Applying process mining to multi-instance processes

Teeuwen, Rob M.A.

Award date:
2020

Department of Mathematics and Computer Science
Process analytics research group

Applying process mining to multi-instance processes

Graduation thesis Data Science in Engineering

Rob Teeuwen

Supervisors:
Prof.dr.ir. Boudewijn van Dongen
Roeland Scheepens PhD

Eindhoven, Friday 31st July, 2020

Abstract

The intended or expected behaviour of business processes can be captured in business process models. These business processes can have a hierarchical structure where a process can contain one or more (nested) subprocesses. Multi-instance processes are hierarchical processes where for each process instance there are multiple subprocess instances.

The enactment of business processes is often supported by information systems that store information about the process execution in an event log. The field concerned with extracting information from event logs is called process mining. Process mining can be divided into process discovery, conformance checking, and process enhancement, the former two being relevant for this research. Traditional process mining methods can not be applied to multi-instance processes as they are not equipped to differentiate the different instances of multi-instance subprocesses. Furthermore, traditional methods cannot express multi-instance subprocesses in a process model.

In this thesis, we extend traditional process discovery and conformance checking methods to become applicable to multi-instance processes. Furthermore, the visualization of process models is extended for multi-instance processes. Our approach is to separate the process for each level in the process hierarchy. This allows us to use traditional methods for each of the separated processes. To get the result for the whole process, we combine the results for the separated processes afterwards. We evaluate our methods using both artificial and real-life datasets.

Contents

Chapter 1

Introduction

A business process is a collection of related, structured tasks, that together produce a service or a product. The flow of these tasks can be structured using business process models. These models indicate what tasks have to be executed, by whom, and in what order. The Business Process Model and Notation(BPMN) standard for process modeling is a widely used method for modeling a business process. [9]

Business processes can have a hierarchical structure, for example a process can have one or multiple subprocesses, where a subprocess is a compound activity that represents a collection of other tasks and subprocesses. As an example, lets look at the process model in Figure 1.1, which represents a model of a loan application. Once the loan application has been submitted and checked, the bank can choose to either refuse the application or to do one, or multiple offers. Here, each offer follows the same subprocess, where each offer can either be accepted or refused by the customer. Note that for each instance of this process, i.e. for each application, there can be multiple instances of its subprocess, i.e., there can be multiple offers, that can be executed in parallel. We call such a subprocess a multi-instance subprocess. We call a process that contains one or more multi-instance subprocesses a multi-instance process. Multi-instance processes are the main topic of this thesis.

The enactment of business processes is often supported by information systems, such as workflow management systems, or enterprise resource planning systems.[10] These systems store information about the execution of the business process in event logs. Each record in an event log corresponds to an event. This event includes information about the task that was executed as well as a timestamp. Additionally, other attributes may be included such as the resource performing the task. Event logs can be analyzed to provide information

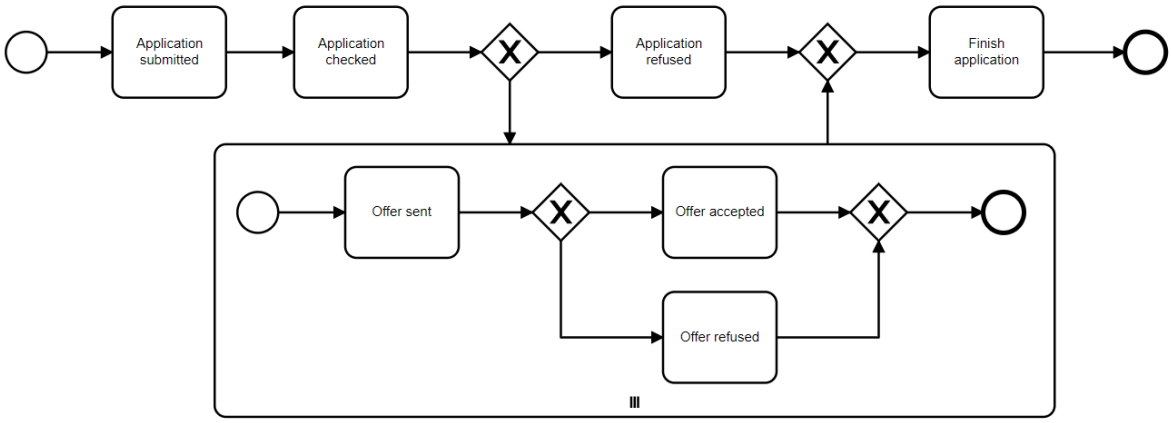


Figure 1.1: An example of a BPMN model of a loan application.

about the execution of business processes. The field concerned with extracting information from event logs is called process mining.

Process mining can be divided into process discovery, conformance checking and process enhancement, the former two being relevant for this research. Process discovery aims to discover a process model based on an event log, thereby, giving a data-driven interpretation of a process. Conformance checking aims at finding differences between a given process model and event log. It can be used to quantify compliance and analyze discrepancies.

Traditional process mining methods are not equipped to handle processes with multi-instance behaviour. The reason for this is made clear in Chapter 3. In this work, we aim to solve this problem. Therefore we define the main research question of this work as:

- **How can process mining methods be applied to multi-instance processes?**

This research is conducted at UiPath, a software company that develops a platform for robotic process automation. Their core product of the Eindhoven office is UiPath Process Mining, a platform which can be used to provide users insights into the execution of their business processes. In this research, we have developed a proof-of-concept prototype for applying process mining methods to multi-instance processes in UiPath process mining.

In this thesis, we first explain relevant background knowledge needed to understand this thesis in chapter 2. Then, in Chapter 3, we explore the limitations of current process mining methods when applied to multi-instance processes. Furthermore, we subdivide the

main research question to further specify our goals and discuss the general strategy that is used to solve these questions. In chapters 4 to 6 we discuss the approach that is used to solve each of the research questions. We evaluate this approach in Chapter 7. Finally, in Chapter 8, we discuss the conclusions, limitations, and future work of this research.

Chapter 2

Preliminaries

2.1 BPMN

A business process consists of tasks or activities that are required to be executed in a specific order. A business process model is used to show the order the activities are intended to be executed in. There exist several different notations and formalisms to express such a business process (e.g. BPMN[9], YAWL[2], Petri nets[12], Process trees[4] and more).

The available notation in the UiPath Process Mining platform is based on the Business Process Model Notation (BPMN).[9] BPMN represents process models using a flowchart notation. The BPMN standard contains over 50 graphical elements that can be used to specify how a business process should be executed. These are divided into two groups: the core set and the extended set. The core set represents only the basic flowchart elements such as tasks, edges, exclusive and parallel gateways, messages, and resource assignment constructs. More advanced constructs such as different event types, loops, different kinds of subprocesses and more, are represented in the extended set. The elements that are relevant for this work are tasks, edges, exclusive and parallel gateways, and multi-instance subprocesses. These elements are shown and explained in Figure 2.1.

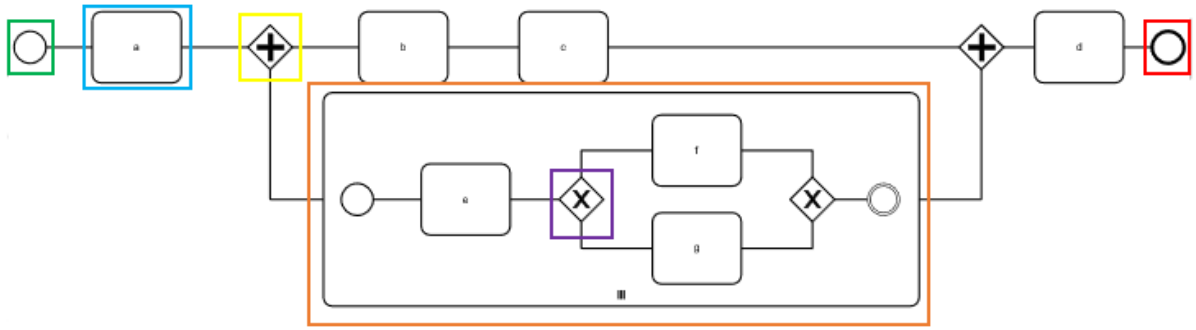


Figure 2.1: En example of a process model represented in BPMN consisting of all relevant elements in this research. The elements are, tasks (blue), start event (green), end event (red), exclusive gateways (purple), parallel gateways (yellow) and multi-instance subprocesses (orange). Tasks correspond to business process activities, and start and end events correspond to the start and end of a process instance. For exclusive gateways, for each instance of the process, only one of the paths can be taken. For parallel gateways, for each process instance, each of the paths need to be taken. Multi-instance subprocesses can be recognized by the frame around the subprocess, the start and end events inside and by the icon in the bottom of the frame.

2.2 Process mining

As discussed before, the enactment of business processes is often supported by information systems that store information about executed tasks in log files, called event logs. Process mining is the discipline of extracting information from such event logs.[10]

2.2.1 Event logs

An **event log** consists of a list of events containing the activities that were executed in the process. Each **event** corresponds to an execution of a business process activity. Each event has at least the following attributes:

- **Case ID**, the identifier corresponding to the specific instance of the process.
- **Activity**, the specific step that was performed in the process.
- **Timestamp**, the date and time the event occurred. The timestamp is used to determine the order of events.

Case ID	activity	timestamp
0	a	07-01-00 18:24
0	e	11-01-00 18:09
0	e	12-01-00 9:49
0	e	13-01-00 21:04
0	b	18-01-00 19:13
0	f	20-01-00 6:04
0	f	20-01-00 16:00
0	g	30-01-00 5:39
0	c	30-01-00 9:39
0	d	25-03-00 8:21
1	a	02-01-00 15:28
1	e	02-01-00 21:00
1	e	03-01-00 12:46
1	g	04-01-00 15:19
1	e	07-01-00 0:38
1	f	08-01-00 2:48
1	b	09-01-00 18:50
1	c	20-01-00 21:57
1	g	23-01-00 7:47
1	d	09-02-00 0:36
2	a	03-01-00 16:36
2	e	08-01-00 20:01
2	f	14-01-00 11:16
2	b	18-02-00 3:12
2	c	06-03-00 3:38
2	d	11-03-00 18:36

[<a, e, e, e, b, f, f, g, c, d>,
<a, e, e, g, e, f, b, c, g, d>,
<a, e, f, b, c, d>]

Figure 2.2: An example of an event log(left) and its simplified event log(right).

Events may also have other attributes, such as the resource performing the activity or the cost associated with the activity. An example of an event log corresponding to the model in Figure 2.1 is shown in Figure 2.2. A **trace** is a sequence of activities, ordered by timestamp, containing all events corresponding to a specific case. A **simplified event log** is the set of traces of an event log. An example of a simplified event log is shown in Figure 2.2. Event logs are used in process mining to conduct process discovery and conformance checking.

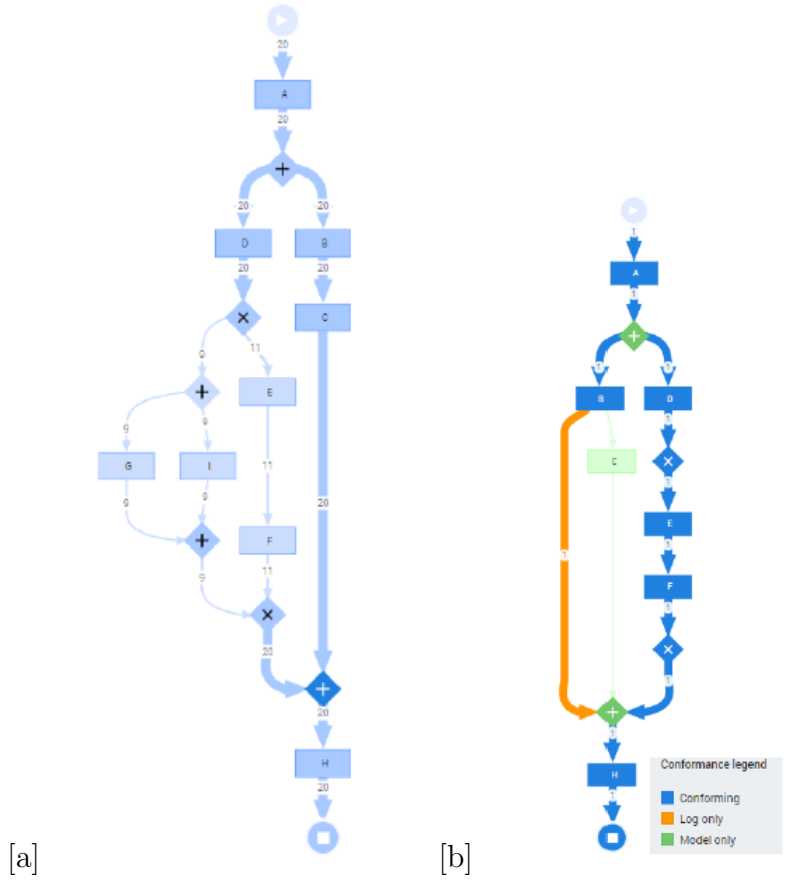
2.2.2 Process discovery

A process discovery technique takes an event log as input, and produces a process model capturing the behaviour seen in the log as output. Workflow management and enterprise resource management systems are often designed with a certain execution of a process in mind. However, in reality, the actual execution often deviates from this intended model. Process discovery can be used to give a data-driven model of the underlying process. There are several types of process discovery algorithms such as the heuristic miner[13], the inductive miner[7], and region based mining[11]. These algorithms aim to produce a model based on an event log while balancing the following criteria:

- **Fitness**, how much of the behaviour in the event log is captured in the model
- **Precision**, how much "extra" behaviour, that is not present in the event log, does the model allow for.
- **Generalization**, how much does the model generalize example behaviour in the log.
- **Simplicity**, how simple is the discovered model.

2.2.3 Conformance checking

In conformance checking, a process model is compared to an event log of the same process with the goal of finding commonalities and discrepancies between modeled and observed behaviour.[10] If a model is used to describe a process, conformance checking can be used to check how well a model (found in process discovery) conforms to the data in the event log. This can give insights in the quality of the model. If the model dictates what should happen in the process, conformance checking can be used to find mistakes in the execution of the process. Different techniques for conformance checking exist. These techniques are often based on some kind of mapping between the model and the log.[10] These mappings can be used to compute a more formally defined metric of the criteria (usually fitness and precision) mentioned in the previous section.



[a] [b]

Figure 2.3: a) A process graph and b) a visual alignment

2.3 The UiPath Process Mining platform

In this research, we developed a proof-of-concept prototype component in the UiPath Process Mining platform.[1] In the following subsections, we discuss the relevant components in the platform used in this research.

2.3.1 Inductive Miner

The process discovery technique in the platform is the Probabilistic Inductive Miner.[6] It is an implementation of the Inductive Miner framework that generates process trees.[7] These process trees are formal models that can be converted into BPMN models. The Probabilistic Inductive Miner discovers structures in the model based on probabilities derived from the frequency information in the data. The input is an event log which first is transformed into a simplified log before it is used by the inductive miner to generate a block-structured process model.

2.3.2 Process graph

The process graph is the visualization of the process model in the UiPath Process Mining platform. The nodes, edges, and gateways use BPMN notation. The graph layout is determined by assigning each node and edge a rank and order using Tracy[8], an algorithm that determines the relative positions between the nodes. Then, this information is used to compute the graph layout. In this step all nodes and paths of edges are assigned a control point, consisting of an x- and y-coordinate, based on their rank and order. Each node has a single control point, edges have a list of control points determining the path the edge follows. An example of the process graph is shown in Figure 2.3a.

2.3.3 Visual alignments

Visual alignments[3] is a method for conformance checking where the event log is aligned over the model in a visual way, as shown in Figure 2.3b. It shows where log and model moves are made by adding log only nodes and edges to the model, and by using color to distinguish log-only, model-only, and conforming behaviour.

Chapter 3

Research motivation & goals

In this chapter, we discuss the research motivation and goals of this work. First, we more clearly define what a multi-instance process is. Second, we explain the limitations of traditional process mining methods when we apply them to these processes. Then, we divide our research question in multiple sub questions in order to set clear goals for our research. Finally, we discuss the general solution strategy that is used to answer these questions.

A **subprocess** is a compound activity that represents a collection of activities and subprocesses. A **multi-instance subprocess** is a type of subprocess where for each instance of the process, multiple instances of the subprocess are executed in parallel. We refer to processes containing one or more multi-instance subprocesses as **multi-instance processes**. In Figure 3.1, an example of a multi-instance process is shown using BPMN notation. The subprocess can both be collapsed into a single compound task if a higher-level overview is desired, or expanded to show the entire subprocess.

Note that for each case, a multi-instance subprocess has multiple subcases. In this research, we assume that the event log of a multi-instance process has a subcase ID attribute for each multi-instance subprocess. Furthermore, we assume that for events that are not part of a subprocess, its subcase ID is null. This subcase ID is used to distinguish different instances of subprocesses.

Multi-instance processes are not limited to a single multi-instance subprocess, i.e., a multi-instance process can contain multiple different multi-instance subprocesses. Furthermore,

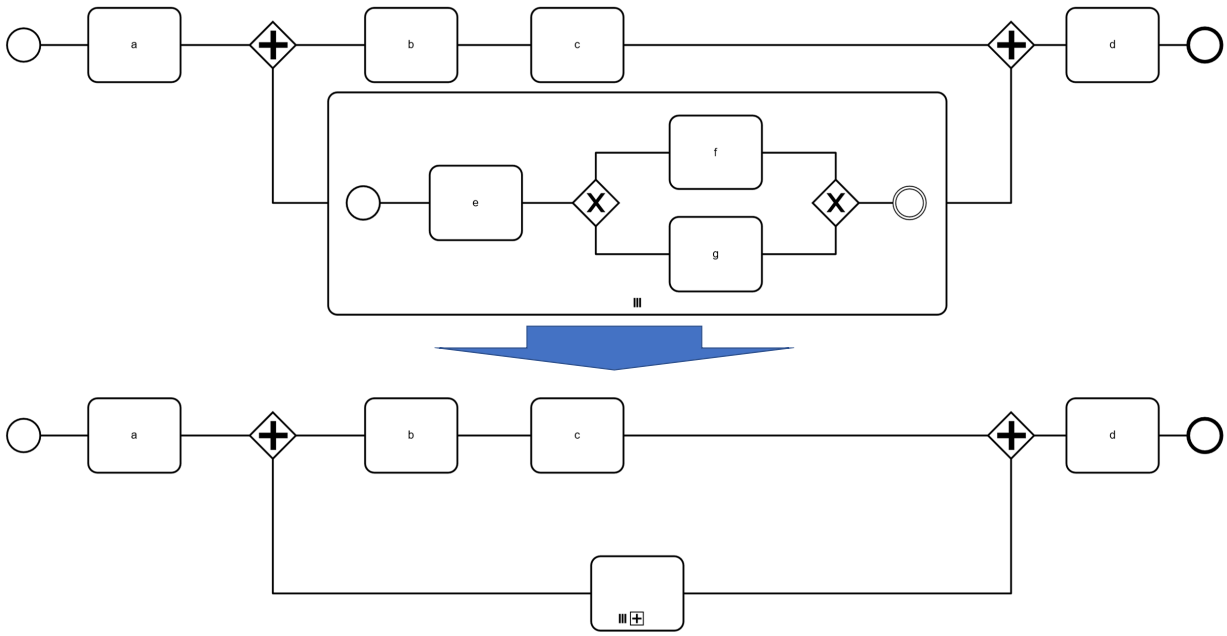


Figure 3.1: An example of a multi-instance process in BPMN notation. The subprocess can both be expanded to show the subprocess(top) or collapsed into a single compound task(bottom).

multi-instance subprocesses can be nested, i.e., a multi-instance subprocess can contain another multi-instance subprocess.

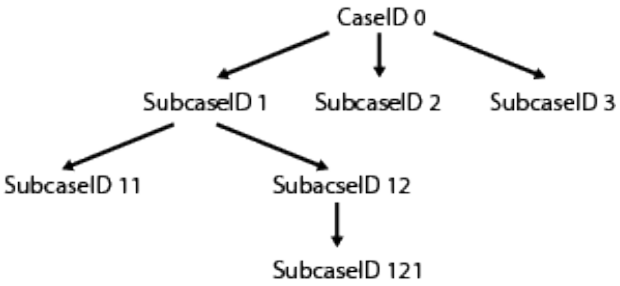


Figure 3.2: An example of a hierarchy of a process.

The relations between the subprocesses can be captured in a process hierarchy. An example of such a hierarchy is shown in Figure 3.2. The hierarchy has the form of a tree where there is a one-to-many relationship between each parent and child. The root of this tree is the main process and all other nodes are (nested) subprocesses. We can differentiate the instances of each of these nodes by their respective subcase ID. When we refer to the relative position of the subprocess in the hierarchy tree we use the term **hierarchy level**. In this project, we assume that the hierarchy of the process is known.

Several issues can occur when using traditional process mining techniques on multi-instance processes. Most of these issues arise from the fact that traditional process mining methods cannot distinguish different instances of a multi-instance subprocess. Furthermore, traditional process mining methods usually use models in the form of a petri net, process tree, or model using the core set of BPMN, all of which lack the capability to express the presence of a multi-instance subprocess. These problems can become even more severe for more complex process hierarchies. It is therefore important to find a way to apply process mining methods to these multi-instance processes so that they can be visualized in an insightful manner. We formulate the following research questions.

1. **How can we discover multi-instance processes?** The inductive miner cannot distinguish different instances of subprocesses and lacks the capability of expressing a multi-instance subprocess in the process model. We must adapt the inductive miner to solve this problem.

2. **How can we derive conformance metrics from multi-instance processes?** Conformance checking using visual alignments needs to be adapted to be able to distinguish different process instances and to express a multi-instance subprocess in

the visual alignments.

3. **How can we visualize multi-instance processes?** This covers both the design of what a multi-instance subprocess looks like in the process graph as well as the actual graph layout. For usability, the design of the multi-instance subprocess in the process graph needs to be intuitive and easy to understand. As hierarchies can become complex, the models will become more complex. It is important that a user can have a clear view of the process. Furthermore we want the process graph to be stable under the collapsing of a subprocess. For example, when a subprocess is collapsed, the rest of the graph should not change too much, as to preserve the mental map of the user.

4. **How can the above be used to obtain new insights (compared to traditional process mining methods)?** The results of the above need to be tested on multiple datasets in order to see if these can be used to obtain new insights.

The expected output of this project is a prototype component apply process discovery, derive conformance metrics, and visualize multi-instance processes integrated in the UiPath Process Mining platform.

This component is evaluated by using it to analyse both artificial and real-life data sets. The real-life dataset is a pathology use case where the cases consist of a hierarchy of subprocesses. This set has explicit multi-instance information in the data. These datasets are used to evaluate if the component gives the desired output. Furthermore, it is investigated if the component can be used to obtain new insights compared to more traditional process mining techniques.

3.1 Solution strategy

The main issue when applying traditional process mining methods to multi-instance processes is that traditional methods cannot distinguish the different instances of subprocesses. Another way to look at this restriction is: traditional process mining methods can only be applied to one hierarchy level at a time. We use this insight to solve research questions 1, 2 and 3. The general solution strategy is to separate a multi-instance process into several processes, one for each hierarchy level. An example of how we could do this is shown in Figure 3.3. Now, we can use traditional methods for each separated process and merge the results. We can isolate each hierarchy level by omitting all higher level processes and by collapsing all the subprocesses of the current hierarchy level. Note that this general solution strategy aims to keep the existing process mining methods as intact as possible. The advantage of this is that when, for example, improvements are made in the induct-

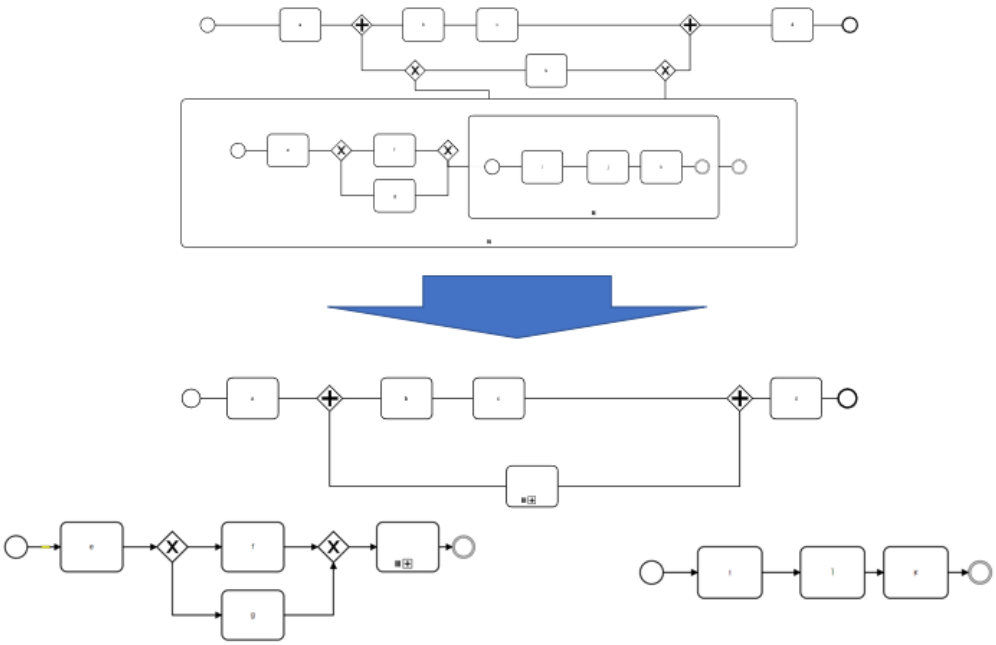


Figure 3.3: An example of how we can separate each hierarchy level of a model.

ive miner, these improvements can instantly be used in the discovery of multi-instance processes.

Chapter 4

Process discovery

In this chapter, we discuss the approach that we take in discovering processes with multi-instance subprocesses. One of the main problems with using traditional mining algorithms on these processes is that they cannot differentiate the different instances of subprocesses. This can lead to models that do not reflect the underlying process accurately. An example of this is shown in Figure 4.1. Here, a comparison is shown between the models discovered by the inductive miner and the multi-instance miner presented in this chapter. The inductive miner produces an incorrect model, e.g. for each instance of the process, according to this model, there is only one instance of activity k. However, since this is the most deeply nested multi-instance subprocess, there are multiple instances of k in the log for each case. Specifically, the 'traditional' method used by the inductive miner results in a model that cannot detect any of the Multi-instance structure while the Multi-instance miner provides this in a structured way.

An overview of the approach that was taken in this research is shown in Figure 4.2. While traditional process mining algorithms cannot distinguish different instances of subprocesses, our approach solves this problem by splitting the log for each level in the process hierarchy. This enables us to use the (sub)case ID to distinguish the different instances of the (sub)process. Now, we can use traditional process discovery methods for each level separately. In this project we used the inductive miner from the UiPath Process Mining platform in the process discovery step. After obtaining the models for each hierarchy level we can merge them together to obtain the model of the whole process.

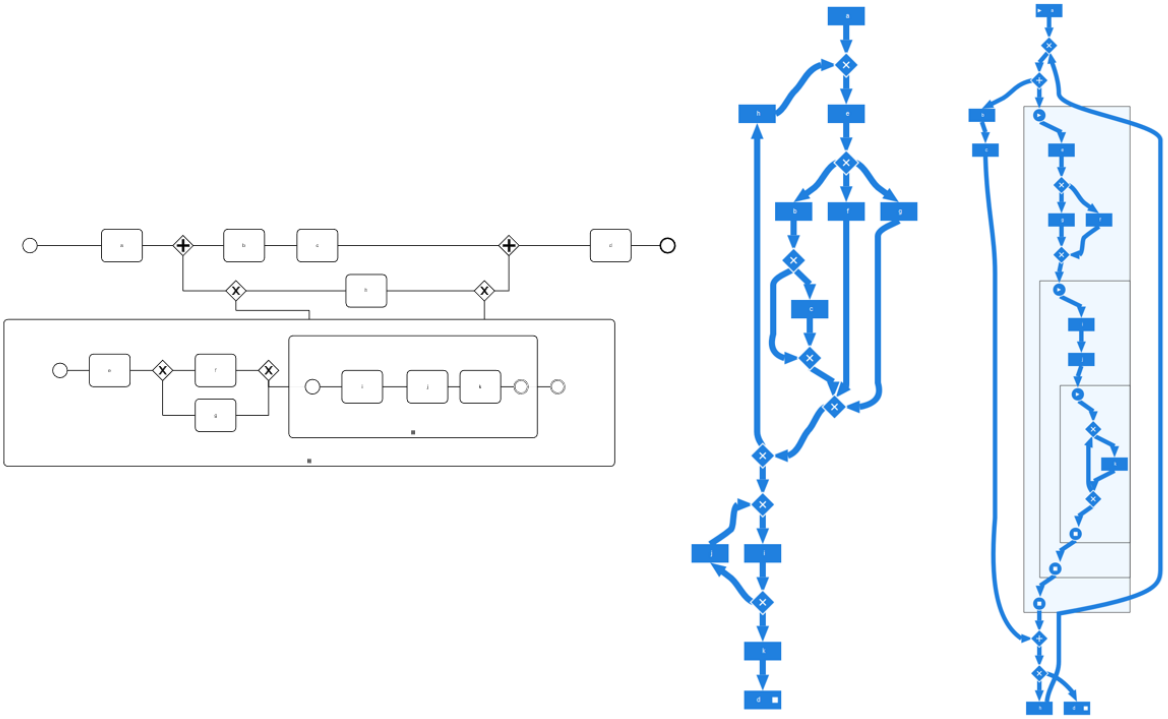


Figure 4.1: A model used to create an artificial log (left), the model discovered by the inductive miner (middle), and the model discovered by the Multi-instance miner(right) of this log.

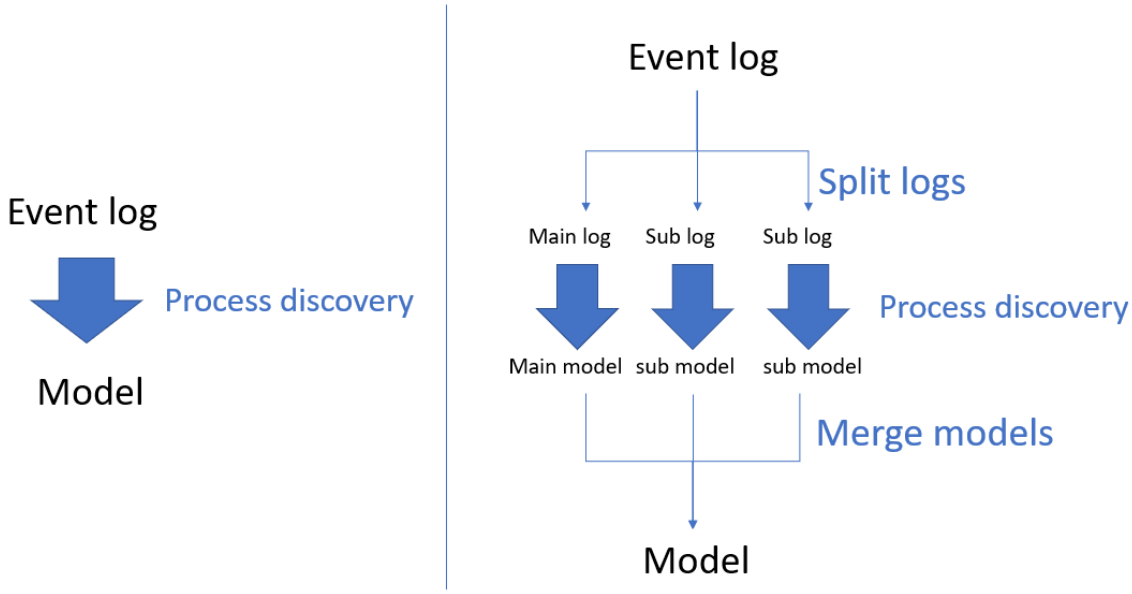


Figure 4.2: A comparison between traditional process mining and the Multi-instance mining method presented in this project.

Algorithm 1 discover the multi-instance process with log L and hierarchy H

```
1: function MultiInstanceMiner(L, H, i)
2:     SL ← getSimplifiedLog(L, H)
3:     model ← RunInductiveMiner(SL, H)
4:     for each child C in H do
5:         model ← Merge(model, MultiInstanceMiner(L, C))
6:     end for
7:     return model
8: end function
```

This approach is represented in recursive Algorithm 1 that takes as arguments an event log L, and a hierarchy H. First using function GetSimplifiedLog we convert L into a simplified log SL. In this step we also incorporate the log splitting step from Section 4.1 where we only get the (sub)log of the current level in the hierarchy. Now we have the simplified log, we use the 'normal' inductive miner to obtain the model corresponding to the simplified log. If H has one or more subprocesses, for each subprocess, i.e. for each subtree with root child C of H, we merge the current model with the model obtained by using Algorithm 1 with arguments L and C.

In the following sections we will first explain the way the log splitting is handled. Second, we will discuss how to mine the split logs. Finally we explain how to merge the resulting models.

4.1 Log splitting

The idea behind the log splitting approach comes from the way we can collapse a subprocess in the BPMN modeling language. In Figure 4.3 a simple model is shown with a multi instance subprocess in the BPMN modeling language. Here, the subprocess can be collapsed into a single activity. This can be used, for example, if a user wants a general overview of the process as a whole and is not interested in the details of the subprocess. This basically provides two ways to view a process with a sub process. The whole process can be shown in one process graph, as shown in the top graph in Figure 4.3. Alternatively it can be shown in two separate process graphs, one of the main process where the subprocess is collapsed, and one of the sub process separately, like the bottom graphs in Figure 4.3. For the log splitting we take a similar approach where we split the log for each level in the hierarchy. The log splitting can be divided into two parts:

1. Omit events that correspond to higher level processes

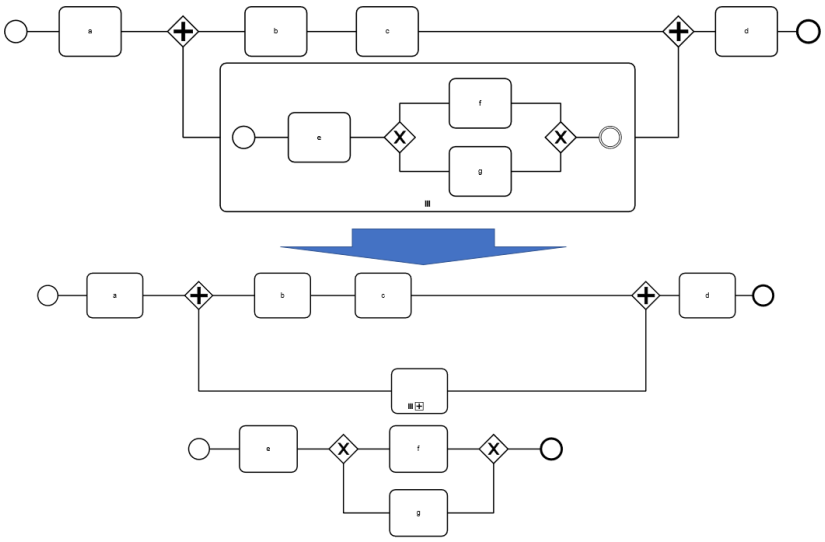


Figure 4.3: A BPMN model of a process with a multi-instance sub process. The subprocess can be collapsed into a single activity block (bottom).

2. Collapse subprocesses

In the case of Figure 4.3, we create a sub log based on the events in the subprocess where we omit all events that correspond to the main process (1) and, create a log for the main process where we need to collapse the subprocess (2). Note that for processes with more complex hierarchies, a process with a nested multi-instance subprocessfor example, we will need to apply both steps for a single level. The log splitting is implemented in the simplification of the event log. Here, the full event log is simplified into the trace variations.

To omit all higher level events we simply use the $subcaseID$ as case identifier. Thereby we effectively filter the log on the $subcaseID$ and filter out the events where the $subcaseID$ has a null value, as events with no case identifier are ignored when simplifying the log. In Figure 4.4 a part of the event log of the process of Figure 4.3 is shown. All the events where the $subcaseID$ does not have a null value are marked red. The sub log contains only the red marked events.

To collapse the subprocesses we cannot just filter out events that do not have a null value for the subcase ID (the reverse of the above). Otherwise it would not be possible to identify where the subprocess needs to be inserted. Instead, we want to 'collapse' each instance of a subprocess into a single event similar to the way in which the subprocess model in the process model is collapsed into a single node in Figure 4.3. We call this activity that repres-

ID	activity	timestamp	subID
0	a	07-01-00 18:24	
0	e	11-01-00 18:09	1
0	e	12-01-00 9:49	2
0	e	13-01-00 21:04	0
0	b	18-01-00 19:13	
0	f	20-01-00 6:04	0
0	f	20-01-00 16:00	1
0	g	30-01-00 5:39	2
0	c	30-01-00 9:39	
0	d	25-03-00 8:21	
1	a	02-01-00 15:28	
1	e	02-01-00 21:00	1000
1	e	03-01-00 12:46	1002
1	g	04-01-00 15:19	1002
1	e	07-01-00 0:38	1001
1	f	08-01-00 2:48	1000
1	b	09-01-00 18:50	
1	c	20-01-00 21:57	
1	g	23-01-00 7:47	1001
1	d	09-02-00 0:36	
2	a	03-01-00 16:36	
2	e	08-01-00 20:01	2000
2	f	14-01-00 11:16	2000
2	b	18-02-00 3:12	
2	c	06-03-00 3:38	
2	d	11-03-00 18:36	

Figure 4.4: An example of a part of an event log of the process model in Figure 4.3. The events with a subcase ID that do not have a null value are marked red.

ents the subcase "MISP" (Multi-Instance SubProcess). However, collapsing the subprocess into a single activity can lead to some ambiguity as to where the subprocess event needs to be inserted into the log. This ambiguity arises because by collapsing the subprocess, we possibly lose completeness of the log.

For example, if we look at the subcase with $subcaseID = 1$ in Figure 4.4, we see that one event in the subcase takes place before the event with activity b and one event takes place after. Therefore, there are two possible ways of inserting the subprocess event. A decision has to be made of where to put the subprocess event if there is ambiguity. Note that there can only be ambiguity if there is concurrency between the subprocess and parts of the main process. As we wish to have a modular solution that is independent of the miner we do not want to solve this in the miner itself. Instead we want to 'set up' the miner in the best way possible to find the concurrency when this behaviour is present in the log. In other words, we want to reduce the loss of completeness, caused by the collapsing of the subprocess, as much as possible.

In this project, we consider three ways of inserting the subprocess event. First, for the sake of simplicity, the subprocess activity is placed at the position of the first event of the subcase. As expected this leads to a bias in the log where the collapsed subprocess was placed more often before the concurrent activities instead of in between or after concurrent activities. When mining the main log on several artificial datasets using this insertion method no concurrency was found between the collapsed subprocess and other activities in the resulting process.

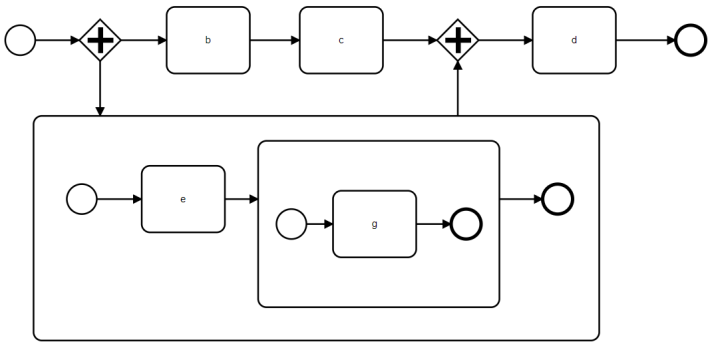


Figure 4.5: An example of a nested multi-instance process. An example of a trace of this process is shown in Figure 4.6b

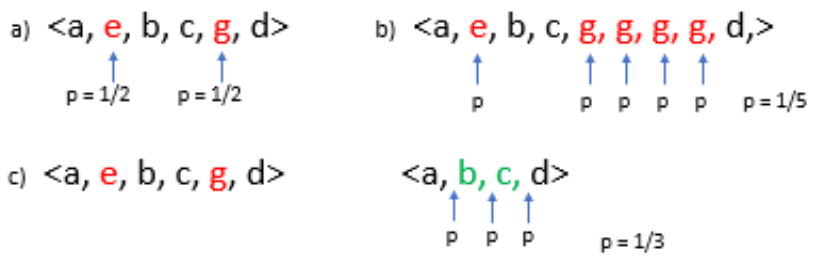


Figure 4.6: Examples of the second (a) & (b) and third (c) insertion methods. (a) the second method inserts the subprocess randomly at one of the positions of one of the activities in the subprocess. In this case before b or after c with probability $p = 1/2$. (b) A nested multi-instance subprocess where multiple instances of activity g are present within the subprocess with activities e and g. Here, the subprocess can be inserted in the same two positions as (a) only with skewed probabilities. (c) the third method inserts the subprocess randomly at one of the possible insertion positions. It keeps track of the main process activities that take place between the start and end of the subprocess (b and c in this case) and inserts it randomly before or after one of these activities. Note that the trace in (b) would get the same insertion probabilities as the trace in (c) for the third insertion method.

The second insertion method inserts the collapsed subprocess at one of the positions of the subprocess events. It does this by keeping track of all the events in the subprocess and picking one of these events to insert the subprocess activity at random with equal probability. This already provided improved results compared to the first method. However, an underlying assumption of this method is that the subprocess activities are distributed equally in the event order. This is not always the case, especially in nested subprocesses. A skewed distribution of subprocess events can consequently lead to a bias in where to insert the subprocess. This was also shown in nested multi instance sub processes to prevent the inductive miner from discovering concurrency. An example of this is shown in Figure 4.6 b. Here we can see that nested multi-instance subprocesses (Figure 4.5) can lead to skewed probabilities as inserting the MISP activity at one of the spots of the g activity would lead to the same insertion position.

The third insertion method aims to solve this problem by keeping track of all possible effective insertion positions(see Figure 4.6) and randomly inserting it in one of the effective positions. This is done by keeping track of the first and last activity in each subprocess. Then we find the activities in the main process (where subcase ID = null) that take place between the start and end of the subprocess (marked green in Figure 4.6c). The effective positions are the positions before or after the activities in the main process that were found in the previous step (arrows in Figure 4.6c). By doing this the subprocess event will be

inserted in one of the effective positions with equal probability. If this behaviour is frequent in the log, each of the three resulting traces will show up in the simplified log with equal frequency. Therefore we 'set up' the inductive miner to find the concurrency between the subprocess and the other activities.

4.2 Mining models

After the log splitting we can mine all the simplified event logs separately for each level in the hierarchy. The main and sub logs are mined using the case ID and subcase ID as case identifiers respectively. The mining algorithm that was used is the inductive miner that takes the simplified event log as input and returns a model.

4.3 Merging models

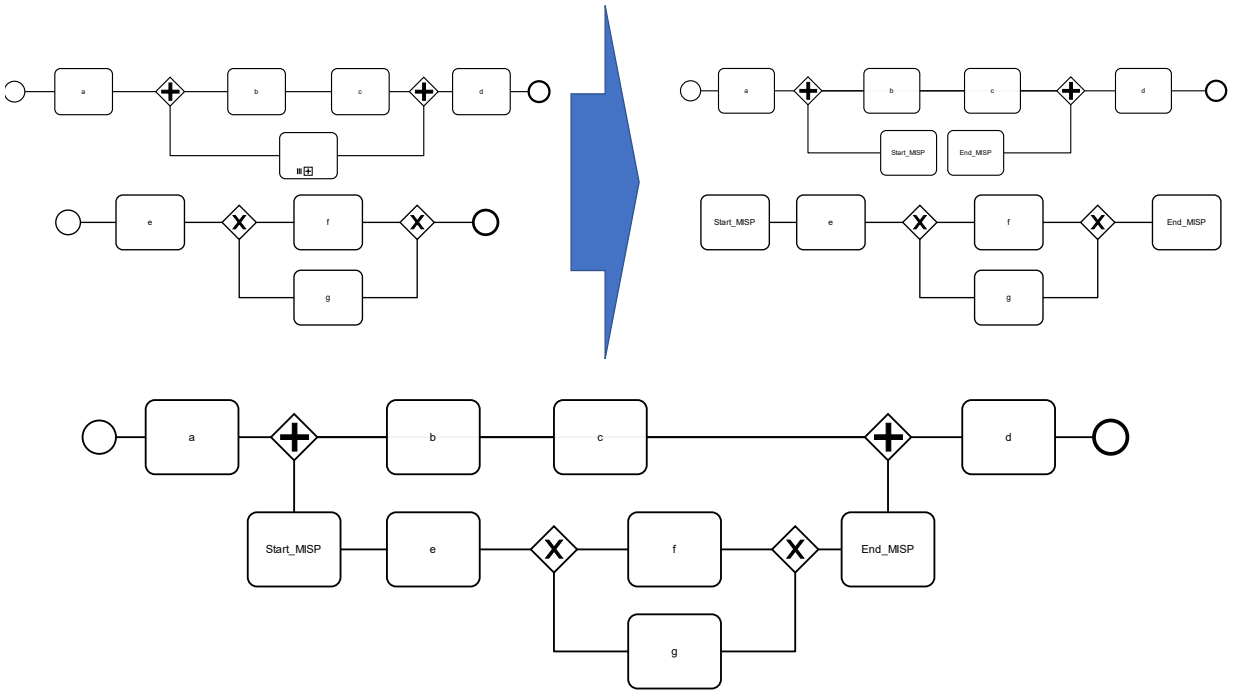


Figure 4.7: The merging of the models. The MISP node in the main process is split into a StartMISP and endMISP node and the start and end nodes of the subprocess are replaced by startMISP and endMISP nodes. The model in the bottom is obtained by combining the two models.

We merge the models two at a time where one model corresponds to the subprocess model and one is the main process model(the model the subprocess needs to be inserted in to). In order to identify where a subprocess starts and ends, we introduce two new node types

to our model: a startMISP node and an endMISP node. The startMISP/endMISP node represent the start/end of a multi instance subprocess respectively. As shown in Figure 4.7, to merge the models, in the main process, we simply replace all to-nodes of the the ingoing edges to the MISP node to startMISP nodes and all the from-nodes of the outgoing edges of the MISP node to endMISP nodes. For the subprocess we simply replace all the start nodes with startMISP nodes and all the end nodes by endMISP nodes. We can now simply aggregate the two models to obtain the combined process model.

Chapter 5

Conformance checking

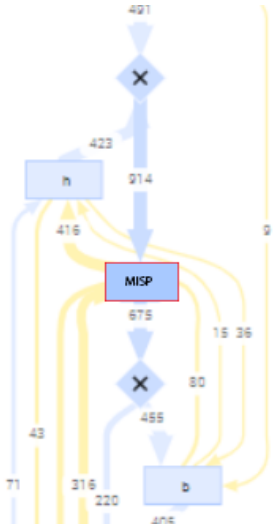


Figure 5.1: An example of non conforming behaviour of a collapsed multi-instance subprocess

Now that we have discussed how we can discover multi-instance processes we can look at how to adapt conformance checking methods to be applicable to these processes. As mentioned in the preliminaries, we already have a traditional process mining method available that is based on Visual Alignments. Similar as in process discovery, the event log is first converted into a simplified log containing the trace variants. Then, each trace variant is mapped on the process graph. For the parts of a trace that are conforming, conforming edges are added. Log-only and model-only edges are added for non-conforming behaviour.

The current implementation cannot just be used for multi-instance processes as it cannot distinguish different instances of subprocesses in the log. Furthermore it has no way to handle the StartMISP and EndMISP nodes we introduced in the previous chapter. In this project we chose not to design a whole new method for computing the visual alignments. Instead, we use a similar approach to process discovery, where we first split the log and the process graph for each level in the hierarchy, get their visual alignments separately and merge them afterwards.

The main advantage of this approach is that we can use the visual alignments method. However, there is also a drawback to this approach. As we separate each part of the process, if there is, for example, non-conforming behaviour regarding the collapsed subprocess activity in the main process, we wont know which parts of the subprocess this corresponds to. In Figure 5.1, a part of a visual alignment is shown where log only edges are present from and to the multi-instance subprocess. Using our method, there is no way of knowing what activities in the subprocess correspond to the log only edges. There is, however, a possible extension to the current method that can provide insights in this. Unfortunately the implementation of this extension was beyond the scope of this project. Nonetheless, we discuss a proposal of what this method could look like in Section 8.5.

Algorithm 2 Get the visual alignments of the multi-instance process with log L, model M, hierarchy H and visual alignments VA

```
1: function GetVisualAlignments(L, M, H)
2:     SL ← getSimplifiedLog(L, H)
3:     PrepareModel(M, H)
4:     MapEventlogToModel(SL, M, H, VA)
5:     for each child C in H do
6:         GetVisualAlignments(L, M, C, VA)
7:     end for
8: end function
```

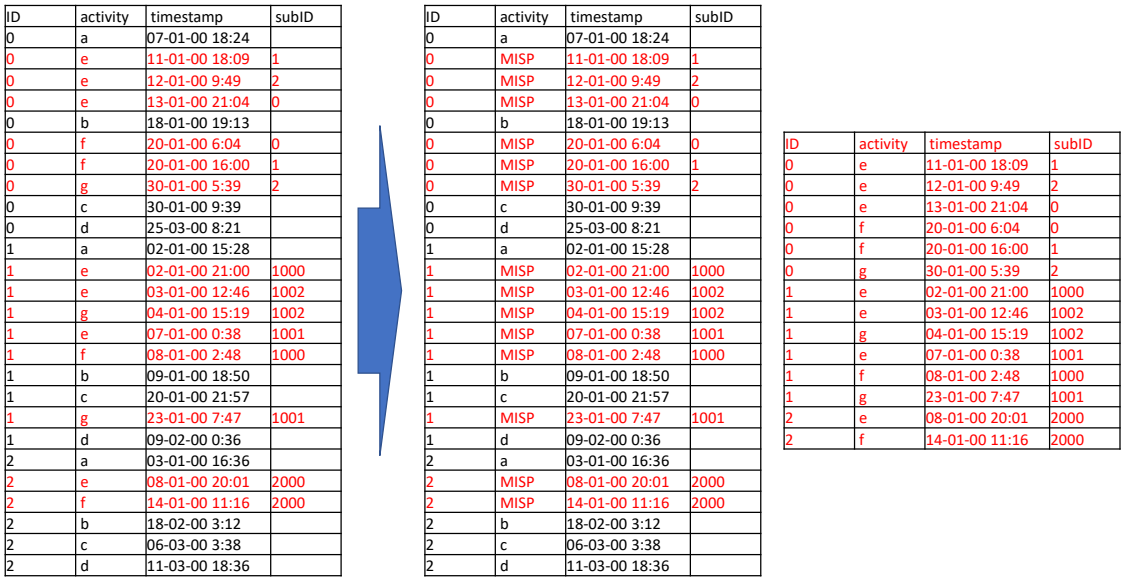
The approach we use is summarized in Algorithm 2. The input of the recursive algorithm is a model M, an event log L, and an (empty) object in which we insert the visual alignments VA. We first need to split both the log and model. First using function GetSimplifiedLog we convert L into a simplified log SL. In this step we also incorporate the log splitting step that we explain in Section 5.1.1. In PrepareModel we extract the model of the current hierarchylevel and do another preparation step as explained in Section 5.1.2. Now we can map the event log to the model using MapEventlogToModel. This method is just the normal method for computing the visual alignments with the incorporation of the merging step that is explained in Section 5.2. If H has one or more

subprocesses, for each subprocess, we perform Algorithm 2 for each subtree with root C of H, thereby traversing down the tree. Once we have reached all leaves, each level will have been mapped. Similar to the previous chapter, we first discuss each step of our approach in the following sections.

5.1 Splitting the log and model

In this section we discuss the splitting of the log and the model.

5.1.1 Log splitting



ID	activity	timestamp	subID
0	a	07-01-00 18:24	
0	e	11-01-00 18:09	1
0	e	12-01-00 9:49	2
0	e	13-01-00 21:04	0
0	b	18-01-00 19:13	
0	f	20-01-00 6:04	0
0	f	20-01-00 16:00	1
0	g	30-01-00 5:39	2
0	c	30-01-00 9:39	
0	d	25-03-00 8:21	
1	a	02-01-00 15:28	
1	e	02-01-00 21:00	1000
1	e	03-01-00 12:46	1002
1	g	04-01-00 15:19	1002
1	e	07-01-00 0:38	1001
1	f	08-01-00 2:48	1000
1	b	09-01-00 18:50	
1	c	20-01-00 21:57	
1	g	23-01-00 7:47	1001
1	d	09-02-00 0:36	
2	a	03-01-00 16:36	
2	e	08-01-00 20:01	2000
2	f	14-01-00 11:16	2000
2	b	18-02-00 3:12	
2	c	06-03-00 3:38	
2	d	11-03-00 18:36	

ID	activity	timestamp	subID
0	a	07-01-00 18:24	
0	MISP	11-01-00 18:09	1
0	MISP	12-01-00 9:49	2
0	MISP	13-01-00 21:04	0
0	b	18-01-00 19:13	
0	MISP	20-01-00 6:04	0
0	MISP	20-01-00 16:00	1
0	MISP	30-01-00 5:39	2
0	c	30-01-00 9:39	
0	d	25-03-00 8:21	
1	a	02-01-00 15:28	
1	MISP	02-01-00 21:00	1000
1	MISP	03-01-00 12:46	1002
1	MISP	04-01-00 15:19	1002
1	MISP	07-01-00 0:38	1001
1	MISP	08-01-00 2:48	1000
1	b	09-01-00 18:50	
1	c	20-01-00 21:57	
1	MISP	23-01-00 7:47	1001
1	d	09-02-00 0:36	
2	a	03-01-00 16:36	
2	MISP	08-01-00 20:01	2000
2	MISP	14-01-00 11:16	2000
2	b	18-02-00 3:12	
2	c	06-03-00 3:38	
2	d	11-03-00 18:36	

ID	activity	timestamp	subID
0	e	11-01-00 18:09	1
0	e	12-01-00 9:49	2
0	e	13-01-00 21:04	0
0	f	20-01-00 6:04	0
0	f	20-01-00 16:00	1
0	g	30-01-00 5:39	2
1	e	02-01-00 21:00	1000
1	e	03-01-00 12:46	1002
1	g	04-01-00 15:19	1002
1	e	07-01-00 0:38	1001
1	f	08-01-00 2:48	1000
1	g	23-01-00 7:47	1001
2	e	08-01-00 20:01	2000
2	f	14-01-00 11:16	2000

Figure 5.2: An example of the log splitting: The subprocess is found by filtering out the events that have a null value for the subcaseID and the main process is found by replacing the activities of the events in the subprocess by the activity "MISP".

In Figure 5.2 an example of the log splitting is visualized of a process with a single multi-instance subprocess. Similar to the previous chapter, the log splitting can be divided into two parts:

1. Omit events that correspond to higher level processes

2. Collapse subprocesses

We perform the first part in exactly the same way as in Section 4.1. For the collapsing we use a similar approach as in Section 4.1. However, in the case of conformance checking we do not wish to lose information by collapsing all subprocess activities into a single activity. Instead we will replace each subprocess activity by the same activity: MISP. The reason for this will become clear in the next subsection.

5.1.2 Model splitting

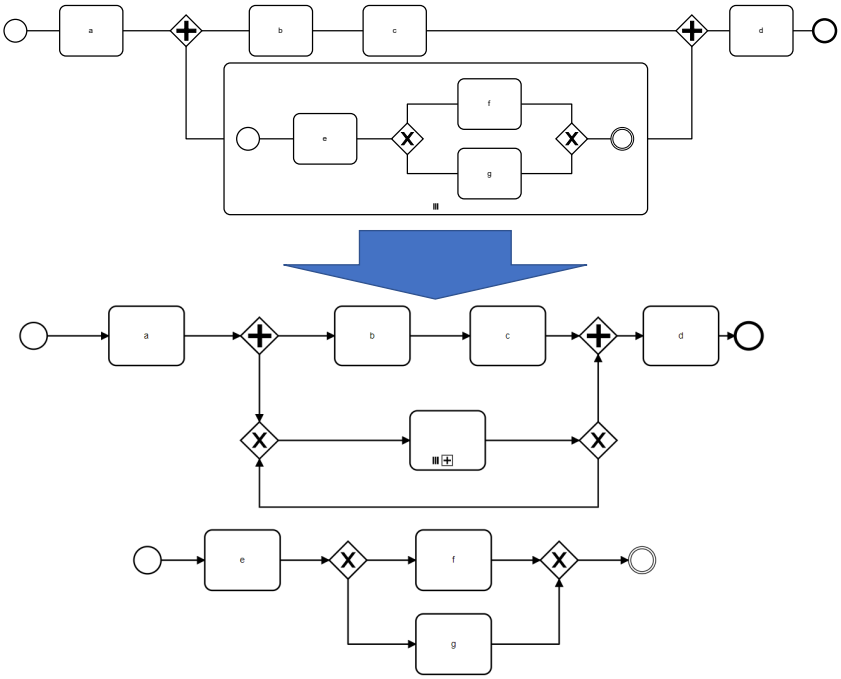


Figure 5.3: An example of the model splittng. In the main process the collapsed subprocess activity is put in a self loop to account for the multiple MISP activites in the log.

We show an example of the model splitting in Figure 5.3. For the subprocess, we simply isolate the subprocess in between the startMISP and endMISP nodes. To collapse a subprocess, we connect the ingoing edges to the startMISP node and the outgoing edges to the endMISP node by a block with a MISP node in a self loop. Hereby we take into account the effect of having multiple MISP activities in the log for each instance of a subprocess. By doing this we can still find non-conforming behaviour if an activity of a subprocess does not take place at the correct moment in the main process like in Figure 5.1.

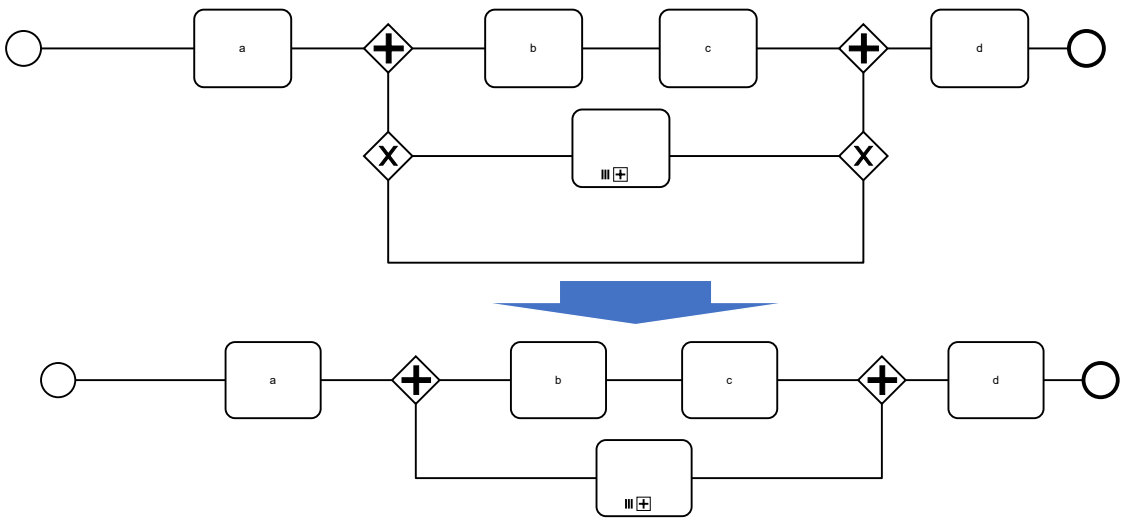


Figure 5.4: The aggregation of the self loop block that was inserted during the model splitting.

5.2 Mapping & Merging process graphs

After separating the log and process graph, we obtain the visual alignments of the log and model of each hierarchy level separately using the visual alignment method that was available. Before we merge the visual alignments, as shown in Figure 5.4, we first aggregate the self loop of the MISP node into a single node. For the merging of the visual alignments we use the same procedure as in Section 4.3.

Chapter 6

Graph visualization

Visualization is key to gaining insights into process models. It is important that a user can get a clear overview of the model based on the process graph. It has to be intuitive and easy to understand. Furthermore, analysis of these processes generally requires interactive filtering to explore the data and the process graph. It is therefore also important that the graph layout is stable under filtering, i.e. that the overall layout remains similar, as to preserve the mental map of the user.[8] Besides filtering, in multi-instance processes, we want the layout to be stable when we collapse a subprocess.

In this chapter, we discuss the visualization of the process graph. This chapter is divided into two sections. In Section 6.1 we explain the different choices that were taken regarding the design of the multi-instance subprocesselements in the process graph. In Section 6.2 we discuss how the layout of the process graph is computed.

6.1 Design

In Figure 6.1, the design of a multi-instance subprocess is shown. The presence of a multi-instance subprocess in a process graph needs to be intuitive and easy to understand. Furthermore it needs to be very clear which parts of the process are part of the subprocess and which parts are not. We therefore use a similar design as in the BPMN notation where the subprocess is surrounded by a frame. The startMISP and endMISP nodes get a similar design as regular start and end nodes.

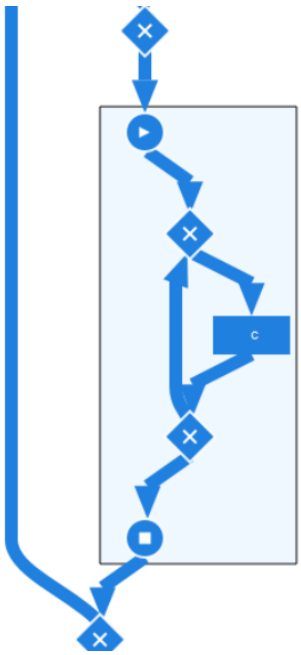


Figure 6.1: The design of a multi-instance subprocessas implemented in the prototype component

6.2 Graph layout

For the layout of the process graph it is important that the nodes and edges in a subprocess are grouped together, such that the subprocesses are easier to identify. Furthermore, if the subprocess nodes are grouped together, if we collapse the subprocess, the graph layout will remain stable.

Algorithm 3 Compute the layout for the Process graph of Multi-instance process P with hierarchy H

```
1: function GetGraphLayout(P, H)
2:     Map[startnode, [translationLeft, translationRight, height]]SizePG
3:     Map[startnode, [graphlayout]] Layouts
4:     for each level in H do
5:         layout ← ComputeGraphLayout(P, level)
6:         for Each subprocess S of current process do
7:             MoveNodesEdges(layout, SizePG[S.startnode])
8:             InsertSubProcesGraph(layout, Layouts[S.startnode])
9:         end for
10:        SizePG[layout.startnode] ← GetDimensionsOflayout(layout)
11:        layouts[layout.startnode] ← layout
12:    end for
13: end function
```

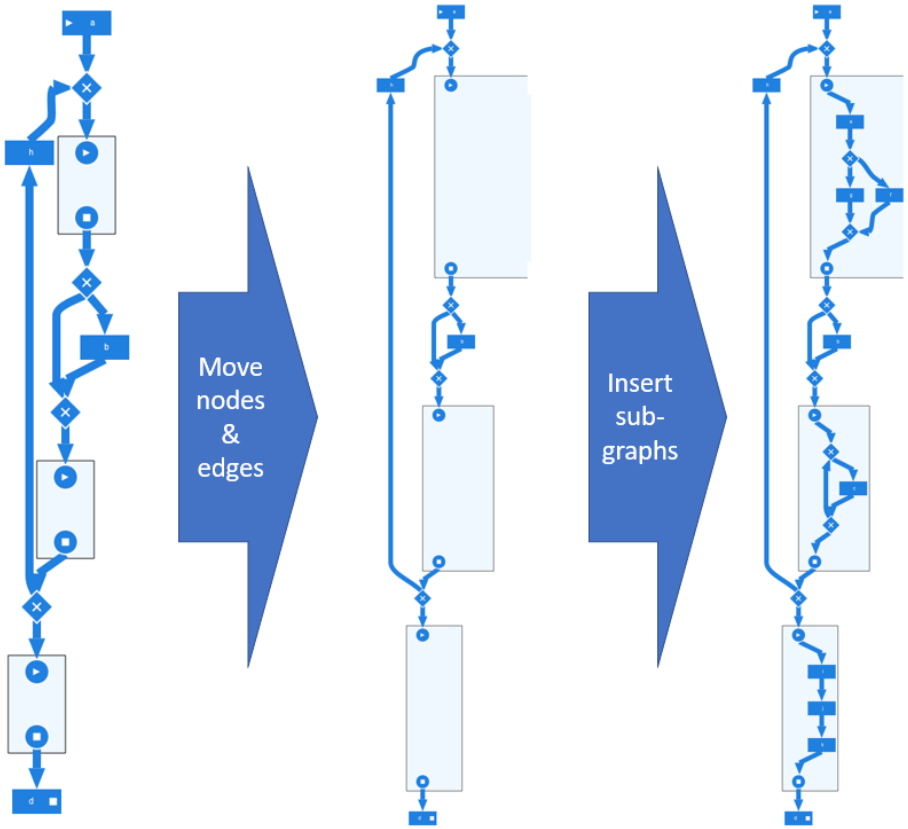


Figure 6.2: An example of the steps we perform for each hierarchylevel, we first move the nodes and edges and then we insert the subgraphs.

To achieve this, our general strategy is to separate the process graphs for each hierarchy level again. Hereby, we effectively ignore all higher level processes and collapse all subprocesses for each level. Now, we can compute the graph layout of each level separately and combine them afterwards. This is shown in Algorithm 3. We start out with a process model, P, containing the nodes, edges, and their attributes, and a hierarchy H.

For each hierarchy level, we can compute the graph layout. The order in which we compute the graph layout for each hierarchy level is in a post order tree traversal in the process hierarchy tree. This ensures that when we insert the process graph of a subprocess later, we already have computed the graph layout of that subprocess.

The graph layout of the current level is computed using ComputeGraphLayout . Once we have the graph layout of the current level, we have to insert the subprocesses of the current level. Before we do this, however, we have to make space in the process graph to insert the subprocess, as shown in Figure 6.2. This is done in MoveNodesEdges, which takes as arguments the current layout and a list containing the size of the subprocess graph layout. Once we have moved the nodes and edges, we insert the subprocess graph using

InsertSubProcessGraph. Once we inserted all the subprocesses we store the dimensions of the current process graph in *SizePG* and store the process graph in *layouts*. Once we finish the last hierarchylevel we have computed the whole graph layout. In the next sections we go through all components of the algorithm.

6.2.1 Compute graph layout

Before we compute the graph layout, we first filter the model so that only the current hierarchy level remains. Thereby we effectively ignore all higher level (sub)processes and collapse all subprocesses. We do, however, include the startMISP and endMISP of the subprocesses of the current level. This is needed later to determine where the subprocess needs to be inserted. Additionally, we add an artificial edge between the startMISP and endMISP nodes to ensure the structure of the graph is sound and that the startMISP and endMISP nodes are connected.

The actual computation of the graph layout was already available in the platform and has not been changed. It uses the global order and ranking of the whole process graph to assign the nodes and paths of edges an x- and y-coordinate.[8] Ideally we would compute the rank and order of each level separately, however we decided that this is beyond the scope of this project. Instead, we use the order and ranking of the whole process which produced results that are satisfactory.

6.2.2 Move nodes and edges

Algorithm 4 Move the nodes and edges of process graph *layout* to make space for inserting a subprocess S with dimensions $translationLeft$, $translationRight$, $height$

```
1: function MoveNodesEdges(layout, S, translationLeft, translationRight, height)
2:     for each node N in layout do
3:         if N.x < S.start.x and (S.end.y < N.y < S.start.y) then
4:             N.x− = translationLeft
5:         end if
6:         if N.x > S.start.x and (S.end.y < N.y < S.start.y) then
7:             N.x+ = translationRight
8:         end if
9:         if N.y ≤ S.end.y then
10:            N.y− = height
11:        end if
12:    end for
13:    for each edge E in layout do
14:        for each control point P in path of edge E do
15:            IPx = InterPolate(P.x, E.from.x, E.to.x)
16:            IPy = InterPolate(P.y, E.from.y, E.to.y)
17:            P.x+ = IPx
18:            P.y+ = IPy
19:            if P.x < S.start.x and (S.end.y < P.y < S.start.y) then
20:                P.x− = translationLeft + IPx
21:            end if
22:            if P.x > S.start.x and (S.end.y < P.y < S.start.y) then
23:                P.x+ = translationRight − IPx
24:            end if
25:        end for
26:    end for
27: end function
```

We move the nodes and edges in the process graph, using Algorithm 4, based on their positions relative to the startMISP and EndMISP nodes of the subprocess that needs to be inserted. If a node is positioned between the startMISP and EndMISP nodes vertically, we move it to the left if the node is left from the startMISP node and to the right if the node is to the right from the startMISP node. All nodes that are positioned below the EndMISP node are moved down. The distance that we need to move the nodes is computed in GetDimensionsOfLayout. The edges consist of a path that contains multiple control points. Each control point is simply an x- and y-coordinate where the edge moves along. The translation of control points is based on the translations of the from- and to-node of its corresponding edge. In Figure 6.3, we show an example of the translation of a control

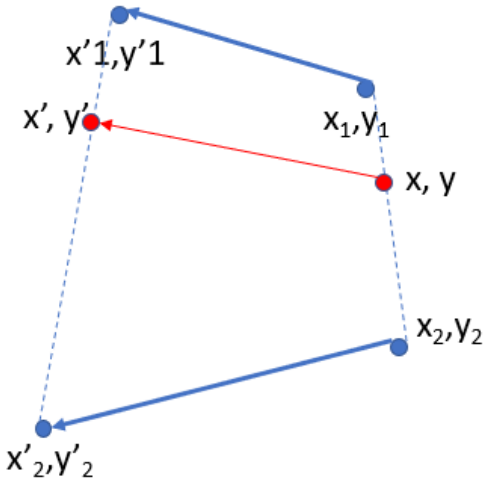


Figure 6.3: An example of the interpolations of points in the paths of edges. If the from-node of the edge is moved from (x_1, y_1) to (x'_1, y'_1) and the to edge is moved from (x_2, y_2) to (x'_2, y'_2) , we move the point in the edgepath from (x, y) to (x', y') using equations 6.1 and .

point. Here, the from-node of the edge is moved from (x_1, y_1) to (x'_1, y'_1) and the to-node of the edge is moved from (x_2, y_2) to (x'_2, y'_2) . Using this, we compute the translation of control point (x, y) with:

$$\Delta x = (1 - \frac{i}{n-1})\Delta x_1 + (\frac{i}{n-1})\Delta x_2, \tag{6.1}$$

$$\Delta y = (1 - \frac{i}{n-1})\Delta y_1 + (\frac{i}{n-1})\Delta y_2, \tag{6.2}$$

Where i is the index of the control point in the path, n is the number of control points in the path, $\Delta x = x' - x$, $\Delta x_1 = x'_1 - x_1$, $\Delta x_2 = x'_2 - x_2$ and $\Delta y = y' - y$, $\Delta y_1 = y'_1 - y_1$, $\Delta y_2 = y'_2 - y_2$. Here we move the control points based on the translation of the from- and to-node of the corresponding edge with a weight based on the relative distance between the control point and the from- and to-node. This also ensures that the first and last control points of the path still latch on to the edges, as we translate the first control point by Δx_1 and the last control point by Δx_2.

Now that we have interpolated the edge positions based on the translation of the from- and to-node most of the edges will have moved in a way that they are not in the space we plan to insert the subprocess. However, to make sure, we check whether the control point is

still in the space that we want to free up. If that is the case, we move the control point to the left or right depending on its relative position to the startMISP node in a similar way that we move the nodes, as shown in Algorithm 4. Here, we compensate for the amount the edge has already been moved.

6.2.3 Insert graph of subprocess

As shown in Figure 6.2, now that we moved the nodes and edges, the subprocess can be inserted by adding the process graph of the subprocess to the process graph of the current level. As both the process of the current level and the subprocess that needs to be inserted include the startMISP and endMISP of the subprocess, we can simply move the whole subprocess such that the control point of the startMISP of the current process and the subprocess are the same.

6.2.4 Get dimensions of layout

We need to store the dimensions of the current layout to know how much space needs to be made available when the current layout needs to be inserted as a subprocess. We do this by finding the minimal and maximal values of the x- and y-coordinates of the control points. The difference between the maximal and minimal y-coordinate is the height of the subprocess: $height = y_{max} - y_{min}$. The amounts we potentially need to translate nodes to the left or right are determined by: $translationLeft = x_{startMISP} - x_{min}$ and $translationRight = x_{max} - x_{startMISP}$.

Chapter 7

Results & discussion

In this chapter, we evaluate the approaches discussed in the previous chapters, using both artificial and real-life dataset. The artificial data is obtained by creating a log based on a predefined multi-instance process model. Since we know the process model the log is based on, we can use the artificial datasets to evaluate the approach for both the process mining and the visual alignment methods. The real-life dataset is data from a pathology process. We use the pathology dataset to answer our fourth research question: How can we obtain new insights (compared to traditional process mining methods) using our methods? Furthermore, for both datasets, we discuss the quality of the visualization of the process graphs. The results from the artificial datasets are evaluated in Section 7.1. In Section 7.2, we evaluate the results from the pathology dataset.

7.1 Artificial datasets

We use two artificial datasets to evaluate the approaches from the previous chapters. The datasets are event logs that are created based on a multi-instance process model that was constructed with the intention of exposing several aspects that our methods might have difficulties with. Both datasets contain 500 cases.

7.1.1 Dataset I

The first dataset we used is based on the multi-instance process model shown in Figure 7.1. This model contains three multi-instance subprocesses, two of which are executed in parallel. Furthermore, there is a loop via activity h back to the multi-instance subprocess containing activities e, f, and g.

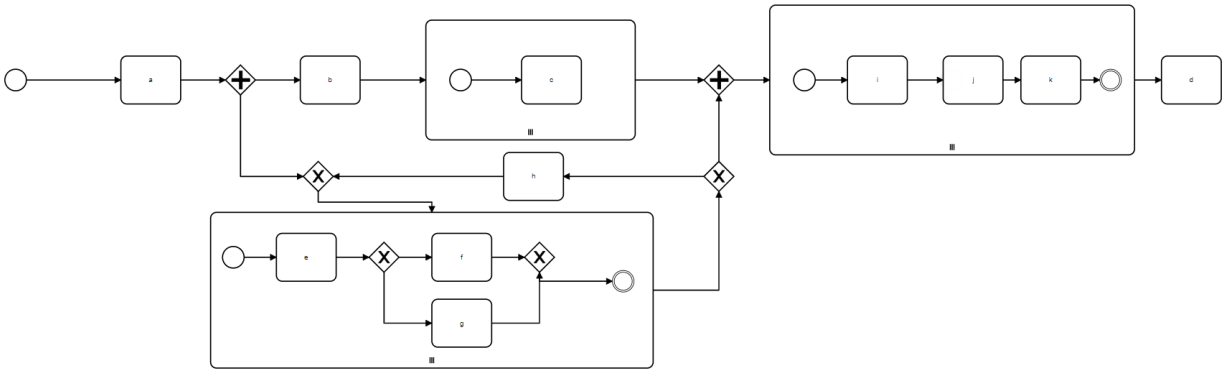


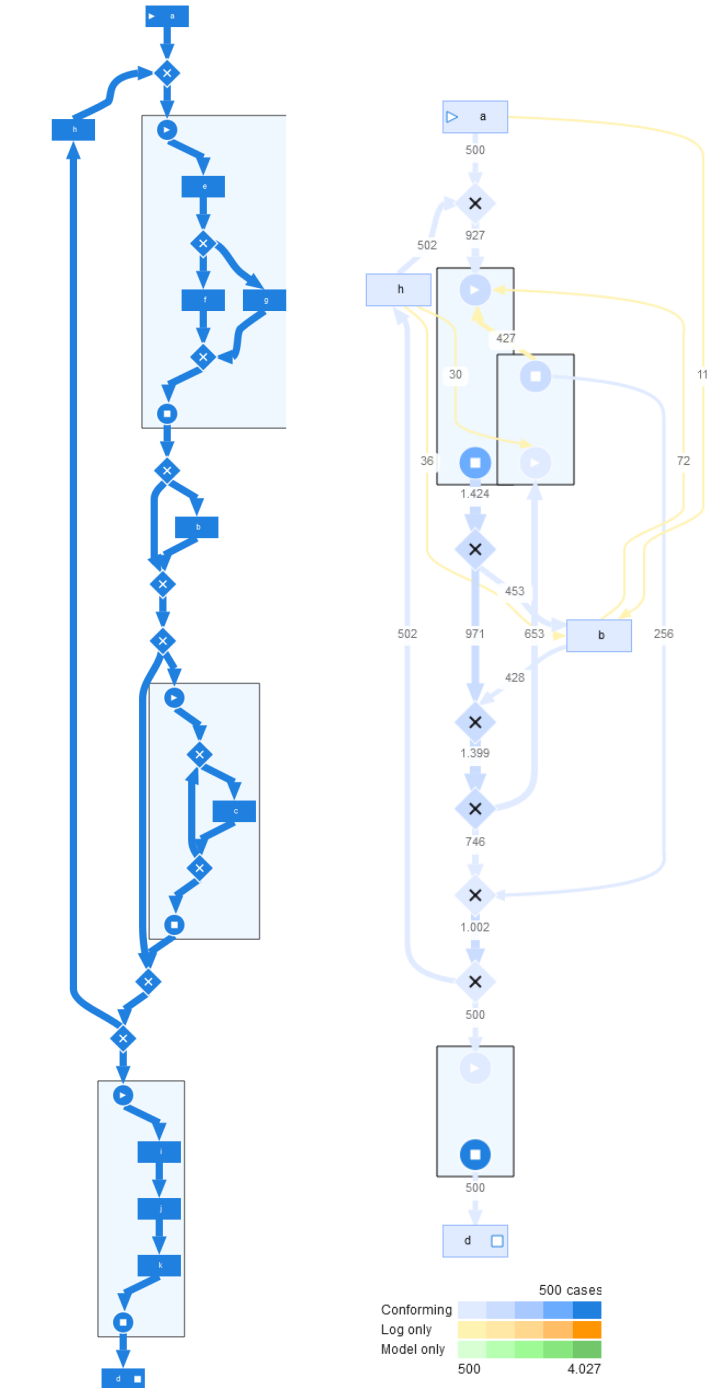
Figure 7.1: The model used to create dataset I

In Figure 7.2a, we show the model obtained using our process discovery method. There are differences between the model in Figure 7.2a and the model we used to produce the event log. First, the subprocess with activity c contains a self loop. This behaviour is not present in the event log as each record with activity c has a unique subcase ID. We also checked the simplified log that was given to the inductive miner and it also gave only a single trace variation, indicating that it is not caused by a bug in the log splitting step. We must therefore assume that this is a mistake from the inductive miner itself.

Second, the parallelism between the two multi-instance subprocessesis not present in the discovered model. Instead, the two parallel branches are in a sequence in the discovered model where activity b can be skipped. This model allows for similar behaviour as the original model. Although, for example, activity a cannot be followed by activity b in the discovered model. The discovered model is also less precise as activity b can be executed multiple times in the discovered process. There can be two reasons that the parallelism from the original process model is not found by our method. First, it could be caused by the way the event logs are generated. Although we checked that the logs are complete(every variation of the process is present in the log), the distribution of activities in the parallel branches in the log could be skewed. This could cause the inductive miner not to pick up on parallel behaviour. Second, it could be caused by the inductive miner itself. For the inductive miner, we used a filter threshold of 1, indicating that there is no noise in the logs (which is the case). So although some indications of parallel behaviour may be rare, the inductive miner should pick up on this behaviour.

The visualization provides a readable process graph. The subprocesses are grouped together and a frame is placed around them. This immediately shows the hierarchical structure of the process. There are no crossings between nodes and edges, or edges and edges, and there is no overlap between edges. Because of this the edges are very traceable.

In Figure 7.2b the visual alignment of dataset I on the model is shown. As there was no non-conforming behaviour in the subprocesses, we collapsed them for a more clear



[a] [b]

Figure 7.2: The process model obtained by using our process discovery method on dataset I (a) and Visual alignment of dataset I on that model (b)

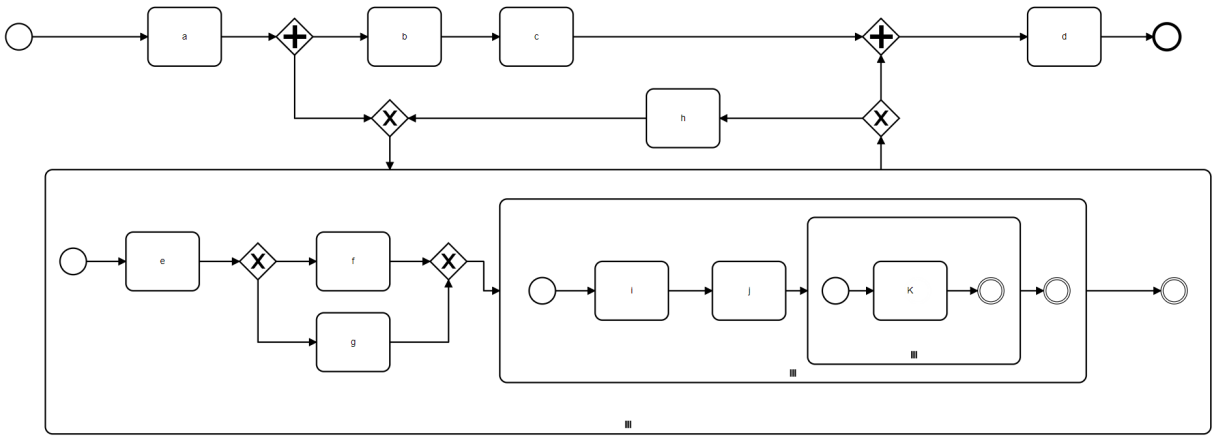


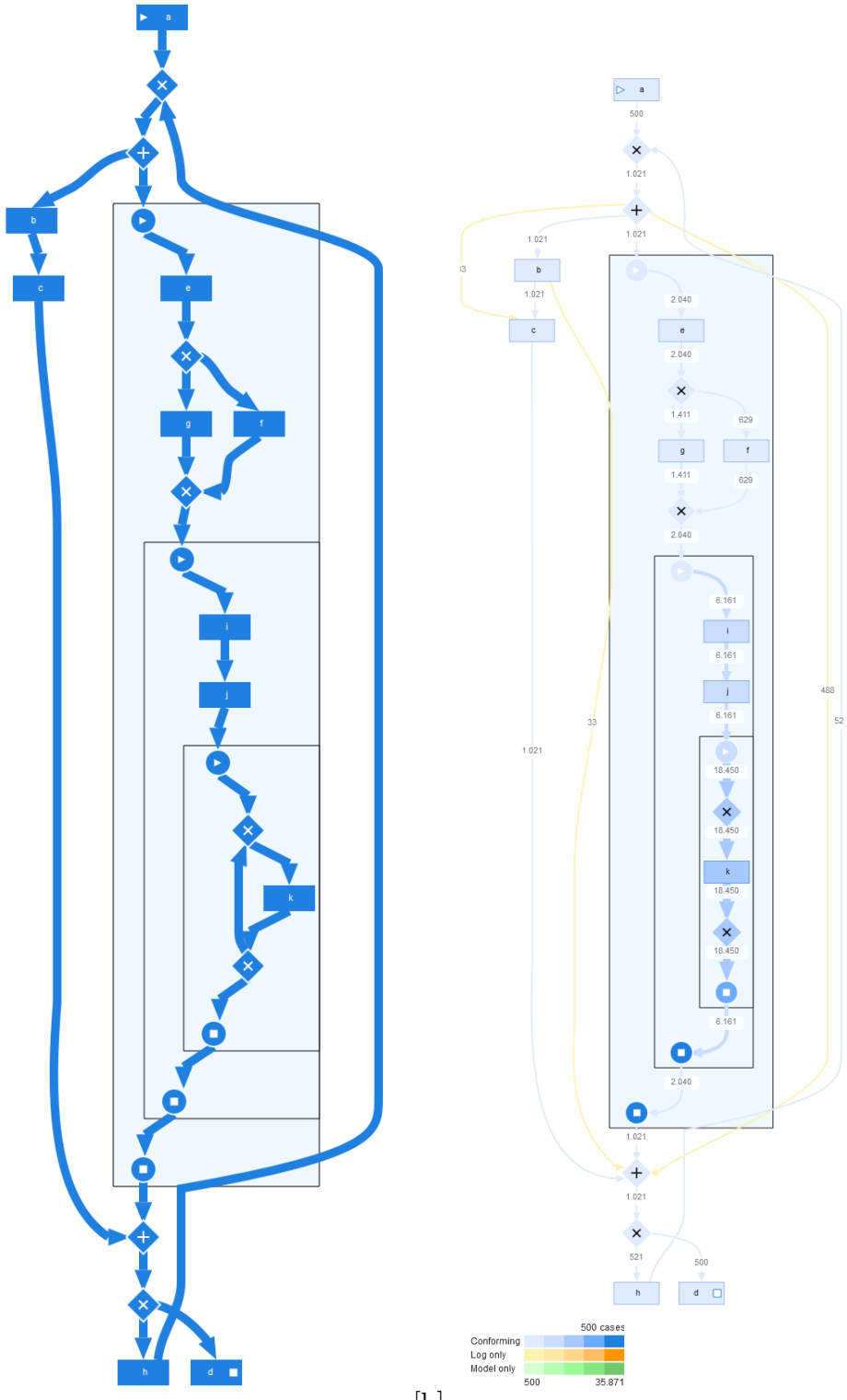
Figure 7.3: The model used to create dataset II

overview of the main process. As expected, we see some non-conforming edges. All of these can be explained by the deviations between the two models. The most common non-conforming edge is where activity c is followed by an activity from the multi-instance subprocess containing activities e, f and g. This is not surprising as those are two parallel multi-instance subprocesses. The other non-conforming edges can be explained in a similar fashion.

7.1.2 Dataset II

The second dataset we used is based on the process model in Figure 7.3. This model contains three nested multi-instance subprocesses. Furthermore, there is a loop from the end of the multi-instance subprocess back to the start of the multi-instance subprocess with activity h in the loop body. Additionally, there is a parallel branch between the multi-instance subprocess and activities b and c.

In Figure 7.4a, the model obtained by using our process discovery method is shown. There is only one difference between the model in Figure 7.2a and the model we used to produce the event log. The main difference is that the loop with activity h now includes both parallel branches while it should be inside the parallel branches. This is probably a mistake of the inductive miner as it is known to have difficulties with loops.[6] Furthermore, similarly to the multi-instance subprocess with activity c in the process from dataset I, the multi-instance subprocess with activity k has a self loop. Again, we suspect this is a mistake from the inductive miner.



[a] [b]

Figure 7.4: The process model obtained by using our process discovery method on dataset II (a) and Visual alignment of dataset II on that model, visualized in UiPath Process Mining(b)

In Figure 7.4b, the visual alignment of dataset II on the discovered model is shown. Here, we see that there are only three non conforming edges in the visual alignment. All three can be explained by the incorrect position of the loop. The non-conforming edges on the left and middle happen when, first the multi-instance subprocess is executed in parallel with activity b, then is looped back via activity h and then the multi-instance subprocess is executed in parallel with activity c. The non-conforming edge on the right occurs every time we skip activity b and c as those activities are only executed once per case.

The visualization of both the model and the visual alignment give a clear overview of the processes. The hierarchical structure of the process is clear. There are no crossings between nodes and edges, and there is almost no overlap between edges. There are a few edge crossings but in all cases it is clear what edge goes where.

7.2 Pathology dataset

7.2.1 Description data

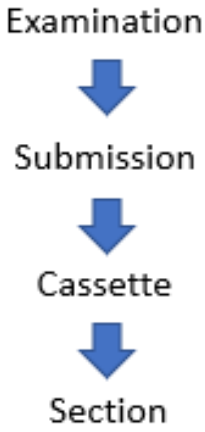


Figure 7.5: The process hierarchy of the pathology process. Each examination has multiple submissions, each of which has multiple cassettes, each of which has multiple sections.

The pathology dataset consist of a process where one or multiple samples are taken in an examination and those samples are analysed for a multitude of tests. As shown in Figure 7.5, the process has a nested hierarchy, where each examination has one or multiple submissions, corresponding to the number of samples that were taken. These samples are then divided over one or more cassettes. These cassettes are then divided into one or more sections. Each examination, submission, cassette, and section has a unique ID in the dataset, satisfying our assumption that the subcase ID is found explicitly in the data. The dataset contains 42.949 cases with 1.048.575 records. There are, however, some records

with a missing cassette ID. For these records, we use the section ID as the cassette ID. We do this because, by the process hierarchy, we need each record with a section ID to also have a cassette ID. Otherwise, activities from the section level would end up in the sub log of the submission level as they have no cassette ID.

7.2.2 Results

In Figure 7.6, we show the process model that is found using our process discovery method. Overall, the model gives a clear overview of the process. There is one structure we suspect is not found by the process discovery method: In the cassette subprocess, activities *cassette extra aanvraag*(request extra cassete) and *cassette extra snijden*(cut out extra cassette) are parallel to the section multi-instance subprocess and can be skipped. We suspect that these two activities should loop back to the section multi-instance subprocess as a new cassette is requested.

The visualization of the model gives a clear overview of the model. The hierarchical structure of the process can easily be identified by the frames around the subprocesses. There are no crossings between nodes and edges, or edges and edges, and there is no overlap between edges. Because of this, the edges are very traceable. However, there are also some issues with the layout. The edge from activity *inzending uitsnijden* to the parallel gateway has a unnecessarily complex path. This is caused by the MoveNodesEdges step where first the edge is interpolated and then it is checked whether the control points of the edge still overlap with the process that needs to be inserted. We suspect that after the interpolation the path of this edge was a straight line from the activity *inzending uitsnijden* to the parallel gateway. Some control points were sufficiently moved by the interpolation and others not. Therefore, the control points that were not sufficiently moved are translated to the right, in this case, causing a 'bump' in the edge. Another issue with the layout is that the activity boxes extend over the subprocess frames in some cases. This is caused by the fact that when moving the nodes and edges, and drawing the frames around subprocesses, the standard node size is used. However, for activities with long names, the node size is larger.

In Figure 7.7, we show the visual alignments obtained by mapping the pathology dataset onto our discovered model. To make them easier to interpret, we separated the visual alignments for each hierarchylevel. Note that we could not use the visualization method discussed in Chapter 6, due to the large size of the dataset. Therefore, we will not evaluate the visualization of the visual alignments of the pathology dataset.

There are a lot of non-conforming edges in the visual alignment of the Examination level. This can be caused by several things. The event log can contain noise. Furthermore, there is the possibility that the log contains cases that have been cut off, i.e., the log contains

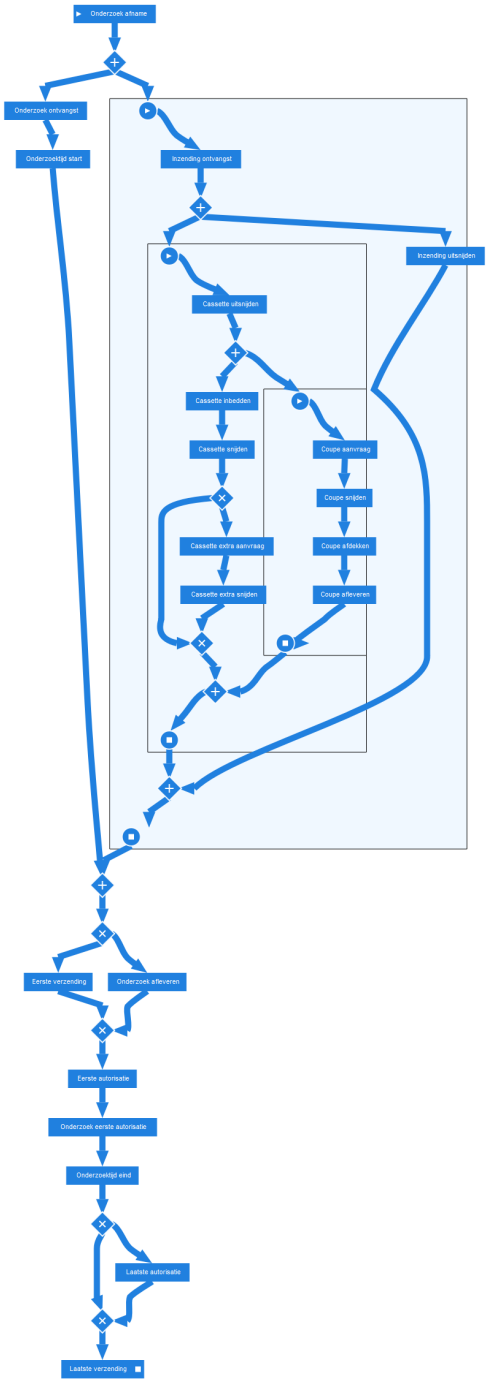


Figure 7.6: The model obtained by our process discovery method on the pathology dataset.

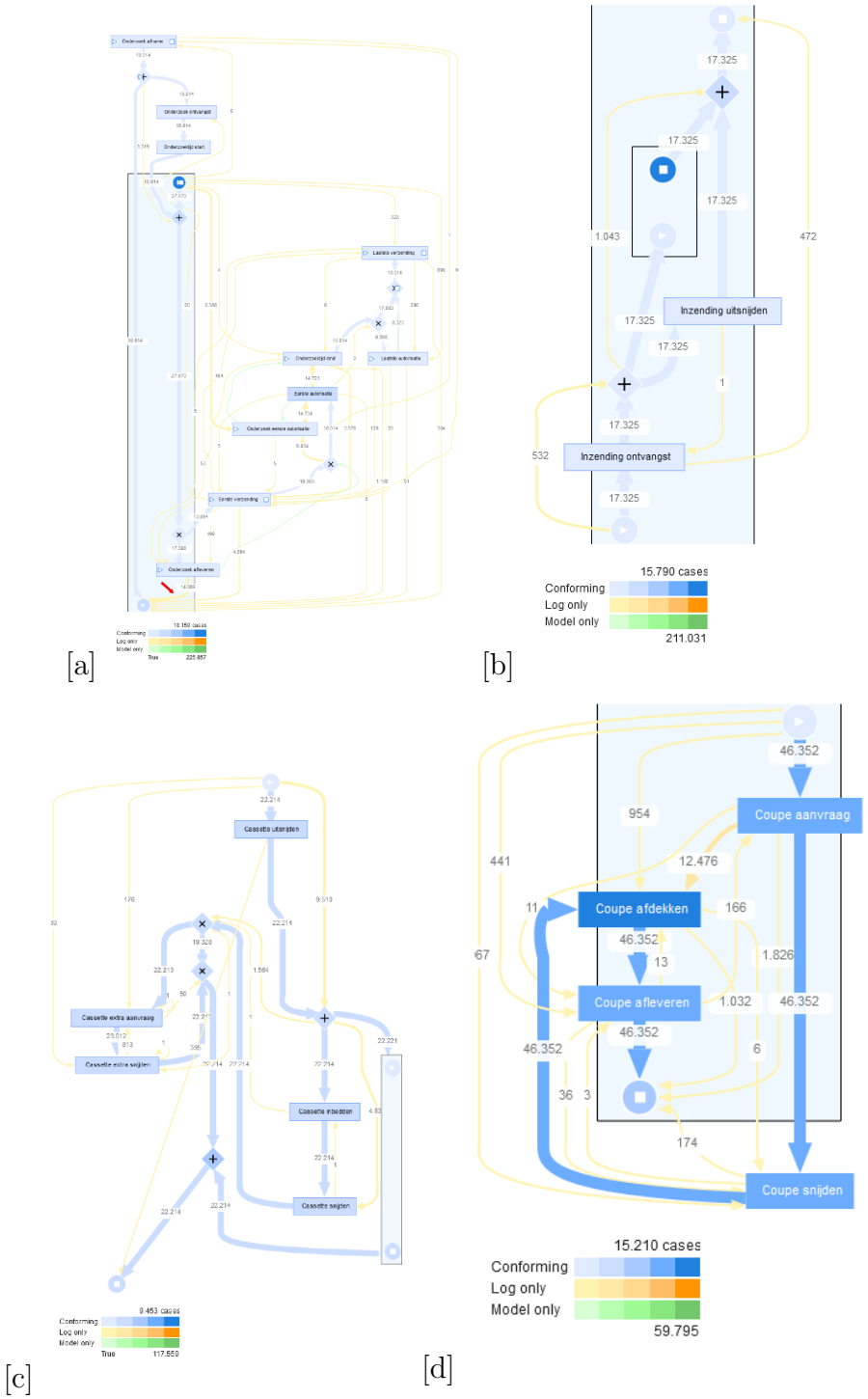


Figure 7.7: The visual alignments obtained by mapping the pathology dataset onto our discovered model, separated by hierarchylevel.

cases that have either not been completed or that already have been instantiated before the start of the log. Another explanation is of course the inaccuracy of the obtained process model. The non-conforming edge between *onderzoek afleveren* and the startMISP node (indicated by the red arrow) suggests that this activity might also take place in parallel with the multi-instance subprocess.

The visual alignment of the Submission level has relatively few non-conforming edges. Furthermore, the weight of the non-conforming edges (max 1.043) is much lower than the weight of the conforming edges (17.325). Therefore, we assume that the model obtained by our process discovery algorithm is correct and that the deviating behaviour is noise or rare exceptions in the log.

The visual alignment of the Cassette level shows some deviations between the log and model. The largest deviations are that the activities *cassette uitsnijden* and *cassette inbedden* get skipped quite a lot in the log but not in the model. This can either be a logging mistake or our model should have the option of skipping those activities.

Similarly to the Cassette level, the visual alignment of the Section level has only a deviation where an activity is skipped, namely *coupe snijden*. Again this can either be a mistake in the logging or behaviour that our model should allow for.

7.2.3 Comparison with traditional methods

In Figure 7.8, we show the model obtained by applying the traditional inductive miner on the pathology dataset. The process model is very complicated with several loops and possibilities of skipping activities. This is caused by the fact that the inductive miner cannot distinguish the different instances of the subprocesses. This model shows why our methods are needed. Traditional methods provide models that are not only overcomplicated, but also give an inaccurate depiction of the process. Using our methods, we can provide a process model that is much simpler, fits the data more accurately, and shows the hierarchical structure of the data in a clear way.

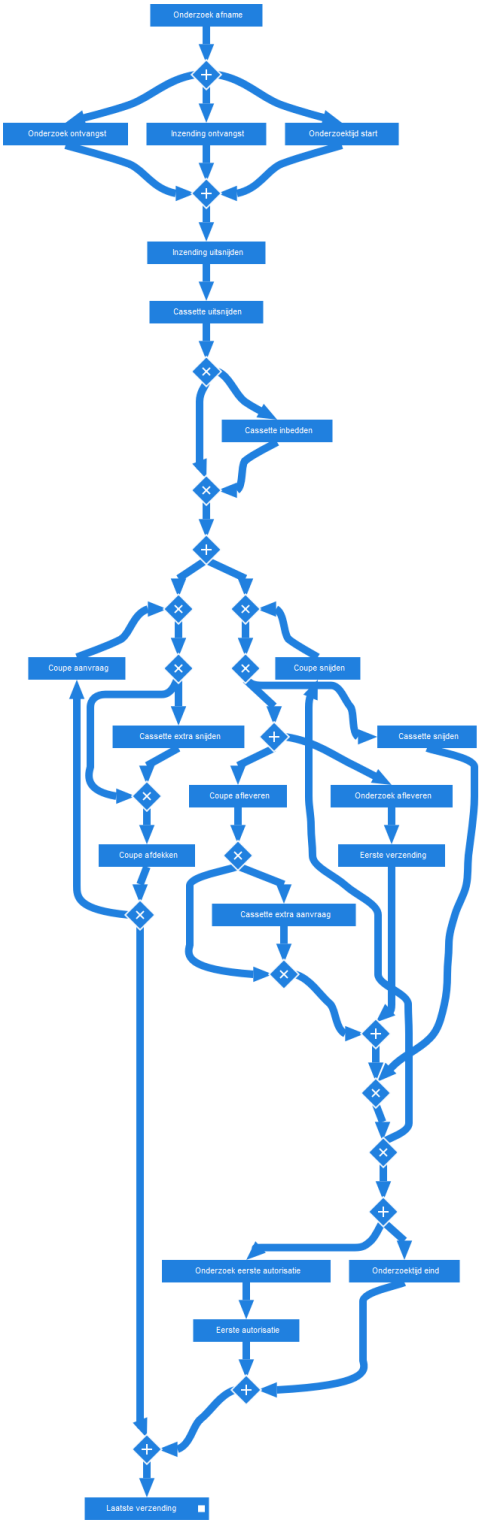


Figure 7.8: The model discovered by the "traditional" inductive miner from the pathology dataset.

Chapter 8

Conclusion

In this chapter we discuss the results from the previous sections. We divide this chapter into four parts. First, we discuss the process discovery method. Second, we discuss our method for the visual alignments. Third, we discuss the visualization of the process graph. Fourth, we discuss the comparison between our method and traditional methods. Finally, we discuss the future developments of this work.

8.1 Process discovery

For process discovery, we separated the log for each level in the process hierarchy. This allows us to use the traditional inductive miner to get a model for each level. After merging these models, we obtain our multi-instance process model. Our approach of separating the log for each hierarchy level, mining the models separately, and merging them afterwards, solves the problems of traditional methods: We use the relevant subcase ID to differentiate the instances of subcases. Furthermore the introduction of the startMISP and endMISP node types gives us the possibility of expressing that a part of a process is a multi-instance subprocess. This is also seen in the results for both the artificial datasets and the pathology dataset. However, from the artificial datasets we can see that there are some differences between the model the dataset was based on, and the model found with our process discovery methods. These deviations most likely arise from the inductive miner itself. Therefore, we conclude that our approach does work, but is only as strong as the underlying traditional process discovery methods.

8.2 Visual alignments

We separated the log and model for each hierarchy level. Then we got the visual alignments separately and merged the visual alignments afterwards. Our approach for the visual alignments solves the limitations of traditional methods. As, similar to the process discovery method, our method for visual alignments can differentiate different instances of subcases by their respective subcase ID and is able to cope with the startMISP and endMISP nodes in our models with multi-instance subprocesses. This is also confirmed by our results for the artificial datasets. The visual alignments show non-conforming edges that are expected from the differences between the models the datasets were based on and the discovered models.

The visual alignment of the pathology dataset also exposed some deviations between the event log and the process model. These can either be caused by noise in the event log (like mistakes in the logging) or by mistakes in the discovered model.

As discussed in Chapter 5, the key limitation of our approach is that, if there is non-conforming behaviour within a multi-instance subprocess, we cannot see which activities within the multi-instance subprocess this non-conforming behaviour corresponds to. To be able to do this, our approach would need to be extended. A proposal for this is discussed in Section 8.5.

8.3 Visualization

For the visualization, similarly to the other methods, we computed the graph layout for each hierarchy level and merged them afterwards. This allows for models where the subprocesses grouped together in a single block. This also guaranties that the layout will be stable under the collapsing of a subprocess. We used a design where subprocesses are surrounded by a frame. Because of this, the different hierarchylevels of the process can be easily identified. However, in some cases our approaches causes edges to follow overcomplicated paths. Although our approach already provides process graphs with clearly recognizable multi-instance subprocesses, it needs to be extended further for optimal visualization.

8.4 Comparison with traditional methods

As shown in the previous chapter, traditional methods cannot give a sufficient overview of multi-instance processes, as they cannot express, or deal with the hierarchical structure of the process. Therefore, we conclude that our methods can be used to not only obtain new insights compared to traditional methods, but are also needed to be reliable at all.

8.5 Future work

As with all research, our research has several limitation. In this section, we propose directions that could be taken for future work.

The first limitation comes from our assumption that the hierarchy is known. If our other assumption is true, namely that each subcase has an attribute in the event log that can be used to identify each instance of that subcase, the hierarchy is already implicitly present in the data. Instead of requiring a manual input of the process hierarchy, a preprocessing method could be implemented to find the hierarchy from the event log. This has already been investigated by Conforti et al[5], who use an approach inspired by the notions of *key* and *foreign key* from relational databases to find the process hierarchy from an event log. This approach could also be implemented in our research.

The limitation of our work that is arguably most important, is that our approach for conformance checking can only be applied to one level in the hierarchy at once. As we discussed before, this leads to several limitations for checking the conformance of multi-instance process models. However, our approach could potentially be extended to solve this problem. We provide an outline of what this approach could look like. Instead of changing the activities in subprocesses to 'MISP' in the log splitting step, we could instead label those activities as 'MISP' activities but keep the original activities in the log. Now we can use this labeling in the computation of the visual alignments to make sure that these activities are mapped onto the 'MISP' task in the model. However, when we store the record of a mapped edge we can store the original activity instead of the 'MISP' activity, solving the problem. However, the visual alignments method would have to be adapted quite a bit to implement this approach. Therefore, we decided the implementation of this approach was beyond the scope of this project.

In an alternative approach to this research, it could be investigated how the inductive miner can be extended to discover multi-instance processes by itself.

The visualization of multi-instance processes and visual alignments of multi-instance processes could be extended in multiple ways. Multiple designs could be tested to visualize multi-instance subprocesses. Furthermore, the collapse of a subprocess should be adapted in a way that is more intuitive. Lastly, multi-instance processes should be integrated in Tracy[8], so the graph layout could be computed taking the process hierarchy into account.

Bibliography

[1] Uipath process mining. https://www.uipath.com/product/process-mining. 12

[2] Modern Business Process Automation. Yawl and its support environment. *Modern Business Process Automation: YAWL and its Support Environment, Springer*, 2010. 8

[3] B. d. Bie. Visual Conformance checking using BPMN. Master's thesis, 2019. 13

[4] JCAM Buijs. Flexible evolutionary algorithms for mining structured process models. *Unpublished Ph. D. Thesis, Eindhoven University of Technology, Netherland*, 220, 2014. 8

[5] Raffaele Conforti, Marlon Dumas, Luciano García-Bañuelos, and Marcello La Rosa. Bpmn miner: Automated discovery of bpmn process models with hierarchical structure. *Information Systems*, 56:284–303, 2016. 53

[6] Brons D. Discovering precise and understandable process models by enhancing the Inductive Miner. Master's thesis, 2019. 12, 43

[7] Sander JJ Leemans, Dirk Fahland, and Wil MP van der Aalst. Discovering block-structured process models from event logs containing infrequent behaviour. In *International conference on business process management*, pages 66–78. Springer, 2013. 11, 12

[8] Robin JP Mennens, Roeland Scheepens, and Michel A Westenberg. A stable graph layout algorithm for processes. In *Computer Graphics Forum*, volume 38, pages 725–

737. Wiley Online Library, 2019. 13, 33, 36, 53

[9] OMG Business Process Model. Notation (bpmn). object management group, formal. 2011. 5, 8

[10] Wil Van Der Aalst. Data science in action. In *Process mining*, pages 3–23. Springer, 2016. 5, 9, 11

[11] Wil MP Van der Aalst, Vladimir Rubin, HMW Verbeek, Boudewijn F van Dongen, Ekkart Kindler, and Christian W Günther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software & Systems Modeling*, 9(1):87, 2010. 11

[12] Wil MP van der Aalst, Kees M van Hee, and GJPM Houben. Modeling workflow management systems with high-level petri nets. In *Proceedings of the 2nd Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms*, pages 31–50, 1994. 8

[13] AJMM Weijters and JTS Ribeiro. Flexible heuristics miner (fhm). In *2011 IEEE symposium on computational intelligence and data mining (CIDM)*, pages 310–317. IEEE, 2011. 11