Recall that the Similarity Function (Function 4) tries to match single words with each other, so we need to check suggestions by Word2Vec for the two words separately. Table 8.5 shows the highest scoring Word2Vec suggestions for *opnieuw* and Table 8.6 shows the suggestions for *insturen*. As you can see, using Word2Vec for *opnieuw* does not result in many valid suggestions. For *insturen* however, the resulting set of suggestions seems to be more of use. Even some suggestions with a score < 0.57 are valid. However, to avoid that we find pattern matches with many invalid matches for the other terms, we stick to $\pi = 0.57$. This also means that we lose some options when trying to find a match for *insturen*.

Table 8.6 also shows that there is a possibility of obtaining incorrect Dutch words. The word *eenbe* is not a valid Dutch word. The same goes for *merstudie* in Table 8.1.

Dutch Term	**English Translation**	**Similarity Score**
weer	again	0.7342
meteen	*right away*	*0.6780*
eerst	*first*	*0.6300*
daarna	*afterwards*	*0.5726*
snel	*quick*	*0.5669*
eindelijk	*finally*	*0.5659*
terug	*return*	*0.5631*
...	...	...

TABLE 8.5: List of similar terms to *opnieuw*, obtained through Word2Vec.

Dutch Term	English Translation	Similarity Score
binnensturen	to submit	0.6242
ingestuurd	submitted	0.6145
opsturen	to send	0.5815
instuurt	submits	0.5293
eenbe	-	*0.5091*
doorsturen	to forward	0.5079
instuurde	submitted	0.5055
binnengestuurd	submitted	0.5053
uitkiezen	*to select*	*0.5041*
...	...	...

TABLE 8.6: List of similar terms to *insturen*, obtained through Word2Vec.

Keeping the value of π at 0.57, we get the following results when searching for matches of the extended variants of the approval pattern. For the variant without a loop (Figure 8.5), see Table 8.7 and the matches of the variant with a loop (FIgure 8.6) are summarized in Table 8.8. Clearly, the variant without a loop gets recognized more often than the variant which includes repeats. Note that Figure 8.6 is an extension of Figure 8.5.

Type Of Matching	Models	Matches	Distinct Matches	Avg Score
Induced (1)	4	4	2	0.981
Induced (+)		4	2	0.981
Embedded (1)	46	46	32	0.992
Embedded (+)		**50**	**34**	0.992

TABLE 8.7: Using Figure 8.5 as pattern, $\pi = 0.57$ and $\epsilon = 0$.

Type Of Matching	Models	Matches	Distinct Matches	Avg Score
Induced (1)	0	0	0	-
Induced (+)		0	0	-
Embedded (1)	9	9	5	0.983
Embedded (+)		9	5	0.983

TABLE 8.8: Using Figure 8.6 as pattern, $\pi = 0.57$ and $\epsilon = 0$.

From these results, we can make up that the approval pattern is mostly applied once in a workflow instance. In other words, whenever there is an approval step in a workflow, this step is mostly done only once (even though we have added an event used for resubmission).

8.3 Extending the Approval Pattern (*goedkeuren*)

We can also extend the approval pattern where something has been approved. We can add an extra event, *afhandelen* (to finish) after *goedkeuren*. This extension captures the process of ending the (sub)workflow whenever some process is approved.

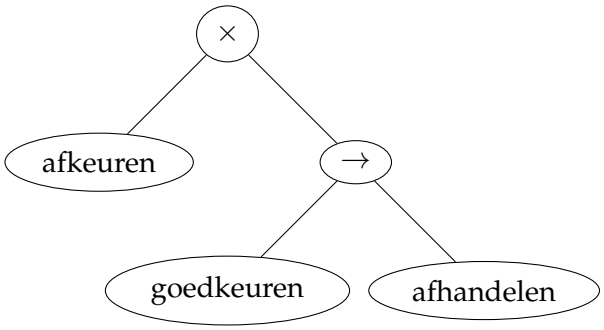

FIGURE 8.7: An extended variant of the basic approval-pattern, shown in Figure 8.1. This variant forces that the event *afhandelen* occurs after event *goedkeuren*.

Considering Word2Vec suggestions of afhandelen, we obtain Table 8.9. The retrieved suggestions seem to be quite good. They also mostly fall in the range of (0.57-1), which we have been using so far. We can still use $\pi = 0.57$ when searching for any matches.

Dutch Term	English Translation	Similarity Score
afgehandeld	concluded	0.6569
afhandeling	settlement	0.6162
afhandelt	concludes	0.5993
afwikkelen	to wind off	0.5960
regelen	*to arrange*	*0.5790*
afronden	to round off	0.5763
oplossen	*to solve*	*0.5731*
uitklaren	*to clear up*	*0.5702*
nasturen	*to forward*	*0.5539*
doornemen	*to go through*	*0.5483*
...	...	...

TABLE 8.9: List of similar terms to *afhandelen*, obtained through Word2Vec.

The matching results of this extended variant are shown in Table 8.10, which shows that the *afhandelen* variant occurs very rarely in induced form. However, the number of matches is quite high when we apply embedded matching.

Type Of Matching	Models	Matches	Distinct Matches	Avg Score
Induced (1)	8	8	3	1
Induced (+)		8	3	1
Embedded (1)	75	75	45	0.994
Embedded (+)		**112**	**61**	0.994

TABLE 8.10: Using Figure 8.7 as pattern, $\pi = 0.57$ and $\epsilon = 0$.

The terms used in the distinct embedded matches for *afhandelen* after given in the Table below. Note that, besides the *afhandelen* itself, the other two matches are suggestions from Word2Vec.

Matching Word	Translation	Type of Match	Score	#Matches
afhandelen	to finish	Same word	1	52
afgehandeld	finished	Word2Vec	0.6569	6
afhandeling	finish	Word2Vec	6162	3

8.4 Evaluation

Ultimately, we conclude that the approval pattern occurs in different multiple variations in our set of workflow models. By extending the pattern definition, the number of matches found decreases quickly. However, this does not mean that a pattern should not be extended. Studying extended variants of the pattern shows us that repeats rarely occur when it comes down to approving some process.

Although we make use of synonyms and Word2Vec to find matches which use words other than those which are included in the pattern itself, the majority of the matches found making use of the exact term. This shows the importance of choosing which terms you want to include in your pattern definition. Still, we have shown that the use of synonyms and Word2Vec suggestions is definitely useful during the matching process.

As we would expect, the number of embedded matches found for each pattern is much higher than the number of induced matches. From this, we can state that most workflow models do contain the pattern, but the nodes used for the pattern match are often not directly linked. This makes us believe that the workflow logs in the dataset used mostly contain events similar to those that are part of the approval pattern. However, the corresponding workflows show behavior which makes it harder for IM to recognize a clear split between *goedkeuren* and *afkeuren*. This can also be caused by noise in our workflow logs.

It may also help to increase the noise threshold ϵ of the Inductive Miner infrequent. This results in smaller and simpler models since it removes infrequent behavior from a workflow model, but also increases the possibility of missing some matches because they are (partly) removed by IMi. Still, you should only consider increasing ϵ if you are mainly interested in frequent events instead of all events.

Chapter 9

Discussion

This chapter concludes this thesis, discusses observations of the accomplished work and proposed directions for future work.

9.1 Conclusion

This thesis has shown that we can define a semantical workflow pattern in the form of a compact abstract representation called *process trees* (Buijs et al., 2012b). This way, we only include the control-flow of events, but also give semantical meaning to each event and the pattern itself. In our case, we added semantical meaning to an event by describing which action this event represents. We also managed to convert our set of workflow logs into process trees automatically through the use of the Inductive Miner (infrequent) (Leemans et al., 2013), (Leemans et al., 2014b).

Furthermore, we have defined a pattern matching rule (Definition 16), which states when a workflow model contains a workflow pattern. We proposed an algorithm (Chapter 7.3.4) that is able to discover occurrences of a given workflow pattern in a set of workflow models. During this discovery process, we also make use of lists of synonyms, antonyms and Word2Vec to extend the number of matches found and lower the possibility of obtaining an invalid match (Chapter 7.3.2).

Ultimately, we studied different variations of the approval pattern in Chapter 8. From these results, we observe that this pattern is actively used in the data of AFAS and shows which variations of the pattern and its events are mostly used. According to these results, we can conclude that the proposed method shows potential and offers insights into the usage of business processes. Nevertheless, we believe that there are many ways to improve the effectiveness of this method.

9.2 Future Works

Concluding, we have come up with some suggestions that can help to improve this research.

- Reduce the amount of noise in your dataset as much as possible. Although we have applied some methods to reduce noise in our dataset, it is highly likely that it still contains some unwanted traces or invalid fields, as we explained in Chapter 2. Noise can have a large effect on the process trees after applying the Inductive Miner. Others (Tax et al., 2017) propose techniques to filter random activities which should be removed to obtain a cleaner set of traces.

- Use the Evolutionary Tree Miner instead of the Inductive Miner. In Chapter 6.3 we mentioned that the user can steer different qualitative properties of process

trees with the Evolutionary Tree Miner, which is not possible with the Inductive Miner. As (Buijs et al., 2012a) states, we are confident that the application of ETM can result in a big improvement of the quality of mined process trees.

- Training a Word2Vec model on your own data might help, as long as you have enough data. For instance, the *SoNaR* corpus (Oostdijk et al., 2013) is trained on over 28 million sentences. Important to note is that the choice of which word embeddings dataset you use, depends on your type of application. If you want to use Word2Vec like a common dictionary of Word2Vec, than I advice to use a very general corpus like the *sonar* corpus. When applying the matching algorithm on a very specific dataset which exists barely of regular Dutch sentences, then applying Word2Vec on your own corpus should be preferred over a generic corpus.

- Stemming/lemmatization of words could be very useful when computing which words are similar. These techniques would make it possible to recognize abbreviations of a term and normalize them to their base form. Where stemming does not always lead to significant improvements (Hollink et al., 2004), we are confident that a dutch lemmatizer would be very helpful in our situation. The only Dutch lemmatizer we could find is called Frog[1], which consists of many NLP modules for Dutch.

- Improve the similarity formula between two sentences. Recall that our similarity function, as described in Chapter 7.3.2, only compares loose words. We expect that computing sentence similarity based on sets of words would improve pattern matching. According to Kenter and de Rijke (2015), using an aggregate function is a rather poor way to capture the distribution of word embeddings. They propose a similarity function which also considers the weight of a word in a given dataset. This way, words which are commonly used have less influence on the similarity measure than words which are rarely used.

- The pattern matching algorithm as we proposed can potentially be improved. Although the algorithm proposed in Chapter 7.3 is able to give valid pattern matches, it does not guarantee to return an optimal match. Besides, in Chapter 8 we have seen that setting the similarity threshold π can be quite difficult. Especially since each word used in a workflow pattern often has a different similarity range when it comes down to Word2Vec suggestions. This often forces the user to generalize over all terms used in the pattern, which can result in missing Word2Vec suggestions or none at all. We believe that a noise threshold per event would be a good alternative.

- Instead of searching for occurrences of a given workflow pattern, it would be also interesting to search for interesting workflow patterns. This could be done through frequent pattern mining (Chapter 5.4).

[1] http://languagemachines.github.io/frog/

Appendix A

The Extended Pattern Matching Algorithm

As explained in Chapter 7.4, we have added some extensions to the algorithm which are described in Chapter 7.3.4. These extensions lead to a change in Functions 5, 7, 8 and 9. The updated versions of these Functions are described given below. In Function 10 we now keep track of the overall score of matches found. Ultimately, the highest scoring match gets returned. A similar extension is added in Functions 11, 12 and 13. Note that these Functions keep track of the best match of a given node.

Function 10: Main function that returns an induced or embedded match of a given pattern P in a process tree T. This is an extended version of Function 5.

```
1  function SearchMatch (T, P, isInduced)
     Input  : A process tree T, a process tree P, boolean value induced stating
              whether we must only consider induced subtrees as matches or
              consider embedded subtrees.
     Output: A pattern match from P in T if possible. This match is an induced or
              embedded subtree, depending on the value of isInduced.
2  p = P.root;
3  nodeList = T.GetNodesInPostOrder();
4  bestMatch = (∅, 0);
5  while |nodelist| > 0 do
6      t = nodelist.pop;
7      result = null;
8      if AreSimilar(t, p) then
9          if isInduced then
10             result = SearchInducedMatch(t, p);
11         end
12         else
13             result = SearchEmbeddedMatch(t, p, []);
14         end
15         score = result.score;
16         if result ≠ null ∧ score > bestMatch.score then
17             bestMatch = (result, score);
18             if score == 1 then
19                 break;
20             end
21         end
22     end
23 end
24 return bestMatch;
```

Function 11: Function that tries to find an induced match for p and its descendants. This is an extended version of Function 7.

```
1  function SearchInducedMatchOrdered (t, p)
   Input  : A process tree node t, a pattern node p.
   Output: An induced match of p in t if there exists any.
2  if |p.children|== 0 then
3      return [⟨t, p⟩];
4  end
5  matches[];
6  startIndex = 0;
7  for p_c ∈ p.children do
8      bestMatch = (∅, 0);
9      for index ∈ range(startingIndex, |t.children|) do
10         t_c = t.children[index];
11         if AreSimilar(t_c, p_c) then
12             subtreeMatch = SearchInducedMatch(t_c, p_c);
13             score = subtreeMatch.score;
14             if subtreeMatch ≠ null ∧ score > bestMatch.score then
15                 bestMatch = (subtreeMatch, score);
16                 if score == 1 then
17                     break;
18                 end
19             end
20         end
21     end
22     if bestMatch ≠ (∅, 0) then
23         matches.add(subtreeMatch);
24         startIndex = t.children.index(subtreeMatch.t) + 1;
25     end
26 end
27 if p.descendants ∈ matches then
28     return [⟨t, p⟩, matches];
29 end
30 return null;
```

Function 12: Function that tries to find an induced match for p and its descendants. This is an extended version of Function 8.

```
1  function SearchInducedMatchUnordered (t, p)
   Input  : A process tree node t, a pattern node p.
   Output: An induced match of p in t if there exists any.
2  if |pChildren|== 0 then
3  |  return [⟨t, p⟩];
4  end
5  matches = [];
6  for p_c ∈ p.children do
7  |  bestMatch = (∅, 0);
8  |  for t_c ∈ t.children do
9  |  |  if t_c ∉ matches ∧ AreSimilar(t_c, p_c) then
10 |  |  |  subtreeMatch = SearchInducedMatch(t_c, p_c);
11 |  |  |  score = subtreeMatch.score;
12 |  |  |  if subtreeMatch ≠ null ∧ score > bestMatch.score then
13 |  |  |  |  bestMatch = (subtreeMatch, score);
14 |  |  |  |  if score == 1 then
15 |  |  |  |  |  break;
16 |  |  |  |  end
17 |  |  |  end
18 |  |  end
19 |  end
20 |  if bestMatch ≠ (∅, 0) then
21 |  |  matches.add(subtreeMatch);
22 |  end
23 end
24 if p.descendants ∈ matches then
25 |  return [⟨t, p⟩, matches];
26 end
27 return null;
```

Function 13: Function that tries to find a embedded match for p and its descendants. This is an extended version of Function 9.

```
1  function SearchEmbeddedMatch (t, p, previousMatches)
   Input  : A process tree node t, a pattern node p, a list of other matching nodes
            of P and T.
   Output: A embedded pattern match of p in t if there exists any.
2  if AreSimilar(t, p) ∧ ¬IsAncestor(t, previousMatches) then
3      if |p.children|== 0 then
4          return [⟨t, p⟩];
5      end
6      newMatches = [];
7      for p_c ∈ p.children do
8          bestMatch = (∅, 0);
9          for t_c ∈ t.children do
10             if t_c ∉ (newMatches ∪ previousMatches) ∧ AreSimilar(t_c, p_c) then
11                 subtreeMatch =
                     SearchEmbeddedMatch(t_c, p_c, newMatches ∪ previousMatches);
12                 score = subtreeMatch.score;
13                 if subtreeMatch ≠ null ∧ score > bestMatch.score then
14                     bestMatch = (subtreeMatch, score);
15                     if score == 1 then
16                         break;
17                     end
18                 end
19             end
20         end
21         if bestMatch ≠ (∅, 0) then
22             newMatches.add(bestMatch);
23         end
24     end
25     if p.descendants ∈ newMatches then
26         return [⟨t, p⟩, newMatches];
27     end
28 end
29 for t_c ∈ t.children do
30     if t_c ∉ previousMatches then
31         match = SearchEmbeddedMatch(t_c, p, previousMatches);
32         if match ≠ null then
33             return match;
34         end
35     end
36 end
37 return null;
```

Bibliography

Agrawal, R., Gunopulos, D., and Leymann, F. (1998). Mining Process Models from Workflow Logs. *6th International Conference on Extending Database Technology*, 1377 LNCS:467–483.

Buijs, J. C., Van Dongen, B. F., and Van Der Aalst, W. M. (2012a). On the role of fitness, precision, generalization and simplicity in process discovery. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7565 LNCS, pages 305–322.

Buijs, J. C. A. M. (2010). Mapping Data Sources to XES in a Generic Way. *Chelsea*, (March):123.

Buijs, J. C. A. M., van Dongen, B. F., and Van der Aalst, W. M. P. (2012b). A Genetic Algorithm for Discovering Process Trees. *WCCI 2012 IEEE World Congress on Computational Intelligence*, pages 925–932.

Chapela-Campa, D., Mucientes, M., and Lama, M. (2017). Mining Frequent Patterns in Process Models.

David, S., Carmona, J., and Munt, V. (2017). Reducing Event Variability in Logs by Clustering of Word Embeddings. (June).

De Medeiros, A. K. A., Van Dongen, B. F., Van Der Aalst, W. M. P., and Weijters, A. J. M. M. (2004). Process mining: Extending the α-algorithm to mine short loops. *Eindhoven University of Technology Eindhoven*, pages 1–25.

Dijkstra, K. D. B., Kipping, J., and Mézière, N. (2003). Workflow (Control-flow) Patterns. *Odonatologica*, 44(4):447–678.

Ellis, C. A. and Bernal, M. (1982). Officetalk-D: An Experimental Office Information System.

Engel, G. H., Groppuso, J., Lowenstein, R. A., and Traub, W. G. (1979). An office communications system. *IBM Systems Journal*, 18(3):402–431.

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.

Hammer, M. (1990). Rengineering Work: Don't Automate, Obliterate. *Harvard Business Review*.

Hollink, V., Kamps, J., Monz, C., and de Rijke, M. (2004). Monolingual Document Retrieval for European Languages. *Information Retrieval*, 7(1):33–52.

Kenter, T. and de Rijke, M. (2015). Short Text Similarity with Word Embeddings. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management - CIKM '15*, pages 1411–1420.

Klinkmüller, C., Weber, I., Mendling, J., Leopold, H., and Ludwig, A. (2013). Increasing Recall of Process Model Matching by Improved Activity Label Matching.

Kopp, O., Martin, D., Wutke, D., and Leymann, F. (2009). The difference between graph-based and block-structured business process modelling languages. *Enterprise Modelling and Information Systems*, 4(1):3–13.

Leemans, S. J., Fahland, D., and Van Der Aalst, W. M. (2014a). Process and deviation exploration with inductive visual miner. In *CEUR Workshop Proceedings*, volume 1295, pages 46–50.

Leemans, S. J. J., Fahland, D., and van der Aalst, W. (2014b). Discovering Block-Structured Process Models from Incomplete Event Logs. In *Application and Theory of Petri Nets and Concurrency, LNCS 8489*, pages 91–110.

Leemans, S. J. J., Fahland, D., and Van Der Aalst, W. M. P. (2013). Discovering block-structured process models from event logs - A constructive approach. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7927 LNCS, pages 311–329. Springer, Berlin, Heidelberg.

Loper, E. and Bird, S. (2002). Nltk: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*, ETMTNLP '02, pages 63–70, Stroudsburg, PA, USA. Association for Computational Linguistics.

McCormick, C. (2016). "word2vec tutorial-the skip-gram model.

Mikolov, T., Corrado, G., Chen, K., and Dean, J. (2013a). Efficient Estimation of Word Representations in Vector Space. *Proceedings of the International Conference on Learning Representations (ICLR 2013)*, pages 1–12.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013b). Distributed Representations of Words and Phrases and their Compositionality. pages 3111–3119.

Măruşter, L., Weijters, a. J. M. M., van der Aalst, W., and van den Bosch, A. (2002). Process mining: Discovering direct successors in process logs. *Discovery Science, Proceedings*, 2534:364–373.

Oostdijk, N., Reynaert, M., Hoste, V., and Schuurman, I. (2013). *The Construction of a 500-Million-Word Reference Corpus of Contemporary Written Dutch*, pages 219–247. Springer Berlin Heidelberg, Berlin, Heidelberg.

Pagliardini, M., Gupta, P., and Jaggi, M. (2017). Unsupervised Learning of Sentence Embeddings using Compositional n-Gram Features.

Riehle, D. and Züllighoven, H. (1996). Understanding and using patterns in software development. *Theor. Pract. Object Syst.*, 2(1):3–13.

Russell, N., Hofstede, A. H. M., Edmond, D., and Aalst, W. M. P. V. D. (2004). Workflow (Data) Patterns. *Business*, 66(FIT–TR–2004–01):–2004–01.

Russell, N., Van Der Aalst, W., and Ter Hofstede, A. (2006). Workflow exception patterns. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4001 LNCS, pages 288–302. Springer, Berlin, Heidelberg.

Russell, N., van der Aalst, W. M. P., ter Hofstede, A. H. M., and Edmond, D. (2005). Work ow Resource Patterns: Identi cation, Representation and Tool Support. *Advanced Information Systems Engineering*, pages 216–232.

Tax, N., Sidorova, N., Haakma, R., and van der Aalst, W. M. (2016). Mining local process models. *Journal of Innovation in Digital Ecosystems*, 3(2):183–196.

Tax, N., Sidorova, N., and van der Aalst, W. M. P. (2017). Discovering More Precise Process Models from Event Logs by Filtering Out Chaotic Activities.

Tulkens, S., Emmery, C., and Daelemans, W. (2016). Evaluating unsupervised dutch word embeddings as a linguistic resource. In Chair), N. C. C., Choukri, K., Declerck, T., Grobelnik, M., Maegaard, B., Mariani, J., Moreno, A., Odijk, J., and Piperidis, S., editors, *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, Paris, France. European Language Resources Association (ELRA).

Umble, E. J., Haft, R. R., and Umble, M. M. (2013). Enterprise resource planning: Implementation procedures and critical success factors.

van der Aalst, W. M. (1997). Verification of workflow nets. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1248, pages 407–426. Springer, Berlin, Heidelberg.

Van der Aalst, W. M., Ter Hofstede, A. H., Kiepuszewski, B., and Barros, A. P. (2003a). Workflow patterns.

Van der Aalst, W. M., Van Dongen, B. F., Herbst, J., Maruster, L., Schimm, G., and Weijters, A. J. (2003b). Workflow mining: A survey of issues and approaches. *Data and Knowledge Engineering*, 47(2):237–267.

Van der Aalst, W. M. P. (1998). the Application of Petri Nets To Workflow Management. *Journal of Circuits, Systems and Computers*, 08(01):21–66.

van der Aalst, W. M. P. and van Dongen, B. F. (2002). Discovering Workflow Performance Models from Timed Logs. pages 45–63. Springer, Berlin, Heidelberg.

Van Der Aalst, W. M. P., Weijters, A. J. M. M., and Maruster, L. (2002). Workflow mining: which processes can be rediscovered?

Verbeek, H. M., Buijs, J. C., Van Dongen, B. F., and Van Der Aalst, W. M. (2010). ProM 6: The process mining toolkit. *CEUR Workshop Proceedings*, 615:34–39.

Verbeek, H. M., Buijs, J. C., Van Dongen, B. F., and Van Der Aalst, W. M. (2011). XES, XESame, and ProM 6. In *Lecture Notes in Business Information Processing*, volume 72 LNBIP, pages 60–75. Springer, Berlin, Heidelberg.

Weijters, A. J. M. M. (2001). Rediscovering Workflow Models from Event-Based Data. (i).

Weijters, A. J. M. M. and van der Aalst, W. (2001). Process mining: Discovering workflow models from event-based data. *Proceedings of the 13th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC'01)*, (i):283–290.

Weijters, A. J. M. M., Van Der Aalst, W. M. P., and Medeiros, A. K. A. D. (2006). Process Mining with the Heuristics Miner Algorithm. *Technische Universiteit Eindhoven, Tech. Rep. WP*, 166:1–34.