Eindhoven University of Technology

MASTER

# Using trace clustering for configurable process discovery explained by eent log data

van Oirschot, Y.P.J.M.

*Award date:*
2014

Link to publication

**Disclaimer**
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

**General rights**
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Technische Universiteit
Eindhoven
University of Technology

Master of Business Information Systems,
Department of Mathematics and Computer Science

# Using Trace Clustering for Configurable Process Discovery Explained by Event Log Data

*Master Thesis*

Author:
ing. Y.P.J.M. van Oirschot

Supervisors:
dr. ir. B.F. van Dongen
ir. J.C.A.M. Buijs
dr. ir. R.M. Dijkman

Eindhoven, August 2014

**Abstract**

Existing configurable process discovery techniques discover models at an organizational level, i.e., directly from the event logs of different organizations. In this thesis we propose an approach that allows for the discovery of models at a behavioral level, i.e., we only consider the observed behavior in the traces and not their origin. Our full approach consists of three steps.

The first step is to find groups of behaviorally similar traces. In order to do this we use hierarchical trace clustering. Clustering algorithms often require a dissimilarity measure, existing approaches towards defining a dissimilarity measure do not include model-awareness and therefore often have difficulties dealing with the order of execution and loops. We propose a new dissimilarity measure, the Syntactic reasoning approach, that overcomes these issues by using a reference process tree.

The second step is to provide insights to the end-user of our full approach in why traces are grouped. We do this with the data of traces and events in such a group. In this thesis we explain how to annotate a hierarchical clustering with data annotations using the naive Bayes or C4.5 decision tree classifier. In real-life situations hierarchical clusterings may become huge. Many methods exist to reduce such hierarchies, these however do not take into account the data annotations. We therefore propose a reduction algorithm that performs reduction based on the data annotations. Furthermore, situations exist where the event log does not contain data that explains behavior, in these cases we cannot annotate the hierarchical clustering using the data from the event log. To solve this we explain how event logs can be enriched with data attributes that abstract from the traces.

The third step is to create a configurable process model from a selection of groups of traces in the hierarchical clustering. In this thesis we focus on process trees. Before this work the only way of discovering configurations for a process tree was a brute force approach. In this thesis we propose a new structured approach that discovers configurations for any process tree and a collection of event logs.

Through experiments on artificial, random generated and real-life event logs and process trees we show that the proposed approaches are effective and applicable in different scenarios.

**Acknowledgments**

First of all, I would like to thank Joos Buijs and Boudewijn van Dongen for their excellent guidance during this graduation project and their in depth feedback on my thesis. I would also like to thank them for the provided insights and their interesting ideas that inspired the work I present in this thesis. Furthermore, I would like to thank Joos Buijs for his flexible availability and his support in the uses of the Evolutionary Tree Miner framework and related techniques.

Second, I would like to express my gratitude to all the speakers of the weekly Thursday meetings of our research group. The presentations during this meeting provided a peak into what is active in the field of process mining and were very interesting. They gave me some deeper insight into the world of academia and sparked some new ideas.

Last but not least, I would like to thank my girlfriend, my parents and brothers, and my friends for their support and understanding in the last six months and the rest of my master study. Their understanding of the busy and long days really helped me staying focused.

Yoran van Oirschot

# Contents


**List of Figures** **vii**

**List of Tables** **xi**

**1 Introduction** **1**
1.1 Running example . . . 2
1.2 Research goal . . . 4
1.3 Outline . . . 5

**2 Preliminaries** **7**
2.1 Event logs . . . 7
2.2 Process trees . . . 8
2.3 Configurable process trees . . . 9
2.4 Alignments . . . 10
2.5 Measuring the quality of a process tree . . . 11
2.6 Evolutionary Tree Miner . . . 12
2.7 Data mining: Clustering and classification . . . 13

**3 Trace clustering** **15**
3.1 Related work . . . 16
3.1.1 Feature-set approaches . . . 16
3.1.2 Syntactical approaches . . . 16
3.2 Adding model-awareness . . . 17
3.3 The Syntactic reasoning approach . . . 17
3.3.1 Partially ordered alignments . . . 18
3.3.2 Relating partially ordered alignments . . . 19
3.3.3 Computing parallel execution difference . . . 20
3.3.4 Calculating dissimilarity for a single model . . . 21
3.3.5 Aggregating dissimilarity for multiple models . . . 22
3.4 Creating a segregation of traces . . . 23
3.5 Conclusion . . . 24

**4 Explaining clusters by data** **25**
4.1 Annotating a hierarchical clustering . . . 26
4.2 Reducing an annotated hierarchy . . . 28
4.3 Enrichment of trace data . . . 30
4.4 Conclusion . . . 30

**5 Configurable process discovery** **31**
5.1 Related work . . . 33
5.2 Discovering process tree configurations . . . 35
5.2.1 Mapping alignments . . . 35
5.2.2 Hiding and blocking . . . 35
5.2.3 Downgrading operators . . . 37
5.2.4 Loop iteration partitioning . . . 38

5.3 Evaluating configurable process trees . . . 39
5.4 Conclusion . . . 40

**6 ProM implementation 41**
6.1 User guide . . . 41
6.2 Extendable framework . . . 42

**7 Experimental evaluation 45**
7.1 Generating process trees and clustered event logs . . . 45
7.2 Comparison of dissimilarity measures . . . 46
7.2.1 Experimental setup . . . 46
7.2.2 Results and discussion . . . 49
7.3 Comparison of configurable process discovery approaches . . . 52
7.3.1 Experimental setup . . . 52
7.3.2 Results and discussion . . . 53
7.4 Evaluation of the full approach . . . 58
7.4.1 Running example . . . 58
7.4.2 BPI challenge logs 2012 . . . 59

**8 Conclusion and future work 65**
8.1 Conclusion . . . 65
8.2 Future work . . . 67
8.2.1 Trace clustering . . . 67
8.2.2 Explaining clusters by data . . . 67
8.2.3 Configurable process discovery . . . 68
8.2.4 Additional experimental evaluation . . . 69

**Bibliography 71**

**Appendices 75**

**A Partially ordered alignment construction algorithm 76**

**B Additional experimental evaluation results 78**
B.1 Comparison of dissimilarity measures . . . 78
B.2 Evaluation of the full approach . . . 83
B.2.1 Running example . . . 83
B.2.2 BPI challenge logs 2012 . . . 83

# List of Figures

1.1 The configurable BPMN model of the process used as running example. The model describes the process of obtaining a drivers license. . . . . . . . . . . . . . . . . . . . . . . . 3
1.2 The configurable process tree of the BPMN model shown in Figure 1.1 with only the letter-codes of the activities instead of the full names. . . . . . . . . . . . . . . . . . . . . . 3
1.3 The configured variants of the configurable BPMN model shown in Figure 1.1 (a, b, c) and the configured variants of the configurable process tree shown in Figure 1.2 (d, e, f). The dashed boxes surround the configured nodes. . . . . . . . . . . . . . . . . . . . . . . . . 4

2.1 The process tree operators and their translations to BPMN. . . . . . . . . . . . . . . . . . . 9
2.2 Downgrade hierarchy of the process tree operators. . . . . . . . . . . . . . . . . . . . . . . . 9
2.3 A configurable process tree (a) and the configured process tree after applying the first configuration (b). These are respectively the configurable process tree of the running example (Figure 1.2) and variant one of the running example (Figure 1.3d). . . . . . . . . 10
2.4 Alignment for a process tree (which is the unconfigured variant of the running example, as shown in Figure 1.2) and trace: $\langle A, B, C, Z, E, C, F, G, I \rangle$. A green line indicates a synchronous move, an orange line indicates a move on model only and a missing line indicates a move on log only. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.5 The four competing quality dimensions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.6 Flower-model for activities: `A`, `B`, `C`, `D`, `E`, and `F`. This model has perfect replay fitness for an event log with only these activities. . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.7 Trace-model for an event log with traces: $\langle A, C, D, E, F, B \rangle$, $\langle A, C, D, F, E, B \rangle$ and $\langle A, E, B \rangle$. This model has perfect replay fitness and precision for the given event log. . . . . . . . . . 12

3.1 Example segregation of traces that form clusters. . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2 The Levenshtein distance internally aligns the two sequences. This figure shows this alignment for the two example traces. The Levenshtein distance for these two traces is 5. 17
3.3 Optimal alignment for a process tree (which is the unconfigured variant of the running example, as shown in Figure 1.2) and trace: $\langle A, B, C, Z, E, C, F, G, I \rangle$. A green line indicates a synchronous move, an orange line indicates a move on model only and a red line indicates a move on log only. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.4 Partially ordered alignment of the alignment of Figure 3.3. The color of a node indicates the type of move; white: synchronous move, gray: move on model only, and black: move on log only. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.5 The table shows the global alignment of the topological sorts for the POAs of $t_1 = \langle A, B, C, Z, E, C, F, G, I \rangle$ and $t_2 = \langle A, F, C, D, G, Z, I \rangle$. The top POA represents $t_2$ and the bottom POA represents $t_1$. Green dotted lines indicate a proper relation between nodes and orange dotted lines indicate a weak relation. . . . . . . . . . . . . . . . . . . . . 20
3.6 Two traces $t_1$ and $t_2$, with identical POAs but different observed execution orderings. This figure illustrates the parallel splits of the POA. It shows the successors and the parallel execution ordering score per split. The final parallel execution ordering score is 2. . . . . . 20
3.7 The table shows traces which represent executions of the running example. The tree represents a hierarchical clustering of these traces. The clusters with green borders represent the preferred clusters. The three preferred clusters in this hierarchy represent the three variants of the running example. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

4.1 The reduced annotated hierarchical clustering of the example given in Figure 3.7 using the data set of Table 4.1. . . . . . . . 26
4.2 The transformation from cluster splits (on the left) to training sets (on the right) to use as input for a classification algorithm. . . . . . . . 27
4.3 The internal probabilistic model (b) of a naive Bayesian classifier for the training set (a), and the resulting annotated hierarchical clustering (c). . . . . . . . 27
4.4 The decision tree (b) created by the C4.5 algorithm for the training set (a), and the resulting annotated hierarchical clustering (c). . . . . . . . 28
4.5 Situation before reduction (a). Followed by two intermediate stages (b, c) and the final reduced hierarchy based on the data annotations (d). The symbol ∅ denotes the empty expression. . . . . . . . 29

5.1 Table of traces clustered into three clusters (a), a reference process tree (b), and the resulting configurable process tree (c). . . . . . . . 32
5.2 Approach 1: Merge individually discovered process models. . . . . . . . 33
5.3 Approach 2: Merge similar discovered process models. . . . . . . . 33
5.4 Approach 3: First discover a single process model then discover configurations. . . . . . . . 34
5.5 Approach 4: Discover process model and configurations at the same time. . . . . . . . 34
5.6 The boxes around the process tree show the mapping between the moves of the alignments (as shown in the top for a trace $t_1$ and a trace $t_2$ in which all move on log only are already discarded) and the process tree for leaf C and all of its parents. . . . . . . . 36
5.7 Mapping of an alignment (a) onto a process tree with a loop construct (b) and how the final configurable tree would look (c). The ∨-node is unexpectedly downgraded to a →-operator. 38
5.8 The partitioned mapping for the loop construct (a) and the resulting configurable process tree (b). . . . . . . . 39

6.1 Interactive visualization of the plug-in for a simulated log of the running example. The overview consists of five panes: 'Settings', 'Hierarchy overview', 'Cluster inspector', 'Tree overview' and 'Tree inspector'. . . . . . . . 42

7.1 The F1 scores with 95% confidence intervals of the best and worst cut in the normal case. 50
7.2 The F1 scores with 95% confidence intervals of the best and worst cut in the loop iterations case. . . . . . . . 50
7.3 The F1 scores with 95% confidence intervals of the best and worst cut in the order of execution case. . . . . . . . 51
7.4 The F1 scores with 95% confidence intervals of the best and worst cut in the parallelism punishment case. . . . . . . . 52
7.5 Configurable process tree discovered by the ETMc algorithm (a) and the configurable process tree discovered by the VisFreq algorithm (b). . . . . . . . 54
7.6 Comparison of the different quality criteria calculated on the results of the VisFreq and ETMc approaches and quality criteria that were calculated on the reference configurable process tree. The error bars denote the 95% confidence intervals. . . . . . . . 57
7.7 Hierarchical clustering of the running example using the Syntactic reasoning approach. . . 58
7.8 Process tree discovered using the Inductive Miner for the initial filtered event log of the BPI challenge 2012. . . . . . . . 60
7.9 Resulting hierarchical clustering of the initial filtered event log of the BPI challenge 2012. 60
7.10 Classification of the BPI challenge 2012 traces. . . . . . . . 61
7.11 Process tree discovered using the Inductive Miner for the filtered event log of the BPI challenge 2012 which includes sending of offers and checks for fraud. . . . . . . . 62
7.12 Partial trace clustering of the filtered event log of the BPI challenge 2012 which includes sending of offers and checks for fraud. . . . . . . . 62

8.1 A process tree with a choice between five activities (a) and a process tree with a nested choice. Both process trees are annotated with callouts that indicate the number of traces associated to the node. . . . . . . . 68

B.1 The recall and precision scores with 95% confidence intervals of the best and worst cut in the normal case. . . . . 79
B.2 The recall and precision scores with 95% confidence intervals of the best and worst cut in the loop iterations case. . . . . 80
B.3 The recall and precision scores with 95% confidence intervals of the best and worst cut in the order of execution case. . . . . 81
B.4 The recall and precision scores with 95% confidence intervals of the best and worst cut in the parallelism punishment case. . . . . 82
B.5 Hierarchical clustering of the running example using the Syntactic reasoning approach. . . 83
B.6 Hierarchical clustering of the running example using the Levenshtein distance. . . . . 83
B.7 Hierarchical clustering of the running example using Set-Of-Activities. . . . . 84
B.8 Hierarchical clustering of the running example using Bag-Of-Activities. . . . . 84
B.9 Hierarchical clustering of the running example using 3-Grams. . . . . 84
B.10 The full hierarchical clustering of the filtered BPI challenge log of 2012 which includes sending of offers and checks for fraud. . . . . 84

# List of Tables

2.1 Two cases of the running examples with data attributes. . . . . 7
2.2 Events of the cases shown in Table 2.1. . . . . 8
2.3 Machine learning data set. Every row is an object in the machine learning task and represents a case of the running example. . . . . 13

4.1 Executions of the running example with data attributes. . . . . 25
4.2 The traces of Table 4.1 enriched with feature data attributes. . . . . 30

6.1 Overview of the possible extensions, their corresponding base class, and the annotation that should be added to enable the extension in the GUI. . . . . 43

7.1 Per case the probability of an operator or leaf node. Followed by the probability of an $\tau$-leaf and the probability of each operator type. Any leaf that is not a $\tau$ describes an actual activity. . . . . 47
7.2 Per case the tree size, number of children per operator node, whether duplicate activities are allowed, and the number of event classes. . . . . 47
7.3 Per case the allowed configuration types, the number of traces per experiment, the number of clusters per event, and the probability of noise. . . . . 47
7.4 Quality criteria of the two trees shown in Figure 7.5. The best scores among the two approaches are in bold. Fewer configuration options is considered to be better. For the other quality dimensions holds that a higher score is considered to be better. . . . . 55
7.5 Per dissimilarity measure the size of the hierarchical clustering in number of clusters and whether a variant of the running example can be distinguished in this hierarchical clustering. 59
7.6 Summarizes the leaf clusters of Figure 7.9. It shows the number of traces and the state of the traces per cluster. . . . . 60
7.7 Summarizes the clusters of Figure 7.12. Per cluster it shows the number of traces, the state of the traces, whether an offer was sent, and whether a fraud check was performed. . 63

# Chapter 1

# Introduction

The increased use of information systems causes more and more data to be recorded. Much of this data is often unstructured and very hard to reason about. One of the main challenges is to extract information from this raw data, such that we get value from it [2]. The field of data mining aims at extracting information from raw data. Often information systems execute processes and the recorded data describes events of this process. The field of *process mining* tries to extract information about the processes of an organization by mainly using the recorded events. The field of process mining is partitioned into three main activities: *discovery*, *conformance checking* and *enhancement*. These are respectively the tasks of discovering a new process model from the recorded events, checking how the recorded events conform to a process model or a given set of rules, and enhancing an existing process model by the use of the recorded events [2]. In this thesis we mainly focus on *process discovery*.

Best practices, legal obligations or enterprise systems cause organizations to organize their business processes in very similar ways [21]. An example of such organizations are municipalities. Municipalities often provide the same services but are bound by government regulations, causing them to execute the same processes but with slight deviations [12], e.g., the size of a municipality can influence the handling of building permits [11]. Another reason is that with the growth and increased use of systems, the development and maintenance cost greatly increases. The increased use of shared business process management infrastructures, Software-as-a-Service (SaaS) and Cloud Computing shows that companies are willing to share the development and maintenance cost [11]. Using such systems does however force them to use similar processes.

The CoSeLoG project[1] aims at creating a cloud infrastructure for municipalities. The CoSeLoG project is a collaboration between the TU/e, 2 IT companies and 10 Dutch municipalities. These municipalities wish to migrate their information systems to a cloud solution. By doing this they might save future development and maintenance costs. They however want to keep variability in their processes such that they can work according to their preferences. *Configurable process models* provide the means to deal with variability in processes. A configurable process model can combine many different variants of the same process. An organization is able to configure the process model in such way that it best suits their way of working.

In the field of process mining an execution of a process is called a *trace* (or case) and is a sequence of *events*. A collection of traces is called an *event log*. Event logs can be extracted from the existing information systems of the municipalities. To support a quick migration from the existing situation to a cloud solution, process discovery can be used to find a configurable process model that supports each of the municipalities. Existing techniques however discover a configurable process model with a configuration for each of the input event logs, which would mean that a configuration is discovered for every municipality. Such techniques discover configurations at an organizational level. In this thesis we propose a new approach that allows discovery at the behavioral level, i.e., we only consider the observed behavior in the traces and not their origin. Discovery at a behavioral level treats the collection of event logs, which we obtain from the municipalities, as a single event log. To discover a configurable process model we do however still need a segregation of the traces. To obtain this segregation we identify traces that are behaviorally similar, i.e., the same choices are made during process execution, and group them.

[1]The official website is: `http://www.win.tue.nl/coselog/wiki/`.

Next we provide insights to the end-user in why these groups are formed by the data of the traces and events in the group. By doing this we show how data implicates the observed behavior. It is now possible to automatically, or by the end-user of our full approach, make a selection of interesting groups for which a configurable process model is discovered.

For instance, municipalities with over 20,000 citizens might handle building permits different from municipalities with fewer than 20,000 citizens. With the usual way of configurable process discovery it is very difficult, or even impossible, to identify these kinds of rules. Discovery at the behavioral level does allow for this. Creating a cloud solution supporting the configurations at a behavioral level rather than at an organizational level allows for quick implementation of new municipalities and easier sharing of the processes. To include a new municipality in the cloud solution it is only required to fill in parameters (e.g., the number of citizens) in order to find an initial configuration.

In this thesis we propose a new approach that allows for the discovery of a configurable process model at a behavioral level with explanations based on data. In Section 1.1 we propose a running example which we use to explain the proposed concepts. Next in Section 1.2 we explain our research goal. Finally, we present an outline of the remainder of this thesis in Section 1.3.

## 1.1 Running example

To explain the proposed concepts in this thesis we use a *running example*. Figure 1.1 shows the configurable BPMN model of our running example and Figure 1.2 shows the same process model as a configurable process tree. For our running example we consider the process of a driving school. The way of payment at a driving schools often differs. Some driving schools require the student to buy a package of classes and a practical exam while others require payment before or after each individual class. A configurable process model can be used to represent these three variants of the process. A driving school can then configure the process model according to their preferred way of payment.

The process starts when a student applies to the driving school. At some schools the student should first buy a package of classes and a practical exam (*Buy Classes + Exam* (B)). Then driving classes start (*Driving Class* (C)). If the student has bought a package, they do not have to pay per class, while students that did not buy a package should either pay before or after the class (*Pay For Class* (D)), depending on the driving school. At some time during the classes the student should take and pass a theoretical exam (*Theoretical Exam* (F)). After the student is deemed good enough, and has taken their theoretical exam, he or she can take part in a practical exam (*Take Practical Exam* (G)). The student can fail this exam and start taking classes again (*Failed* (H)), fail the exam and stop trying (*Stop Trying* (J)), or the student passes the exam and receives a driving license (*Receive License* (I)).

The process models (shown in Figure 1.1 and Figure 1.2) contain callouts which describe the configuration options for each variant. To obtain an executable model one of these configurations should be applied. We now explain the three variants in more detail:

1. **Packages:** The student has to buy a package before starting driving classes. Figure 1.3a shows the configured BPMN model and Figure 1.3d shows the configured process tree. This variant is obtained in the BPMN model and process tree by hiding activity *Pay For Class* (D) and allowing execution of activity *Buy Classes + Exam* (B). In the BPMN model we should also block the branch where *Pay For Class* (D) is executed before *Driving Class* (C).

2. **After class payment:** The student should pay after each class. Figure 1.3b shows the configured BPMN model and Figure 1.3e shows the configured process tree. This variant is obtained in the BPMN model by hiding activity *Buy Classes + Exam* (B) and blocking the branch where *Pay For Class* (D) is executed before *Driving Class* (C). In the process tree we obtain this variant by hiding activity *Buy Classes + Exam* (B) and downgrading the $\wedge$-node to a $\rightarrow$-operator.

3. **Before class payment:** The student should pay before each class. Figure 1.3c shows the configured BPMN model and Figure 1.3f shows the configured process tree. This variant is configured in the BPMN model by hiding activity *Buy Classes + Exam* (B) and blocking the branch where *Driving Class* (C) is executed before *Pay For Class* (D). In the process tree we obtain this variant by hiding activity *Buy Classes + Exam* (B) and downgrading the $\wedge$-node to a $\leftarrow$-operator.

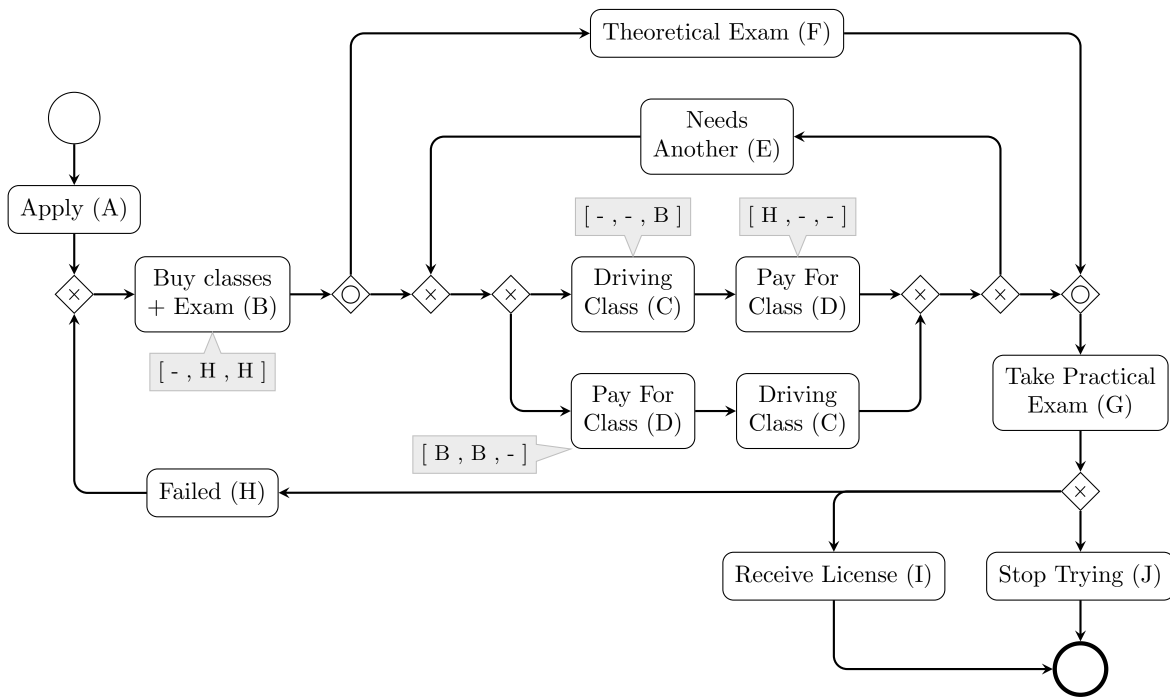



Figure 1.1: The configurable BPMN model of the process used as running example. The model describes the process of obtaining a drivers license.

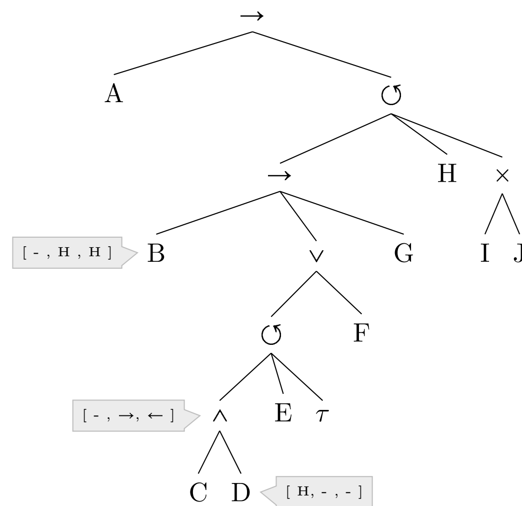
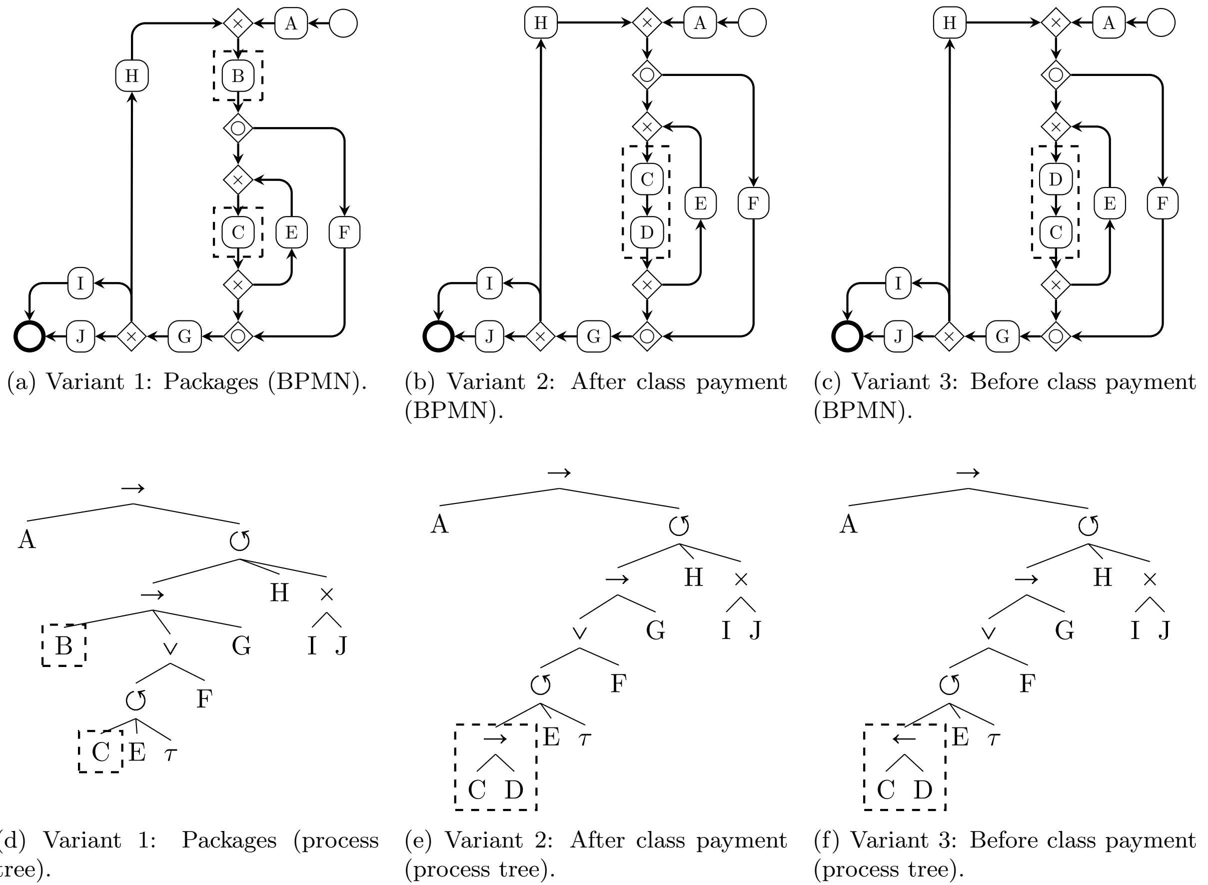



Figure 1.2: The configurable process tree of the BPMN model shown in Figure 1.1 with only the letter-codes of the activities instead of the full names.




(a) Variant 1: Packages (BPMN).

(b) Variant 2: After class payment (BPMN).

(c) Variant 3: Before class payment (BPMN).

(d) Variant 1: Packages (process tree).

(e) Variant 2: After class payment (process tree).

(f) Variant 3: Before class payment (process tree).

Figure 1.3: The configured variants of the configurable BPMN model shown in Figure 1.1 (a, b, c) and the configured variants of the configurable process tree shown in Figure 1.2 (d, e, f). The dashed boxes surround the configured nodes.

## 1.2 Research goal

The goal of our research is to identify groups of traces in an event log on a behavioral level. We wish to explain each of these groups with the data of the traces and events in the group. By doing this we can observe how data implicates the observed behavior. For each of the groups we wish to find a configuration for a given process tree. Because we already explained the groups of traces by the data we can now also observe how data implicates a configuration on the process tree. In order to reach our goal we answer the following research questions:

1. How can we identify groups of behaviorally similar traces in an event log?

2. How can we explain these groups based on data?

3. How can we create a configuration for a given process tree and a group of traces?

4. How can we select a good configurable process tree for a selection of groups of traces?

Consider for instance our running example. Assume we have multiple driving schools which decide to implement a cloud solution. For each of these driving schools we can extract an event log. With our full approach it should be possible to identify different variants of the process. For the running example this would be the three variants of the way of payment. A data attribute describing the way of payment could explain each of these variants. We can discover a configurable process model for these variants. A new cloud solution could now be implemented that supports this configurable process model. When a new driving school wishes to use this cloud solution they only need to fill in which way of payment they wish in order to obtain an appropriate configuration.

## 1.3 Outline

In the previous sections we explained the field of research, introduced our research problem, proposed a running example which is used to explain the concepts of this thesis, and finally explained our research goal. In this section we present an outline of the remainder of this thesis.

In Chapter 2 we explain preliminary knowledge used in the remainder of this thesis. The following concepts are explained: event logs, (configurable) process trees, alignments, process tree quality characteristics, the Evolutionary Tree Miner framework, and finally clustering and classification.

In Chapters 3-5, we explain the theoretical foundation of our full approach and answer our research questions. Our full approach consists of the following steps:

1. **Trace clustering:** In Chapter 3 we answer the first research question. We explain the concept of trace clustering and discuss related work. We propose a new approach towards trace clustering which should improve existing methods by adding model-awareness.

2. **Explaining clusters by data:** In Chapter 4 we answer the second research question. We relate our problem to decision mining and show how a trace clustering can be annotated with data from an event log.

3. **Configurable process discovery:** In Chapter 5 we answer the third and fourth research question. We discuss existing approaches towards configurable process discovery and propose a new method. We also propose a method to evaluate which configurable process model is the best.

Chapter 6 provides a user guide and explains how our plug-in can be extended. In Chapter 7 we present an experimental evaluation in which we compare our proposed approaches with existing approaches and evaluate our full approach. Finally, we conclude this thesis and discuss possible future work in Chapter 8.

# Chapter 2

# Preliminaries

In this chapter we explain preliminary knowledge used in the remainder of this thesis. First we explain event logs in Section 2.1. Second we explain process trees in Section 2.2, then continue on how these process trees can be configured in Section 2.3. Section 2.4 explains alignments of modeled and observed behavior. Section 2.5 explains how the quality of a process tree can be measured. In Section 2.6 we explain the Evolutionary Tree Miner. Finally, we explain the notions clustering and classification in Section 2.7.

## 2.1 Event logs

The *event log* is a key concept in the field of process mining [2]. An event log consists of a set of *traces* and a trace is a sequence of *events* (also called *activities*). Every event log, trace or event can contain data attributes. The data attributes of an event must at least describe the executed type of activity but may also include other information, e.g., the resource or timing information. Every type of activity is also called an *event class*.

Table 2.1 shows two traces of the running example. For every trace is registered: the payment method, the student, the instructor, the examiner, how many classes were taken, and how many times the student failed. Table 2.2 shows the events of these traces. The data of these tables combined forms an event log.

| *Case Id* | *Attributes* | | | | | | |
|---|---|---|---|---|---|---|---|
| | *Payment* | *Student* | *Instructor* | *Examiner* | *Classes#* | *Failures#* | ... |
| 221 | Package | Arthur | John | William | 3 | 0 | ... |
| 285 | AfterClass | Sara | John | Janette | 2 | 1 | ... |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

Table 2.1: Two cases of the running examples with data attributes.

| *Case Id* | *Event Id* | *Attributes* | | | |
|---|---|---|---|---|---|
| | | *Timestamp* | *Activity* | *Resource* | ... |
| 221 | 41251 | 2014-05-02 13:05 | Apply (A) | Jane | ... |
| | 41252 | 2014-05-02 13:12 | Buy classes (B) + Exam | Jane | ... |
| | 41257 | 2014-05-02 13:31 | Driving Class (C) | John | ... |
| | 41258 | 2014-05-02 14:32 | Needs another (E) | John | ... |
| | 41261 | 2014-05-05 15:14 | Driving Class (C) | John | ... |
| | 41262 | 2014-05-05 16:17 | Needs another (E) | John | ... |
| | 41269 | 2014-05-07 13:01 | Theoretical Exam (F) | John | ... |
| | 41271 | 2014-05-10 14:59 | Driving Class (C) | John | ... |
| | 41280 | 2014-05-15 15:57 | Take Practical Exam (G) | William | ... |
| | 41295 | 2014-05-21 14:07 | Receive License (I) | Jane | ... |
| 285 | 53151 | 2014-06-10 09:02 | Apply (A) | Jane | ... |
| | 53155 | 2014-06-10 09:30 | Driving Class (C) | John | ... |
| | 53156 | 2014-06-10 10:26 | Pay For Class (D) | John | ... |
| | 53157 | 2014-06-10 10:31 | Needs another (E) | John | ... |
| | 53162 | 2014-06-17 14:15 | Driving Class (C) | John | ... |
| | 53163 | 2014-06-17 15:16 | Pay For Class (D) | John | ... |
| | 53164 | 2014-06-23 12:59 | Theoretical Exam (F) | John | ... |
| | 53170 | 2014-06-30 16:02 | Take Practical Exam (G) | Janette | ... |
| | 53180 | 2014-07-07 14:12 | Stop Trying (J) | Jane | ... |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

Table 2.2: Events of the cases shown in Table 2.1.

## 2.2 Process trees

The *process tree* [10] is one of many process modeling notations that can be used to represent process models. Other examples of process modeling notations are: Petri Nets [2], Business Process Model and Notation (BPMN) [32], Event Driven Process Chains (EPC) [1], and many more. However for these languages only a small fraction of all possible models is sound, i.e., they do not contain deadlocks or other problems [4]. Process trees do not suffer from this issue. Because of the structure of process trees, which only allows for block-structured models, every possible process tree is sound.

A process tree is a tree that consists of *operator-nodes* and *activity-nodes*. Any leaf-node of the tree is an activity-node and all other nodes are operator-nodes. Operator-nodes specify a relation between their children. There are six types of operators: sequence ($\rightarrow$), parallel execution ($\wedge$), exclusive choice ($\times$), choice ($\vee$), repeated execution ($\circlearrowleft$), and the reverse sequence ($\leftarrow$) which is simply the $\rightarrow$ reversed. Figure 2.1 shows the possible operators that process trees can be composed of and their translations to BPMN. All operators except the $\circlearrowleft$-operator can have one or more children. The $\circlearrowleft$-operator must always have three children, the first specifying the *do* part, the second specifying the *redo* part and the third child specifying the *exit* part of the loop. The activity-nodes of the process tree can also be labeled with a $\tau$ indicating an unobservable activity.

Currently there are two algorithms to discover a process tree from an event log, i.e., the ETMd and the Inductive Miner [28]. The ETMd is an evolutionary algorithm and is further discussed in Section 2.6. The Inductive Miner is a structured approach towards process tree discovery.

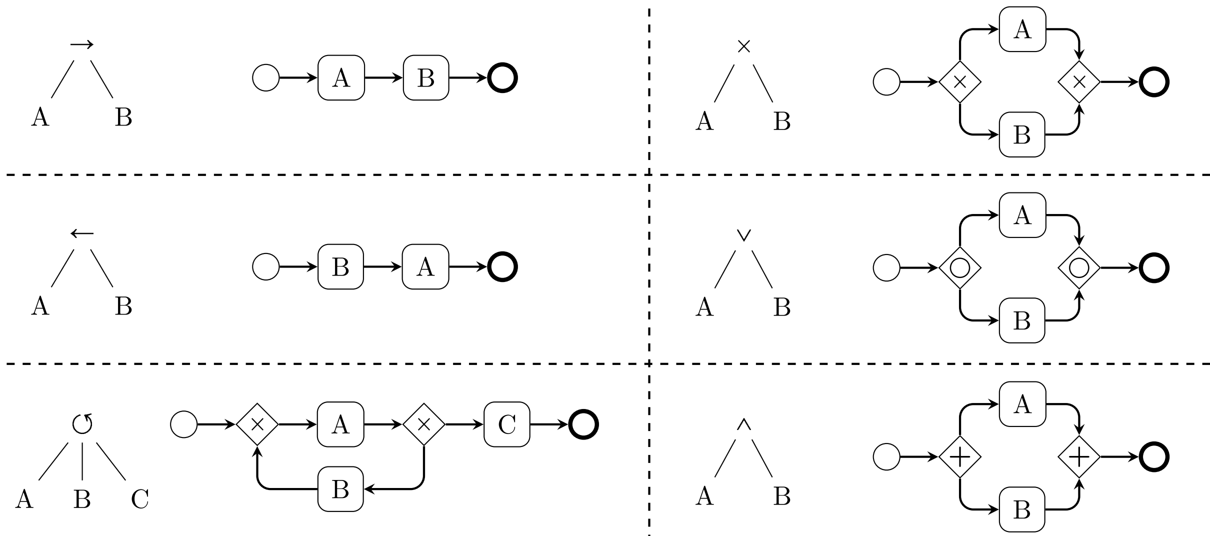



Figure 2.1: The process tree operators and their translations to BPMN.

## 2.3 Configurable process trees

A *configurable process model* describes a family of model variants. The *configurable process tree* [12] is one of many configurable process modeling notations [21, 34, 36]. In most of these notations the basic configuration options are: *allow*, *hide* and *block*. Allow meaning that the node is allowed to be executed, hide meaning that it can be skipped over, and block that the node cannot be executed (and also not be skipped over). Nodes in the process model can be annotated with these options, this results in a configurable process model. A collection of configuration options forms a configuration. In order to obtain an executable model a configuration should be applied on the model.

Configurable process trees support these three configuration options for any node in the tree. However if all children of a node are blocked, or a single child of an $\wedge$, $\rightarrow$, $\leftarrow$-node, or the do or exit-child child of a ↺-node is blocked, it indicates that the node itself is blocked as well. Next to the allow, hide and block options, configurable process trees support the concept of *operator downgrading*. An operator can only be downgraded to a more restrictive operator. Figure 2.2 shows the downgrade hierarchy, in this hierarchy the operators above are less restrictive than the ones below. Process trees can only be restricted using configuration options. There do exist configurable process modeling notations that support extension of a process model [22]. These are however not defined for process trees and therefore not considered in this thesis.

In the remainder of this thesis we visualize configuration options as gray callouts. For example, if a callout contains: `[ -, H, B, → ]`, it indicates that for the first configuration the node should be allowed, for the second configuration the node should be hidden, for the third configuration the node should be blocked, and for the fourth configuration the node should be downgraded to a $\rightarrow$-operator.

Figure 2.3a shows the configurable process tree of the running example. The first configuration consists of a hide option on activity `D`. After applying the first configuration we obtain the process tree as shown in Figure 2.3b, which is variant one of the running example.

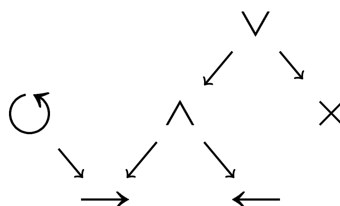
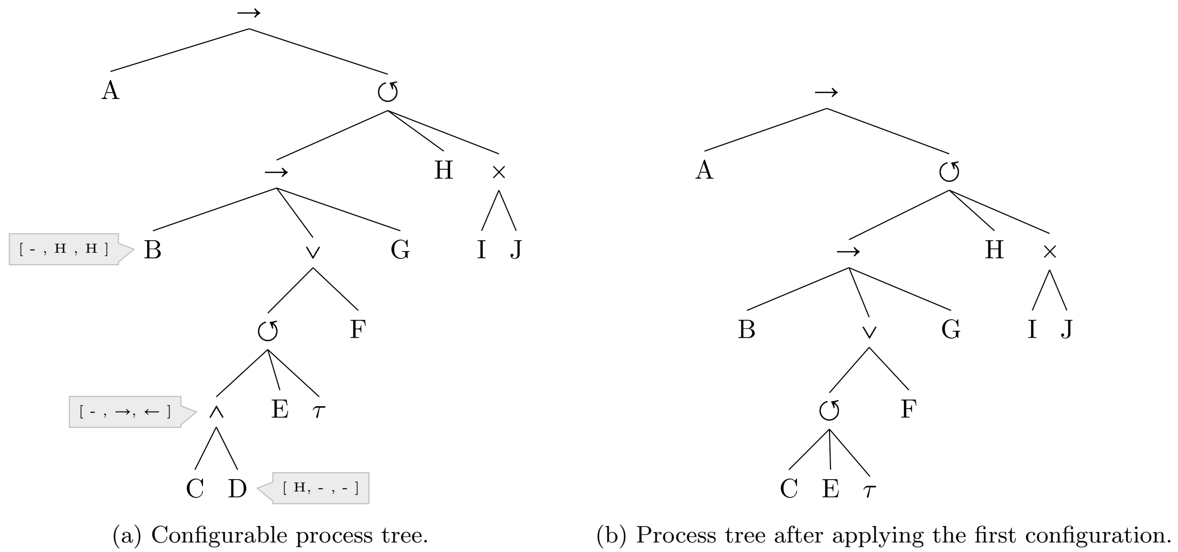


Figure 2.2: Downgrade hierarchy of the process tree operators.




(a) Configurable process tree.

(b) Process tree after applying the first configuration.

Figure 2.3: A configurable process tree (a) and the configured process tree after applying the first configuration (b). These are respectively the configurable process tree of the running example (Figure 1.2) and variant one of the running example (Figure 1.3d).

## 2.4 Alignments

To relate modeled (process tree) and observed behavior (event log), the notion of *alignments* is introduced [3, 6]. An alignment is basically a mapping of as many as possible activities observed in a trace to activities in an execution of the process model. Every alignment consists of a log-execution and a model-execution, respectively denoted by $l$ and $\sigma$. Figure 2.4 shows the alignment of trace: $\langle A, B, C, Z, E, C, F, G, I \rangle$ and the process tree of the running example without configuration options (Figure 1.2). The columns of the table represent the *moves* of the alignment. Every move is a mapping between the log-execution and the model-execution. If a log-execution or model-execution misses this is denoted by a $\gg$-symbol. Three types of moves are distinguished:

1. **Synchronous move:** Is a move that has both an execution in the log and in the model. The green lines in Figure 2.4 indicate the relation between the *synchronous moves* and the model.

2. **Move on model only:** Is a move that has an execution in the model but no corresponding execution in the log. The orange lines in Figure 2.4 indicate the relation between the *model moves* and the model.

3. **Move on log only:** Is a move that has an execution in the log but without a corresponding model execution. The moves in Figure 2.4 without a line to the model are the *log moves*.

Alignments are often used to calculate quality characteristics of a process model like replay fitness or precision. However alignments are also extremely useful in identifying problems of a process model. Every move on log only and move on model only indicates a point where the observed behavior (event log) and the modeled behavior (process tree) did not align.

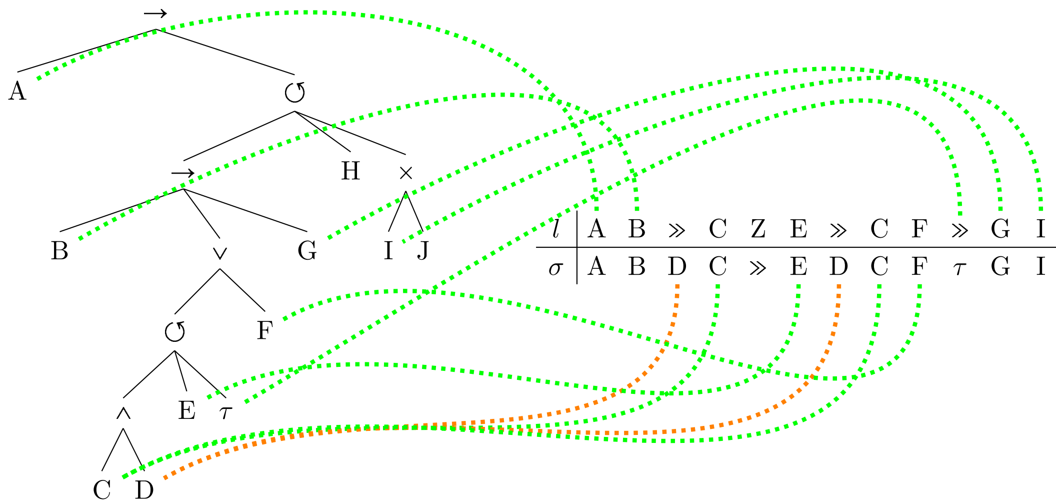



Figure 2.4: Alignment for a process tree (which is the unconfigured variant of the running example, as shown in Figure 1.2) and trace: $\langle A, B, C, Z, E, C, F, G, I\rangle$. A green line indicates a synchronous move, an orange line indicates a move on model only and a missing line indicates a move on log only.

## 2.5 Measuring the quality of a process tree

To compare and discuss process discovery techniques often four different *quality dimensions* are used. These quality dimensions compete with each other, improving in one dimension might make another dimension worse. Figure 2.5 shows these quality dimensions. These quality dimensions are defined as follows [2, 10]:

- **Replay fitness:** Describes how well the process model can execute the behavior as observed in an event log. Models with perfect replay fitness should be able to execute all behavior in the event log. A flower-model is a model that at any point in time allows the execution of any activity. An example of a flower model with six activities is shown in Figure 2.6. The flower model containing all activities can always replay all traces of an event log, meaning that it has perfect replay fitness.
- **Precision:** Perfect replay fitness can easily be achieved by creating a flower-model that consists of all activities. Such a model does however underfit an event log since it allows for too much behavior. Precision describes how well the process model fits an event log. For a flower-model (Figure 2.6) the precision would be very low. Both perfect replay fitness and precision can be achieved by creating a trace-model. A trace-model starts with a choice between sequences of activities which

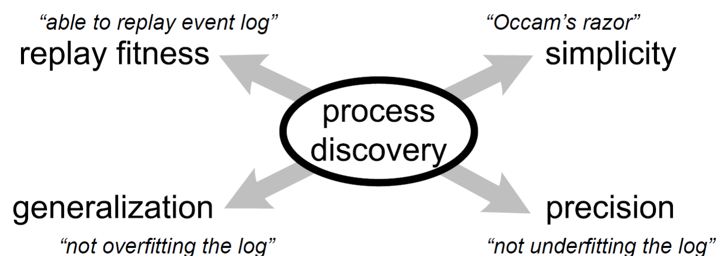



Figure 2.5: The four competing quality dimensions [2].

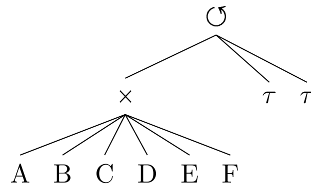



Figure 2.6: Flower-model for activities: A, B, C, D, E, and F. This model has perfect replay fitness for an event log with only these activities.

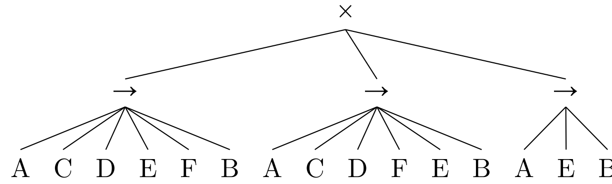



Figure 2.7: Trace-model for an event log with traces: $\langle A, C, D, E, F, B\rangle$, $\langle A, C, D, F, E, B\rangle$ and $\langle A, E, B\rangle$. This model has perfect replay fitness and precision for the given event log.

each describe a trace of the event log. An example of a trace-model for three traces is shown in Figure 2.7. To penalize such models we need the notion of generalization and simplicity.

- **Generalization:** Not all behavior might have been observed in an event log. Generalization describes the relation between the overfitting of a model with respect to an event log. A model that does not generalize is deemed to be overfitting. A trace-model (Figure 2.7) exactly describes the event log and is therefore overfitting it. Such models have a low generalization score.
- **Simplicity:** Captures the principle of Occam's razor. The size of a process model is the main indicator of simplicity, meaning that the smallest process model is considered to be the best model in this quality dimension. A trace-model (Figure 2.7) contains a sequence for every different trace, this causes the model to become huge for many different traces, which results in a low simplicity score.

Finding a process model with perfect scores for a subset of these four dimensions is often easy. For example, a trace-model (shown in Figure 2.7) always achieves perfect replay fitness and precision. However finding a process model with a perfect score on all four dimensions is often impossible. Therefore process discovery techniques have to balance the different quality dimensions to find a model that is most preferable.

## 2.6 Evolutionary Tree Miner

The *Evolutionary Tree Miner* [9, 12], or *ETM* for short, is an evolutionary process mining framework. Evolutionary algorithms work with the notion of generations. It starts by generating an initial population of candidate solutions. Next are multiple generations wherein changes (or mutations) are made to the candidate solutions by mutators. After every generation the best (according to the specified quality characteristics) candidate solutions are kept. Multiple algorithms were implemented in the ETM framework. In this thesis we use the following algorithms:

**ETMd [9]:** Discovers process trees from a single event log. The result of the ETMd algorithm is a single process tree, or a population of process trees that specialize in different quality characteristics. Most process discovery techniques focus on a subset of the quality dimensions, e.g., replay fitness and precision. The ETMd is able to balance all the quality dimensions according to user preferences.

**ETMc [12]:** Discovers configurable process trees from multiple event logs. It can discover a configurable process model from scratch, wherein aspects of the ETMd algorithm are used, or it can discover configurations for an existing process tree.

## 2.7 Data mining: Clustering and classification

Data mining is the practice of finding relationships in data or to summarize large sets of data. Two major areas in this field are: *clustering* and *classification*, also called unsupervised and supervised machine learning [2]. Classification assumes a labeled data set while clustering assumes an unlabeled data set. A data set consists of objects and every object has data attributes associated to it. A data set is considered to be labeled if its objects are assigned to groups. The data set of a machine learning task can be visualized as a table, Table 2.3 shows an example data set. Every row in this table describes an object and its data attributes. The first column in this data set is the label. Classification and clustering work as follows:

**Classification:** In classification, the algorithm tries to find an optimal explanation for the given labels without overfitting it. This explanation is stored as a classifier. This classifier can be used to assign a label to newly arriving unlabeled objects. For our example training data in Table 2.3, the classification algorithm might explain label $C_1$ by $Payment = Package$, $C_2$ by $Payment = BeforeClass$ and $C_3$ by $Payment = AfterClass$. This means that any newly arriving unlabeled object with $Payment = BeforeClass$ is labeled as $C_2$.

**Clustering:** In clustering the labels are unknown and the algorithm tries to find the correct labels. Clustering algorithms either result in a flat partitioning, or in a hierarchical partitioning of the objects wherein in each split the most dissimilar objects are separated. Most clustering algorithms require a dissimilarity measure which describes how dissimilar two objects are. Often such a dissimilarity measure is implemented using a distance function, e.g., for dimensional data: Euclidean distance, Manhattan distance, Hamming distance, and for sequence data: Levenshtein distance or Damerau-Levenshtein distance. If we define a dissimilarity measure which heavily punishes a different value in the *Payment* column, a flat partitioning algorithm should return the labels as shown in the first column of Table 2.3.

| *Label* | *Payment* | *Classes#* | *Failures#* |
|---|---|---|---|
| $C_1$ | Package | 20 | 0 |
| $C_1$ | Package | 27 | 0 |
| $C_1$ | Package | 38 | 1 |
| $C_2$ | BeforeClass | 18 | 0 |
| $C_2$ | BeforeClass | 25 | 0 |
| $C_2$ | BeforeClass | 48 | 2 |
| $C_3$ | AfterClass | 21 | 0 |
| $C_3$ | AfterClass | 35 | 1 |
| $C_3$ | AfterClass | 29 | 0 |

Table 2.3: Machine learning data set. Every row is an object in the machine learning task and represents a case of the running example.

# Chapter 3

# Trace clustering

In this chapter we answer the first research question: **How can we identify groups of behaviorally similar traces in an event log?** This problem is directly related to *trace clustering*. Trace clustering is the task of finding homogeneous groups of traces. Trace clustering is not different from regular clustering aside from the input: an event log which is a collection of traces. Every trace in the event log is an object in the clustering task. One of the first to discuss the significance of trace clustering in process discovery was Medeiros et al. [30]. They mention that process discovery algorithms often work very well on structured processes with little noise. This is however typically not the case. It is very difficult to determine the scope of the process and there is often all kinds of noise. Because of this, process discovery algorithms often produce spaghetti process models. Trace clustering is believed to solve this problem. By clustering an event log we can obtain more homogeneous groups of traces. This leads to more comprehensible process models during discovery. In our approach we see a similar problem. We want to find groups of homogeneous traces such that we can adequately create process model configurations for the groups.

Take our running example, assume we can position the traces of an event log in a two-dimensional space, one might get a result like Figure 3.1. After performing trace clustering we want to find the three clusters as indicated by the borders around the objects (in red, blue and green).

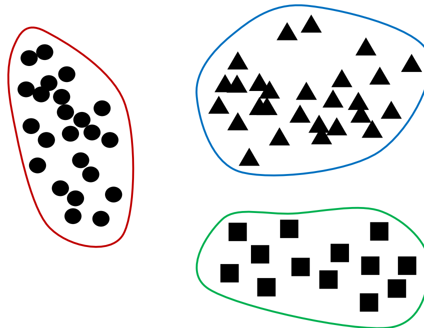


Figure 3.1: Example segregation of traces that form clusters.

Most clustering algorithms use a *dissimilarity measure*. The dissimilarity measure is an important factor in identifying objects that should be grouped together into clusters. Many approaches exist towards defining such a dissimilarity measure for trace clustering but none of these add the context of a process model, while this can greatly improve the clustering performance. Therefore we propose a new approach, the *Syntactic reasoning approach* which does include model-awareness.

Section 3.1 discusses existing approaches of defining a dissimilarity measure and their problems. In Section 3.2 we explain why we include process models in our approach towards trace clustering. Then in Section 3.3 we propose our dissimilarity measure, the Syntactic reasoning approach. Section 3.4 explains how we performed the trace clustering. Finally, we conclude this chapter in Section 3.5.

## 3.1 Related work

Most clustering algorithms use some kind of *dissimilarity measure*. Because of this we need to define a dissimilarity measure between traces. Since we want to cluster traces that are behaviorally similar, we should define this measure on behavior. Dissimilarity is often described as a distance between traces and the dissimilarity measure is therefore often described using a *distance function*. Existing work [5, 23, 24, 30, 37] suggests many different approaches towards defining such a distance function. In this we see a categorization between *feature-set* and *syntactical* approaches. To explain the existing approaches and their problems we use the following two traces: $t_1 = \langle A, B, C, E, C, F, G, I\rangle$ and $t_2 = \langle A, D, C, F, E, D, C, G, J\rangle$, which are executions of respectively model variants one and two of the running example presented in Section 1.1.

### 3.1.1 Feature-set approaches

Feature-set approaches define a vector of features for a trace. The distance between two of such vectors can be calculated using a distance function, e.g., the Euclidean, Hamming or Jaccard distance [37]. The most basic feature-set approaches are Set-Of-Activities and Bag-Of-Activities [24, 37]. In these approaches the occurrence of an activity is a feature of the feature-set. The Set-Of-Activities approach only registers whether an activity occurred or not, while the Bag-Of-Activities approach actually counts the number of occurrences. In the Bag-Of-Activities approach our example traces would result in the vectors: $v_{t1} = (1, 1, 2, 0, 1, 1, 1, 1, 0)$ and $v_{t_2} = (1, 0, 2, 2, 1, 1, 1, 0, 1)$, where the elements respectively denote the occurrence of activity: `A`, `B`, `C`, `D`, `E`, `F`, `G`, `I`, and `J`. The Set-Of-Activities and Bag-Of-Activities approaches both fully ignore the order of execution aspect.

Another basic approach is k-grams [24]. In k-grams the occurrence of a subsequence of the trace of length $k$ is a feature, e.g., in our example 3-grams features would be: $\langle A, B, C\rangle$ and $\langle A, D, C\rangle$. This approach is a generalization of the Bag-Of-Activities and behaves identical for $k = 1$. K-grams partially solves the order of execution problem but explodes the number of features for longer traces.

Jagadeesh Chandra Bose [23, p.81] proposes feature-sets that are context-aware. The idea of their approach is to find recurring patterns in all the traces of an event log, i.e., their common subsequence of activities. Unlike k-grams its features can consist of variable lengths. The so called 'Process-Centric' feature-sets can be classified in two categories: (i) sequence features and (ii) alphabet features. Their sequence features enforce constraints on the order of the activities while the alphabet features, derived from the sequence features, relax the ordering of these sequence features. Choosing the appropriate feature-sets is still largely dependent on the context of the analysis. Depending on which constructs we want to cluster we should pick the appropriate feature-set.

Although the approach by Jagadeesh Chandra Bose [23, p.81] tries to solve problems like loops and ordering of execution, we are still very much dependent on the context of the analysis. Loops might be identified where they do not exist. Different execution orderings of activities might be observed and allowed by the feature-set while the process model behind it does not allow for the behavior.

### 3.1.2 Syntactical approaches

Feature-set approaches try to abstract from the traces by finding features that identify the trace. Syntactical approaches define the distance between traces in terms of error-transformations, i.e., modifications required to change one trace into the other. A trace is a sequence of activities and a fundamental way to calculate the dissimilarity of two sequences is the Levenshtein distance (also referred to as the edit distance). The Levenshtein distance counts the number of insertions, deletions and substitutions required to change one sequence into another.

A global alignment of our example traces: $t_1$ and $t_2$ is shown in Figure 3.2. The Levenshtein distance of these traces is 5. We quickly observe that the execution of `F` in a different position in the trace causes an additional two edits (one insert and one delete). But when also considering the process models of the running example we observe that the key difference is that in variant one we execute activity `B` and in variant two we do not. Therefore in order to find the appropriate trace clustering it is not preferable that the difference in the execution of `F` accounts for a large portion of the distance.

The Damerau-Levenshtein distance is an extension of Levenshtein distance which adds the transposition error-transformation. This error-transformation allows transposition of two adjacent activities. For

| $t_1$ | A | B | C | - | E | - | C | F | G | I |
|---|---|---|---|---|---|---|---|---|---|---|
| $t_2$ | A | D | C | F | E | D | C | - | G | J |

Figure 3.2: The Levenshtein distance internally aligns the two sequences. This figure shows this alignment for the two example traces. The Levenshtein distance for these two traces is 5.

our example the Damerau-Levenshtein distance yields an edit distance of 5. The `F` activities are not transposed since they are too far apart.

Another syntactical approach is the Generic Edit Distance [23, p.86] which is a generalization of the Levenshtein distance and allows for a custom cost function. For our example we could now set the costs of an insertion or deletion of `F` to 0. This improves the final edit distance but this is however a very context-specific task.

## 3.2 Adding model-awareness

Some of the approaches discussed in Section 3.1 try to add context-awareness but this is only considered among traces in the event log. What if we already know a single process model or multiple process models to reason about. These process models could provide important context-information about traces. By adding a process model, we can calculate an alignment for every trace with respect to the process model. This alignment tries to match every activity in the trace to a transition in the model (or a node in case of a process tree). Using this alignment between the trace and the model we can reason about how activities of the trace are related to each other. Adding model-awareness can solve some of the previously mentioned issues. By adding model-awareness we can accurately:

1. Match activities that were executed in a different order based on their relation to the model and each other. The Levenshtein distance punishes a mismatch in ordering with at least two error-transformations. By identifying these kind of mismatches we can assign a more appropriate cost to such a mismatch.

2. Identify loops in the traces and deal with different number of iterations appropriately. In existing approaches an additional loop iteration can cause for a big difference in the distance. For example, if one trace executes a loop 5 times and the other 6 times and every iteration adds 10 activities. With no other differences the Levenshtein distance already counts 10 edits. We think that this $6^{th}$ iteration should however have a very small impact on the final edit distance.

## 3.3 The Syntactic reasoning approach

Existing feature-set approaches often suffer from issues such as: (i) trouble capturing the ordering of the execution, and (ii) an exploding number of features with the growing size of a trace. Existing syntactical approaches suffer from issues such as: (i) no real loop identification, (ii) and punishment of mismatches for a different ordering of activities even while the corresponding process model does not restrict the ordering of these activities. By adding model-awareness we can solve the common issues of syntactical approaches. Therefore we propose a new syntactical approach that overcomes the typical issues of existing approaches by adding model-awareness.

To overcome the issues we first introduce partially ordered alignments and how these should be constructed in Section 3.3.1. Then in Section 3.3.2 we propose a method to relate two partially ordered alignments to each other such that we can compare them to each other. To penalize mismatches in parallel behavior we propose a method of calculating a score that represents this mismatch in Section 3.3.3. Then in Section 3.3.4 we define the distance function for two traces and a process model. Finally in Section 3.3.5 we allow for multiple process models and explain how the scores of the distance function can be aggregated to a single distance function which represents the final dissimilarity measure of the Syntactic reasoning approach.

Our approach is implemented for process trees but can be generalized to any process modeling notation if partially ordered alignments can be constructed, and the *do* and *redo* activities of loops can be identified.

### 3.3.1 Partially ordered alignments

Assume we are given a process tree (which is the running example as shown in Figure 1.2 without configuration options), and a trace: $\langle A, B, C, Z, E, C, F, G, I \rangle$, we can now calculate an alignment. The resulting alignment is shown in Figure 3.3, a green line indicates a synchronous move, an orange line indicates a move on model only and a red line indicates a move on log only.

Using this alignment and the given process tree we construct a *Partially Ordered Alignment (POA)*. We use Figure 3.3 to explain our construction algorithm. The first step of the construction algorithm is to determine the parent node(s) for every move in the alignment. For a synchronous move or a move on model only this is obtained by a traversal of the process tree from the child to the root. For example, for the synchronous move of activity `C` the parent nodes are: $<\rightarrow_1, \circlearrowleft_1, \rightarrow_2, \vee, \circlearrowleft_2, \wedge>$.

Determining the parent node(s) for a move on log only is not trivial. For a move on log only we try to insert it under the parent of the first non-log move that occurred before it or the first non-log move that occurred after it. We never insert the move on log only under a $\times$-operator because if the parent of our neighbor is a $\times$-node it means that this $\times$-node is already executed, in this case we take the first non-$\times$ parent. We prefer insertion under a $\wedge$-operator or $\vee$-operator, as these operators do not enforce an order restriction in the resulting POA and therefore are not an as strong assumption as an insertion under e.g., a $\rightarrow$-operator would be. In our example we have one move on log only which executes activity `Z`. We can either insert it under the parent of activity `E` or the parent of activity `C`. The parent of activity `C` is a $\wedge$-operator while that of `E` is a $\circlearrowleft$-operator, therefore we prefer insertion under the parent of activity `C`. The red line in Figure 3.3 illustrates this insertion. The parents nodes for this move on log only become: $<\rightarrow_1, \circlearrowleft_1, \rightarrow_2, \vee, \circlearrowleft_2, \wedge>$.

The next step is to walk backwards over the alignment moves and compare each move to the moves that were after it (again starting with the last move of the alignment). For every succeeding move we retrieve the common parent. If this common parent does not restrict order (i.e., $\wedge$ or $\vee$-operator), we are done with this move. Else we create a reference to this successor and remove all successors that have now become redundant (the new successor might already refer to some of the old successors and thus these old successors have become redundant). After having obtained the successors for every move, we have obtained the final POA. The pseudo-code of this algorithm is included in Appendix A.

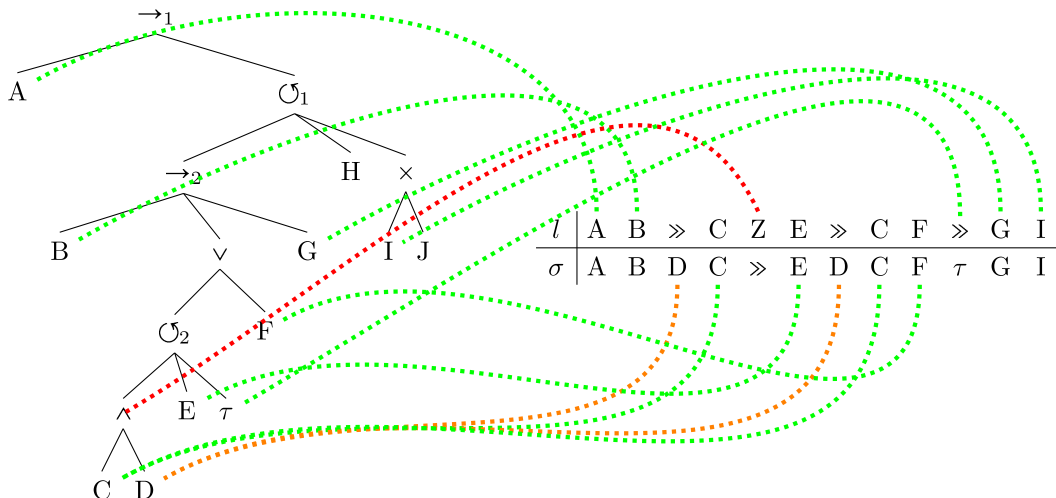



Figure 3.3: Optimal alignment for a process tree (which is the unconfigured variant of the running example, as shown in Figure 1.2) and trace: $\langle A, B, C, Z, E, C, F, G, I \rangle$. A green line indicates a synchronous move, an orange line indicates a move on model only and a red line indicates a move on log only.

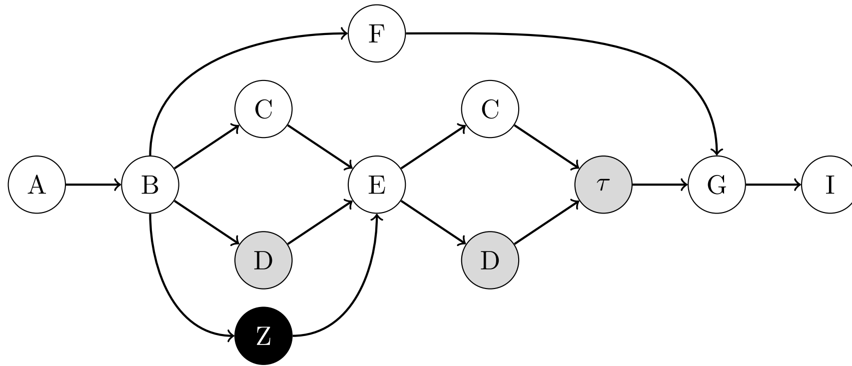



Figure 3.4: Partially ordered alignment of the alignment of Figure 3.3. The color of a node indicates the type of move; white: synchronous move, gray: move on model only, and black: move on log only.

For example, take the synchronous move of activity `F` and the synchronous move of activity I in Figure 3.3. For activity `F` we have the parent nodes: $<\rightarrow_1, \circlearrowleft_1, \rightarrow_2, \vee>$ and for activity `I` we have the parent nodes: $<\rightarrow_1, \circlearrowleft_1, \times>$, their common parent is the node: $\circlearrowleft_1$, this is not a $\wedge$-operator or $\vee$-operator, therefore we create a relation between F and I. Next we compare `F` with the synchronous move of activity `G`. Their first common parent is $\rightarrow_2$, therefore we create a relation between activity `F` and activity `G`. But since `G` already has a relation to `I` we now remove the relation from `F` to `I`. We continue in this fashion for all moves until we have constructed the full POA.

Any POA can be represented by a directed acyclic graph. Figure 3.4 shows the result of our algorithm on the example alignment as shown in Figure 3.3. We could now use a graph edit distance to calculate an edit distance between this POA and the POA for another trace. This solves the issue of punishment of parallel behavior. Only then mismatches in parallel behavior are not punished at all and the issues with loops remain.

### 3.3.2 Relating partially ordered alignments

To reason about a dissimilarity of POAs we need to relate one POA to the other. To illustrate how we propose to relate POAs, we use the trace from the previous section: $t_1 = \langle A, B, C, Z, E, C, F, G, I \rangle$ and introduce a new trace: $t_2 = \langle A, F, C, D, G, Z, I \rangle$. Figure 3.5 shows the POAs of these traces (constructed for the process tree shown in Figure 3.3) and relates them to each other. The dotted green lines show *proper relations* meaning that both nodes have the same activity type and move type. Orange dotted lines show *weak relations* meaning that both nodes have the same activity type but different move types. All nodes that are not connected do not have a node to which they can be related in the other POA. Surprising might be that the move on log only of activity `Z` is not connected to its partner in the other POA, this is caused by their different contexts. Relating the POAs on `Z` would result in a misalignment on: `C`, `D`, `G`, and $\tau$, which is obviously a worse relation.

In order to relate the POAs we first create a flat presentation of both POAs using an modified version of the topological sort algorithm that was first introduced by Kahn [26]. This algorithm starts with an empty list $L$, which eventually contains the sorted nodes, and a set $S$, which contains all nodes with no incoming edges. We initialize $S$ with the nodes that do not have any incoming edges, for Figure 3.5 this would be `A`. The algorithm takes a node $n$ from $S$ and adds it to $L$, it then removes all outgoing edges from $n$ and updates $S$ with the nodes that no longer have any incoming edges. This continues until all nodes are in $L$ and thus we have obtained the topological ordering. The ordered result $L$ is however heavily dependent on the way the nodes are visited. To be able to relate two POAs we want to enforce certain rules upon this visitation.

We start by creating a fixed ordering of all types of activities in the event log. We also create a fixed ordering of the leaf-nodes in the process tree (for which the POAs were created). We now modify the algorithm by making $S$ an ordered set of nodes, which at all times remains sorted, i.e., any newly added elements are automatically sorted. $S$ is sorted in such way that first all move on log only are present and sorted according to the fixed ordering of the types of activities, then second all synchronous moves and model moves are present and sorted according to the fixed ordering of the leaf nodes in the process

| $topoSort(POA(t_1))$ | A | B | Z | C | D | E | C | D | $\tau$ | F | G | - | I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $topoSort(POA(t_2))$ | A | B | - | - | - | - | C | D | $\tau$ | F | G | Z | I |

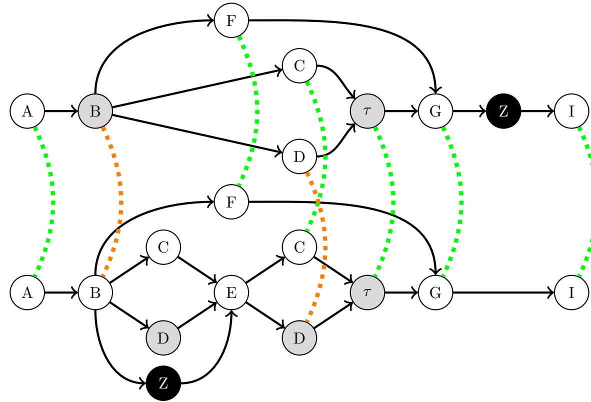


Figure 3.5: The table shows the global alignment of the topological sorts for the POAs of $t_1 = \langle A, B, C, Z, E, C, F, G, I\rangle$ and $t_2 = \langle A, F, C, D, G, Z, I\rangle$. The top POA represents $t_2$ and the bottom POA represents $t_1$. Green dotted lines indicate a proper relation between nodes and orange dotted lines indicate a weak relation.

tree. Because of this sorting the topological sort always returns a flat presentation of the POA that can be optimally aligned to a flat presentation of another POA.

The alignment of the topological sorts is obtained using the global alignment algorithm introduced by Needleman and Wunsch [31]. The resulting global alignment of the flat presentations now provides us information on how the nodes in the POAs are related to each other. The table in Figure 3.5 shows the global alignment of the topological sorts of the example traces.

### 3.3.3 Computing parallel execution difference

Using the POAs we can already calculate a graph edit distance and use this as a distance function. This however fully ignores the difference in the observed execution orderings of activities on which no ordering is enforced by the model. In order to cluster on different execution orderings of these activities we should still penalize differences in these executions. To illustrate this mechanism we use Figure 3.6, which shows two alignments with the same POA.

To compute a score representing this mismatch in parallel behavior, we replay the original alignments on their corresponding POA. For every node that is related to a node in the other POA (can be a weak

| $l_1$ | A | ≫ | C | D | E | C | D | ≫ | F | G | I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\sigma_1$ | A | B | C | D | E | C | D | $\tau$ | F | G | I |

| $l_2$ | A | ≫ | D | C | E | D | C | ≫ | F | G | I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\sigma_2$ | A | B | D | C | E | D | C | $\tau$ | F | G | I |

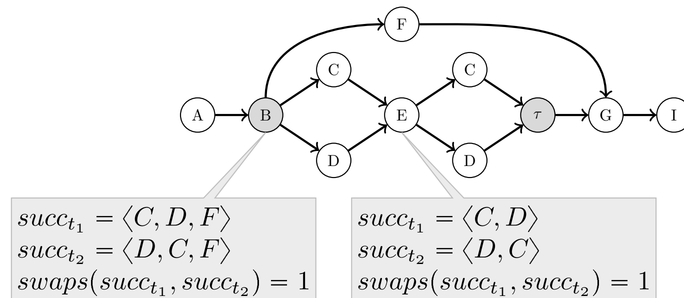



Figure 3.6: Two traces $t_1$ and $t_2$, with identical POAs but different observed execution orderings. This figure illustrates the parallel splits of the POA. It shows the successors and the parallel execution ordering score per split. The final parallel execution ordering score is 2.

or proper relation) that has multiple outgoing arrows, the successor nodes should be further investigated. In the example of Figure 3.6 this would be node `B` and `E`.

We take all successor nodes that are properly related (i.e., they represent the same activity type and move type) and do not represent a move on model only. We ignore move on model only because these moves have no observed execution ordering and therefore cannot be used to inspect the difference in observed execution orderings. We now take the intersection of the two sets of successors and order the sets according to their observed execution in the trace. For the move on model only of activity `B` we would obtain the lists of activities: $\langle C, D, F\rangle$ and $\langle D, C, F\rangle$.

Using permutations it is now possible to calculate how many swaps are required to change the first ordered set of successors to the second ordered set of successors or vice versa. For node `B` this results in one swap. We calculate this number of swaps for every split present in the POA. The sum of the number of swaps is the parallel execution difference.

### 3.3.4 Calculating dissimilarity for a single model

Using the previously described techniques we are now able to define the dissimilarity measure of the Syntactic reasoning approach. We define this measure using a distance function: $d(t_i, t_j, M_n) \rightarrow \{x \in \mathbb{R} : x \geqslant 0\}$. The result of this function is obtained by a multi-step approach. First for both traces a POA is created and the POAs are related to each other, then multiple error-transformation-steps are performed such that both POAs become more similar. After the last step both POAs are identical. Our approach consists of the following error-transformation steps:

1. **Parallel-Difference:** This does not actually transform anything in the POAs but the cost of the parallel execution difference is added to the final distance. Assume the parallel execution difference is $n$, then the total added distance is defined according to the function:

$$c_{pd}(n) = \begin{cases} \log(n+1) \times p_{pd} & \text{if n} > 0 \\ 0 & \text{if n} = 0 \end{cases} \tag{3.1}$$

   Where parameter $p_{pd}$ is a multiplier to influence the cost of the parallel execution difference.

2. **Log to Model synchronization:** The provided model might not represent the traces perfectly. For example, the model might have a $\rightarrow$-operator while it should be an $\wedge$-operator, this causes the alignment algorithm to introduce a move on log only and a move on model only for a single mismatch in the ordering of an activity. This error-transformation merges this move on log only and move on model only to a single synchronous move on the position of the move on model only, but only if the move on log only is not related to a move on log only in the other trace. The cost of a single synchronization is represented by $p_{lms}$. Therefore the cost of $n$ synchronizations is defined by the function:

$$c_{lms}(n) = n \times p_{lms} \tag{3.2}$$

3. **Loop roll-up:** As indicated in Section 3.2 existing approaches can add a lot of distance for just a single extra iteration of a loop, our approach solves this issue by incorporating a loop roll-up error-transformation. Process trees have a loop operator that makes a clear distinction between the *do*, *redo*, and *exit* part of a loop. Using this information we can relate nodes of a POA with the loop construct. Our approach counts the number of iterations for both traces for every loop instance (i.e., an instance starts at the first do-activity and ends before the first exit-activity). Afterwards additional iterations of a loop are added to one POA such that both have the same number of iterations for every loop instance. This way we only penalize the difference in the number of iterations instead of penalizing every individual observed activity.

   The cost of adding loop iterations takes into account that adding a second and third iteration if there was only one iteration should cost more than when adding an $11^{\text{th}}$ and $12^{\text{th}}$ iteration if there were 10 iterations. The idea is that the more iterations there are, the less significant the cost of adding an additional iteration should be. The cost of loop roll-up for a single loop instance is

defined by the function:

$$c_{lru}(i_{high}, i_{low}) = \begin{cases} (\log(i_{high}+1) - \log(i_{low}+1)) \times p_{lru} & \text{if } i_{high} \geqslant i_{low} > 0 \\ \log(i_{high}+1) \times p_{lru} & \text{if } i_{high} > 0 \text{ and } i_{low} = 0 \\ 0 & \text{else} \end{cases} \quad (3.3)$$

Where $i_{high}$ is the number of iterations in the POA with the most iterations, $i_{low}$ is the number of iterations in the POA with the least iterations, and parameter $p_{lru}$ is the multiplier for the logarithmic difference. The final cost of this error-transformation is the sum of the costs per loop instance.

In addition to adding these iterations, this error-transformation also repairs iterations such that the mismatch of a single activity, but repeated in multiple loop iterations, does not result in a big edit-cost in the last step. We only keep the mismatches in one iteration and repair all other occurrences. No additional cost is added for this kind of repair.

4. **Node transfer:** The provided model might not represent the event log perfectly. This can cause for unrelated nodes in the POAs while they should actually have been related. This transformation transfers the node from one place in the POA to a place where it can be properly related to a node in the other POA. The cost of a single node transfer is represented by $p_{nt}$. Therefore the cost of $n$ node transfers is defined by the function:

$$c_{nt}(n) = n \times p_{nt} \quad (3.4)$$

5. **Insertion / Deletion / Substitution:** This is a slightly modified version of a standard graph edit distance. We distinguish between three types of substitutions: (i) Substitution of a non-log-move with a non-log-move that describe different process tree nodes, (ii) Substitution of a non-log-move with a non-log-move that describe the same process tree node, and (iii) all other substitutions. The cost of a single insert / delete, or single substitution of type: (i), (ii) or (iii) is defined respectively by: $p_{id}$, $p_{c1}$, $p_{c2}$ and $p_{c3}$. Therefore the cost of $n_{id}$ insert / deletes, $n_{c1}$ substitutions of type (i), $n_{c2}$ substitutions of type (ii), $n_{c3}$ substitutions of type (iii) is defined by the function:

$$c_{ged}(n_{id}, n_{c1}, n_{c2}, n_{c3}) = n_{id}p_{id} + n_{c1}p_{c2} + n_{c2}p_{c2} + n_{c3}p_{c3} \quad (3.5)$$

Please note that the insertion and deletion costs were not defined separately since these must be identical in order to keep symmetry, i.e., $d(t_i, t_j, M_n) = d(t_j, t_i, M_n)$.

The final distance function $d(t_i, t_j, M_n)$ is defined as the summation of the cost functions of these error-transformations.

### 3.3.5 Aggregating dissimilarity for multiple models

The Syntactic reasoning approach defines the distance function: $d(t_i, t_j, M_n) \to \{x \in \mathbb{R} : x \geqslant 0\}$ to describe the dissimilarity of two traces for a given model. As explained in Section 3.2, adding a process model can provide important context-information about the traces. Evolutionary process discovery algorithms like the ETMd can produce many models which specialize in different aspects. Because of this we might not have a single ideal model, but instead have a lot of models that approach this ideal model. We can calculate distances for each of these models, but for the actual trace clustering we need a single distance. For this we define the following basic aggregation functions:

**Definition 3.1** *(Distance aggregation functions)*
*Let $t_i$ and $t_j$ denote two traces. Let $MC$ denote a collection of process models. We now define four aggregation functions:*

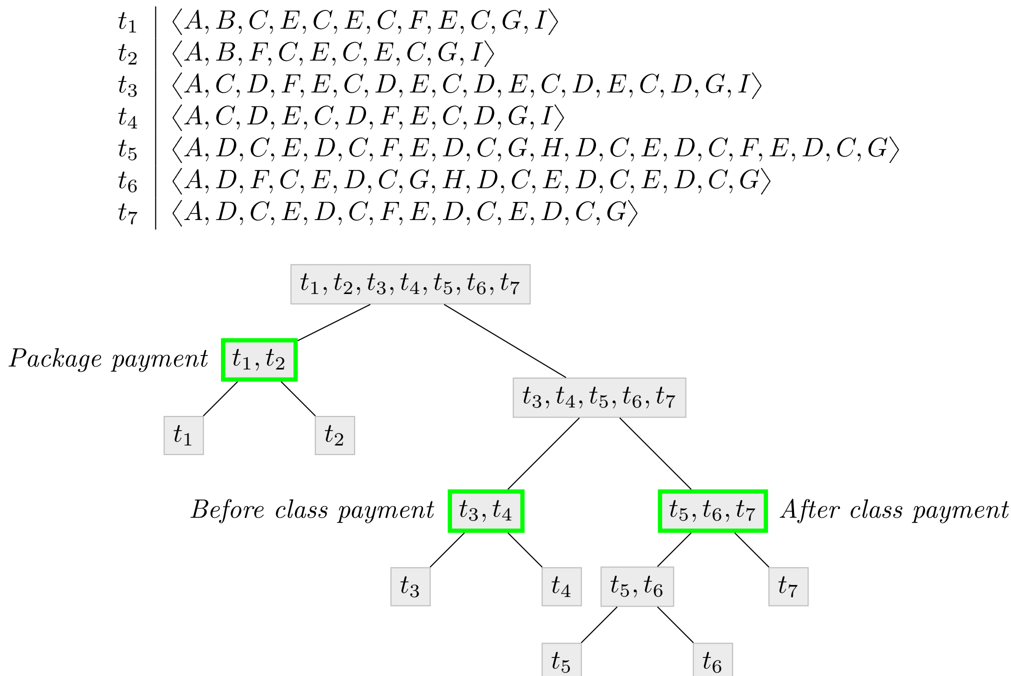
1. ***Maximum:*** $d_{max}(t_i, t_j, MC) = \max\limits_{M_n \in MC} d(t_i, t_j, M_n)$
2. ***Minimum:*** $d_{min}(t_i, t_j, MC) = \min\limits_{M_n \in MC} d(t_i, t_j, M_n)$
3. ***Sum:*** $d_{sum}(t_i, t_j, MC) = \sum_{M_n \in MC} d(t_i, t_j, M_n)$
4. ***Squared:*** $d_{sq}(t_i, t_j, MC) = \sqrt{\sum_{M_n \in MC} d(t_i, t_j, M_1)^2}$

## 3.4 Creating a segregation of traces

Many algorithms exist to perform clustering, e.g., k-means [29], agglomerative/divisive hierarchical clustering [16], DBSCAN [18], OPTICS [7], hierarchical DBSCAN [13], and many more. Some of these algorithms return strict groups of objects while others return hierarchies of objects where in each split the most distant objects are split of from a bigger cluster. For our trace clustering we focus on hierarchical algorithms because: (i) hierarchical algorithms often require less parameters, (ii) parameters of non-hierarchical algorithms are often not trivial and very context-specific, and (iii) end-users of our full approach have more freedom in choosing a selection of clusters and thus retrieving a good set of configurations. The Syntactic reasoning approach (proposed in Section 3.3) is a syntactical approach, meaning that we do not have feature-sets and therefore no features per trace that can be represented as a vector. We therefore cannot place the traces in a dimensional space. This restricts the use of our dissimilarity measure to hierarchical algorithms that do not require this, e.g., agglomerative/divisive hierarchical clustering [16], OPTICS [7] or hierarchical DBSCAN [13].

Hierarchical clustering [16] is one of the most well known clustering algorithms. The bottom-up variant (agglomerative) starts by placing the traces in separate clusters and then merges the closest clusters into a larger cluster until eventually there is one root cluster that contains all traces. Different methods to determine the closest clusters can be used, e.g., (i) single linkage, which takes the distance of the closest two objects, (ii) complete linkage, which takes the distance of the farthest two objects, and (iii) average linkage, which takes the average of the distances between objects of the clusters. Hierarchical DBSCAN [13] is a more recent hierarchical clustering approach and can be seen as an algorithmic and conceptual improvement over OPTICS [7]. Both Hierarchical DBSCAN and OPTICS are density based algorithms and produce hierarchies. Density based algorithms form clusters of traces in the densely populated areas. Objects that fall in between such areas are often discarded as noise.

Figure 3.7 shows a hierarchical clustering of traces (which are executions of the running example) using the Syntactic reasoning approach as a dissimilarity measure. Please note that some clusters have green borders. These are the clusters that distinguish the different configurations of the running example, we call this the *preferred clusters*.



| | |
|---|---|
| $t_1$ | $\langle A, B, C, E, C, E, C, F, E, C, G, I \rangle$ |
| $t_2$ | $\langle A, B, F, C, E, C, E, C, G, I \rangle$ |
| $t_3$ | $\langle A, C, D, F, E, C, D, E, C, D, E, C, D, E, C, D, G, I \rangle$ |
| $t_4$ | $\langle A, C, D, E, C, D, F, E, C, D, G, I \rangle$ |
| $t_5$ | $\langle A, D, C, E, D, C, F, E, D, C, G, H, D, C, E, D, C, F, E, D, C, G \rangle$ |
| $t_6$ | $\langle A, D, F, C, E, D, C, G, H, D, C, E, D, C, E, D, C, G \rangle$ |
| $t_7$ | $\langle A, D, C, E, D, C, F, E, D, C, E, D, C, G \rangle$ |



Figure 3.7: The table shows traces which represent executions of the running example. The tree represents a hierarchical clustering of these traces. The clusters with green borders represent the preferred clusters. The three preferred clusters in this hierarchy represent the three variants of the running example.

Many algorithms exist to extract such preferred clusters. The most basic approaches simply retrieve all clusters with a minimum number of objects, or cut the hierarchy when a certain height threshold is reached (the height is the distance between the clusters that are split of). These methods perform global cuts of the hierarchy. Campello et al. [13, p.166–168] proposes a method that instead performs a local cut of the hierarchy by using the notion of cluster stability. In our approach we wish to keep the hierarchy for the next step but we do use this local cut method to suggest the most preferable clusters to the end-user.

## 3.5 Conclusion

We have explained the relevance of trace clustering in process discovery, by finding homogeneous groups of traces the complexity of discovered models may be reduced. We applied trace clustering to find groups of traces that imply configurations on a process model. Clustering algorithms use some kind of dissimilarity measure, we explained how existing approaches towards defining this measure suffer from problems because of their lack of model-awareness. We proposed a new dissimilarity measure: the Syntactic reasoning approach, which includes process models. By adding model-awareness we are able to adequately identify parallel executable behavior and loops. Finally we have explained how our dissimilarity measure can be used in clustering and how preferred clusters can be suggested to the end-user. The final result is a hierarchical clustering that shows a segregation of the input event log for a given collection of process trees. In Chapter 4 we explain how we can relate the data of the event log to this hierarchical clustering. This provides insights to the end-user in how the data implies behavior as observed in the clusters and finally how the data implies configurations.

# Chapter 4

# Explaining clusters by data

In Chapter 3 we explained how we obtain a hierarchical clustering. Each cluster in this hierarchy is a group of traces. The root of the hierarchy contains the full event log, then at every level the most dissimilar groups are split of. In this chapter we answer the second research question: **How can we explain these groups based on data?** With these groups we refer to the clusters in the hierarchy.

A similar issue is observed in the area of *decision mining* [35]. Decision mining aims at gaining insight into the data perspective of processes. With an event log and a process model we can replay the traces of this event log on the process model. At some points in this process model choices are made. The points where a choice can be made are called decision points. In process trees the decision points are represented by the operators $\times$ and $\vee$ which provide a choice between activities, and $\circlearrowleft$ which provides a choice between exiting the loop and entering another iteration. Decision mining tries to explain the choices made at the decision points using the data of the traces.

We can relate to this problem since the traces are clustered on their behavioral similarity, i.e., the traces are clustered since similar choices were made during execution. Different in our approach from decision mining is that we do not want to explain each choice individually but instead want to explain a collection of choices (represented by a cluster) by the data of the traces. For any hierarchical clustering we wish to *annotate* every split in the hierarchy with an explanation based on data. If such an explanation does not exist we wish to reduce the hierarchy. Hierarchical clusterings can become huge and incomprehensible. Under the assumption that splits which cannot be explained by data are invalid, we can remove such splits. A comprehensible annotated hierarchical clustering provides insights to an end-user in how data implicates a cluster and finally a configuration of the process model.

Assume we have the same traces and partially the hierarchical clustering as shown previously in Figure 3.7. Assume the traces contain the data as shown in Table 4.1. After performing annotation we wish to find a hierarchical clustering as shown in Figure 4.1. This hierarchical clustering clearly shows the three variants of the running example.

Section 4.1 explains how we annotate the hierarchical clustering based on data. Then in Section 4.2 we discuss a method to reduce the hierarchical clustering. In Section 4.3 we explain how the trace data can be enriched to obtain better results. Finally in Section 4.4 we conclude this chapter.

| $t_{\#}$ | $Trace$ | $Payment$ | $Classes\#$ | $Failures\#$ |
|---|---|---|---|---|
| $t_1$ | $\langle A, B, C, E, C, E, C, F, E, C, G, I \rangle$ | Package | 4 | 0 |
| $t_2$ | $\langle A, B, F, C, E, C, E, C, G, I \rangle$ | Package | 3 | 0 |
| $t_3$ | $\langle A, C, D, F, E, C, D, E, C, D, E, C, D, E, C, D, G, I \rangle$ | AfterClass | 5 | 0 |
| $t_4$ | $\langle A, C, D, E, C, D, F, E, C, D, G, I \rangle$ | AfterClass | 3 | 0 |
| $t_5$ | $\langle A, D, C, E, D, C, F, E, D, C, G, H, D, C, E, D, C,$ $F, E, D, C, G \rangle$ | BeforeClass | 6 | 1 |
| $t_6$ | $\langle A, D, F, C, E, D, C, G, H, D, C, E, D, C, E, D, C, G \rangle$ | BeforeClass | 5 | 1 |
| $t_7$ | $\langle A, D, C, E, D, C, F, E, D, C, E, D, C, G \rangle$ | BeforeClass | 4 | 0 |

Table 4.1: Executions of the running example with data attributes.

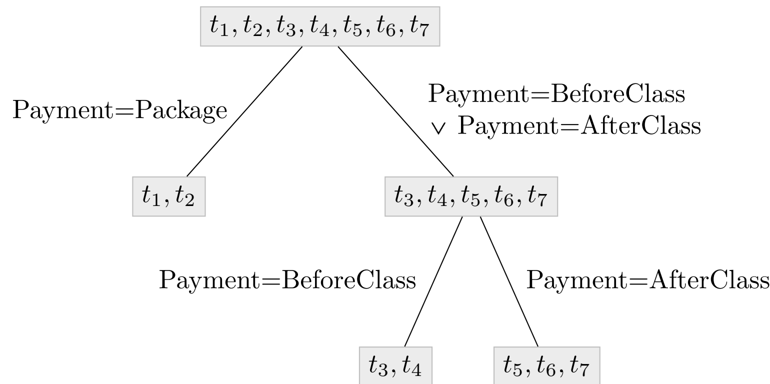



Figure 4.1: The reduced annotated hierarchical clustering of the example given in Figure 3.7 using the data set of Table 4.1.

## 4.1 Annotating a hierarchical clustering

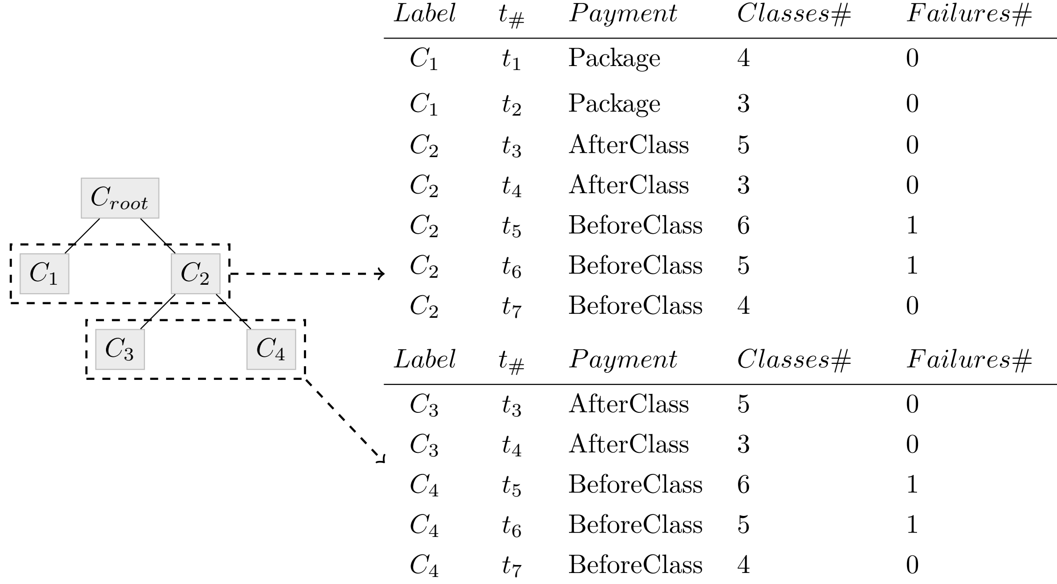
In this section we propose two methods for annotating the hierarchical clustering. We annotate the hierarchical clustering using the data of the traces. In order to do this we transform our problem to a *classification* problem. In classification it is required to have a set of objects for which we know the class, in our case the traces are the objects and the class is the cluster to which they are assigned. Using this input the classification algorithm aims at forming an optimal internal presentation that represents the input classification of the training objects. The resulting classifier can now be used on new objects to assign them to one of the classes. Figure 4.2 shows how for each hierarchy split a training set for the classification algorithm is created. Every cluster is given a class label, and every trace in a cluster is an object in the training set under this label. Every attribute in the data of the traces is a column in the training set. The events of a trace can contain data as well, the training set can be expanded with this data. A classification algorithm can now be asked to create a classifier for this training set.

Different from normal classification is that we are not interested in using the classifier to assign new objects to one of the existing classes, but instead want to use the internal rules of the classifiers to annotate the hierarchical clustering. Many algorithms exist to perform classification, e.g., Bayesian classifier [17], C4.5 decision trees [33], nearest neighbor [15], support-vector networks [14], and many more. However most of these algorithms have internal classification rules that are incomprehensible for people while we want to annotate the hierarchy in such way that people can reason about it. Algorithms like the *Bayesian classifier* and *C4.5 decision trees* (or decision trees in general) do however produce a comprehensible set of rules, we therefore only consider these algorithms.

The *Bayesian classifier* [17] is a statistical approach towards classification. The naive variant of the Bayesian classifier is a probabilistic approach based on the application of Bayes' theorem and assumes that all variables (i.e., the attributes of the training set) are independent. Figure 4.3 is used to illustrate this algorithm. First for any attribute in the training set the probability that a certain value is observed within a class is calculated, Figure 4.3b shows such probabilities for the training set shown in Figure 4.3a. Then using these probabilities and Bayes' theorem we can calculate the probability that an object belongs to a class given the attribute values. The object is assigned to the class with the highest probability. We can annotate the splits in the hierarchical clustering with the probabilities that attributes have certain values given a class. Such an annotation is partially shown in Figure 4.3c.

We observe that for a small example these annotations already become quite complex. To solve this we could order the probabilities on their value and remove any that have a probability below a certain threshold where they are no longer interesting. This however only partially solves the issue and annotations might still become incomprehensible.

Like the Bayesian classifier, the *C4.5 decision tree* [33] algorithm is a statistical approach. To illustrate the C4.5 algorithm we use Figure 4.4. The C4.5 algorithm uses the concept of information entropy. Information entropy is a measure which describes the uncertainty in a random variable. It creates a classifier by splitting the entries of the training set on attributes that provide the most information gain (i.e., normalized difference in information entropy). The algorithm keeps splitting the training set until




| *Label* | $t_{\#}$ | *Payment* | *Classes#* | *Failures#* |
|---|---|---|---|---|
| $C_1$ | $t_1$ | Package | 4 | 0 |
| $C_1$ | $t_2$ | Package | 3 | 0 |
| $C_2$ | $t_3$ | AfterClass | 5 | 0 |
| $C_2$ | $t_4$ | AfterClass | 3 | 0 |
| $C_2$ | $t_5$ | BeforeClass | 6 | 1 |
| $C_2$ | $t_6$ | BeforeClass | 5 | 1 |
| $C_2$ | $t_7$ | BeforeClass | 4 | 0 |

| *Label* | $t_{\#}$ | *Payment* | *Classes#* | *Failures#* |
|---|---|---|---|---|
| $C_3$ | $t_3$ | AfterClass | 5 | 0 |
| $C_3$ | $t_4$ | AfterClass | 3 | 0 |
| $C_4$ | $t_5$ | BeforeClass | 6 | 1 |
| $C_4$ | $t_6$ | BeforeClass | 5 | 1 |
| $C_4$ | $t_7$ | BeforeClass | 4 | 0 |

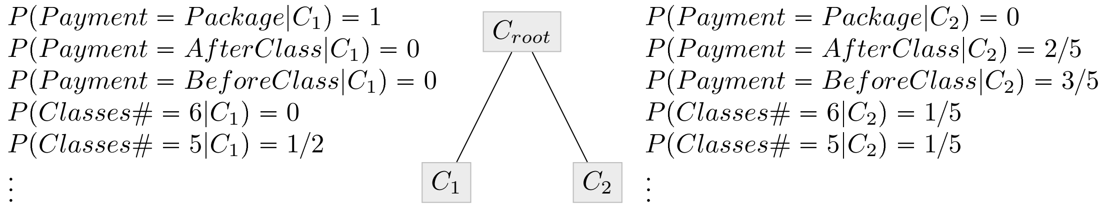
Figure 4.2: The transformation from cluster splits (on the left) to training sets (on the right) to use as input for a classification algorithm.

| *Label* | $t_{\#}$ | *Payment* | *Classes#* | *Failures#* |
|---|---|---|---|---|
| $C_1$ | $t_1$ | Package | 4 | 0 |
| $C_1$ | $t_2$ | Package | 3 | 0 |
| $C_2$ | $t_3$ | AfterClass | 5 | 0 |
| $C_2$ | $t_4$ | AfterClass | 3 | 0 |
| $C_2$ | $t_5$ | BeforeClass | 6 | 1 |
| $C_2$ | $t_6$ | BeforeClass | 5 | 1 |
| $C_2$ | $t_7$ | BeforeClass | 4 | 0 |

(a) Training set

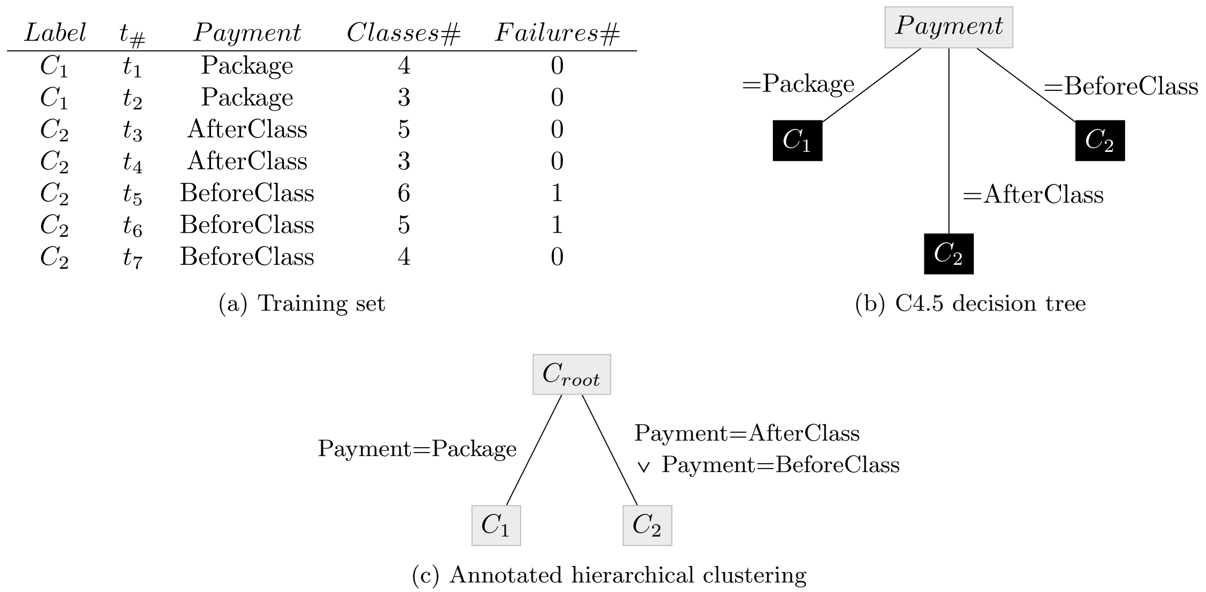
| *Att* | *Val* | $P(Att = Val \mid C_1)$ | $P(Att = Val \mid C_2)$ |
|---|---|---|---|
| Payment | Package | 1 | 0 |
| Payment | AfterClass | 0 | 2/5 |
| Payment | BeforeClass | 0 | 3/5 |
| Classes# | 6 | 0 | 1/5 |
| Classes# | 5 | 1/2 | 2/5 |
| Classes# | 4 | 1/2 | 1/5 |
| Classes# | 3 | 1/2 | 1/5 |
| Failures# | 1 | 0 | 2/5 |
| Failures# | 0 | 1 | 3/5 |

(b) Probabilities of attribute values with respect to a class




(c) Annotated hierarchical clustering

Figure 4.3: The internal probabilistic model (b) of a naive Bayesian classifier for the training set (a), and the resulting annotated hierarchical clustering (c).



| *Label* | $t_{\#}$ | *Payment* | *Classes#* | *Failures#* |
|---|---|---|---|---|
| $C_1$ | $t_1$ | Package | 4 | 0 |
| $C_1$ | $t_2$ | Package | 3 | 0 |
| $C_2$ | $t_3$ | AfterClass | 5 | 0 |
| $C_2$ | $t_4$ | AfterClass | 3 | 0 |
| $C_2$ | $t_5$ | BeforeClass | 6 | 1 |
| $C_2$ | $t_6$ | BeforeClass | 5 | 1 |
| $C_2$ | $t_7$ | BeforeClass | 4 | 0 |

(a) Training set



(b) C4.5 decision tree



(c) Annotated hierarchical clustering

Figure 4.4: The decision tree (b) created by the C4.5 algorithm for the training set (a), and the resulting annotated hierarchical clustering (c).

none of the attributes provide any information gain or if the active set contains a single class label. In a decision tree all nodes (except the leafs) contain an attribute, and every outgoing edge from these nodes are labeled with the value expression on this attribute. Any leaf of a decision tree is a class label. Figure 4.4b shows such a decision tree for the training set shown in Figure 4.4a. By traversing the resulting decision tree from its root to the leafs we can create a logical expression that explains a class. This logical expression can now be used to annotate the hierarchical clustering, Figure 4.4c shows an example of such a hierarchy. We already observe that for this small example the annotations generated by the C4.5 algorithm are much easier than the ones generated by the naive Bayesian classifier.

Using the C4.5 algorithm can however cause contradictions in the resulting annotated hierarchical clustering. Because every split of the hierarchical clustering is classified locally and the C4.5 algorithm does not always perform fully accurate splits, it can occur that after an 'equal to' expression there are still objects in the cluster that contain a different value for the attribute. After the split on the attribute the information gain of the attribute drops, but the information gain might increase again when clusters become smaller. This can cause the C4.5 algorithm to split again on this attribute, which may lead to a contradiction in the final hierarchy. We can solve this issue by: (i) stop annotating the hierarchy when a contradiction occurs, or (ii) remove traces from the cluster that do not match the annotation expression.

Both the naive Bayes and C4.5 algorithm are suitable to annotate a hierarchical clustering. However the naive Bayes classifier often produces annotations which are difficult to understand. This is in contrast with the C4.5 algorithm which produces naturally readable annotations but can result in contradictions. In this thesis our main focus is therefore on the C4.5 decision tree [33] algorithm. However as we have shown other classification algorithms can also be used.

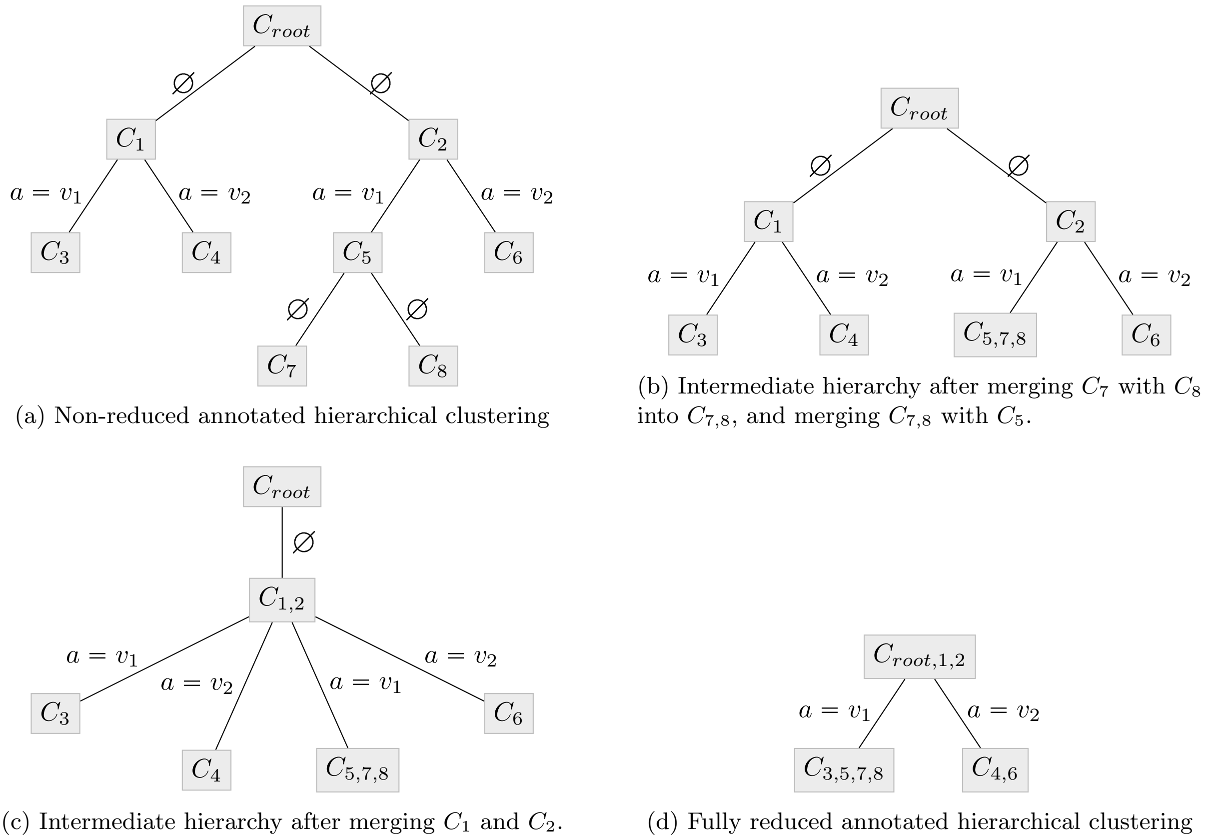
## 4.2 Reducing an annotated hierarchy

Hierarchical clusterings can become huge and incomprehensible for many traces. Often such hierarchies are already reduced by the clustering algorithm, e.g., (i) by cutting the hierarchy at a certain split-height threshold (the height of a split is the value of the dissimilarity measure between the clusters that are split of), or (ii) by cutting the hierarchy at a minimal number of objects. These methods are also applicable in our approach but could result in the loss of information. By doing this we might remove useful clusters from the hierarchy. We propose an approach that uses the annotations to remove irrelevant parts of the hierarchical clustering. In our approach we make the assumption that splits which are annotated with the same expression do not distinguish different behavior and should therefore be merged. Our

approach starts at the leafs and works its way up to the root. At every split it merges children with equal expressions. If only one child of a node remains, we merge the node with the single child. After nodes are merged, the children of the resulting merged node may again have equal expressions. Therefore the children of the merged node should be investigated again.

Figure 4.5a shows a non-reduced annotated hierarchical clustering, which we use as an example. The empty expression (i.e., there is no explanation for the cluster) is denoted by: $\varnothing$. The algorithm starts by merging clusters $C_7$ and $C_8$ into a cluster $C_{7,8}$. This results in $C_5$ having a single child, $C_5$ is therefore merged with $C_{7,8}$ into cluster $C_{5,7,8}$. During every merge, the expressions related to merging clusters are combined using a $\wedge$. Since both $C_7$ and $C_8$ yield the empty expression, the expression of $C_{5,7,8}$ stays $a_1 = v_1$ (the intermediate result is shown in Figure 4.5b). Next are cluster $C_1$ and $C_2$ which have an equal expression (the empty expression) and are therefore merged into the cluster $C_{1,2}$, the resulting cluster $C_{1,2}$ has four children: $C_3$, $C_4$, $C_{5,7,8}$ and $C_6$ (the intermediate result is shown in Figure 4.5c). These children should now be merged into two clusters: $C_{3,5,7,8}$ and $C_{4,6}$. Earlier $C_1$ and $C_2$ were merged, because of this the $C_{root}$ contains only a single child and should therefore be merged with this child, turning it into $C_{root,1,2}$. The final result is shown in Figure 4.5d.

We observe that our approach can greatly reduce the hierarchical clustering if it is properly annotated. Hierarchies with only annotations yielding the empty expression are however reduced to a single cluster. A second issue is that not every interesting cluster might be explainable by data, this can cause such clusters to be removed from the hierarchy. The proposed approach is therefore only suitable in situations where the data describes the interesting clusters. In cases where this is not the case we might be required to skip this step and let the clustering algorithm reduce the hierarchical clustering using other approaches such as cutting the hierarchy at a minimum number of objects.




(a) Non-reduced annotated hierarchical clustering

(b) Intermediate hierarchy after merging $C_7$ with $C_8$ into $C_{7,8}$, and merging $C_{7,8}$ with $C_5$.

(c) Intermediate hierarchy after merging $C_1$ and $C_2$.

(d) Fully reduced annotated hierarchical clustering

Figure 4.5: Situation before reduction (a). Followed by two intermediate stages (b, c) and the final reduced hierarchy based on the data annotations (d). The symbol $\varnothing$ denotes the empty expression.

## 4.3 Enrichment of trace data

Event logs may lack data that implies behavior. Without adding additional data to such event logs it is impossible to annotate the hierarchical clustering. By enriching the event log with derived data we can solve this issue. Section 3.1.1 discussed dissimilarity measures of traces that are based on feature-sets. These feature-set approaches try to abstract from traces by finding features that identify the trace. We can enrich the data of the traces with such features. The classification algorithm might find a direct relation between a feature and a cluster. Some examples of features are:

1. **Occurrence:** For any activity in the trace we can count the number of occurrences. We can add this as a numeric attribute to the trace data.
2. **Direct-succession:** We can count the number of times an activity directly succeeds another activity and add this as a feature.

Table 4.2 partially shows a data set that is enriched with the occurrence and direct-succession features. For this enriched data set the C4.5 algorithm can relate the occurrence and direct-succession of activities to the variants of the running example. For example, the cluster containing traces of variant one would be annotated with the expression $\#B = 1$, which describes the execution of activity *Buy Classes + Exam* (B).

Many other feature-sets like k-grams [24] or context-aware feature-sets [23, p.81] could be included to improve annotations results. The classification algorithm automatically decides which features describe the clustering in the best way.

| $t_{\#}$ | Trace | $\#A$ | $\#B$ | $\#C$ | ... | $C > D$ | $D > C$ | ... |
|---|---|---|---|---|---|---|---|---|
| $t_1$ | $\langle A, B, C, E, C, E, C, F, E, C, G, I \rangle$ | 1 | 1 | 4 | ... | 0 | 0 | ... |
| $t_2$ | $\langle A, B, F, C, E, C, E, C, G, I \rangle$ | 1 | 1 | 3 | ... | 0 | 0 | ... |
| $t_3$ | $\langle A, C, D, F, E, C, D, E, C, D, E, C, D, E, C, D, G, I \rangle$ | 1 | 0 | 5 | ... | 5 | 0 | ... |
| $t_4$ | $\langle A, C, D, E, C, D, F, E, C, D, G, I \rangle$ | 1 | 0 | 3 | ... | 3 | 0 | ... |
| $t_5$ | $\langle A, D, C, E, D, C, F, E, D, C, G, H, D, C, E, D, C, F, E, D, C, G \rangle$ | 1 | 0 | 6 | ... | 0 | 6 | ... |
| $t_6$ | $\langle A, D, F, C, E, D, C, G, H, D, C, E, D, C, E, D, C, G \rangle$ | 1 | 0 | 5 | ... | 0 | 5 | ... |
| $t_7$ | $\langle A, D, C, E, D, C, F, E, D, C, E, D, C, G \rangle$ | 1 | 0 | 4 | ... | 0 | 4 | ... |

Table 4.2: The traces of Table 4.1 enriched with feature data attributes.

## 4.4 Conclusion

Decision mining tries to explain every choice made during the execution of a process model by the data of the traces. By doing this a direct relation between data and behavior may become visible. We can relate to this since the groups of traces in the hierarchical clustering are grouped together because they describe similar behavior, i.e., they made similar choices. We annotate a hierarchical clustering with the data of these traces such that the end-user can pick a set of preferred clusters. We have shown how our problem can be transformed to a classification problem and have proposed ways to use the naive Bayes classifier and C4.5 decision tree classifier. After annotating the hierarchical clustering it may still be huge and incomprehensible for an end-user, we therefore proposed an algorithm that can reduce such a hierarchy based on the data annotations. This algorithm is however only effective if the trace data actually describes the interesting clusters. Finally, we explained how we can enrich event logs which lack data that implies behavior. We can add trace-features as data to the traces such that the hierarchical clustering can be annotated. The final result is a (reduced) annotated hierarchical clustering. This hierarchy can be presented to the end-user such that a selection of preferred clusters can be made. Chapter 5 proposes a new approach towards discovery of a configurable process tree for these preferred clusters and proposes a method to evaluate the quality of a configurable process tree.

# Chapter 5

# Configurable process discovery

In this chapter we answer the third research question: **How can we create a configuration for a given process tree and a group of traces?** And the fourth research question: **How can we select a good configurable process tree for a selection of groups of traces?**

Best practices, legal obligations or enterprise systems cause organizations to organize their business processes in very similar ways [21]. *Configurable process models* provide the means to deal with variability in processes. A configurable process model can combine many different variants of the same process and then allow an organization to configure the process in such way that it best suits their way of working.

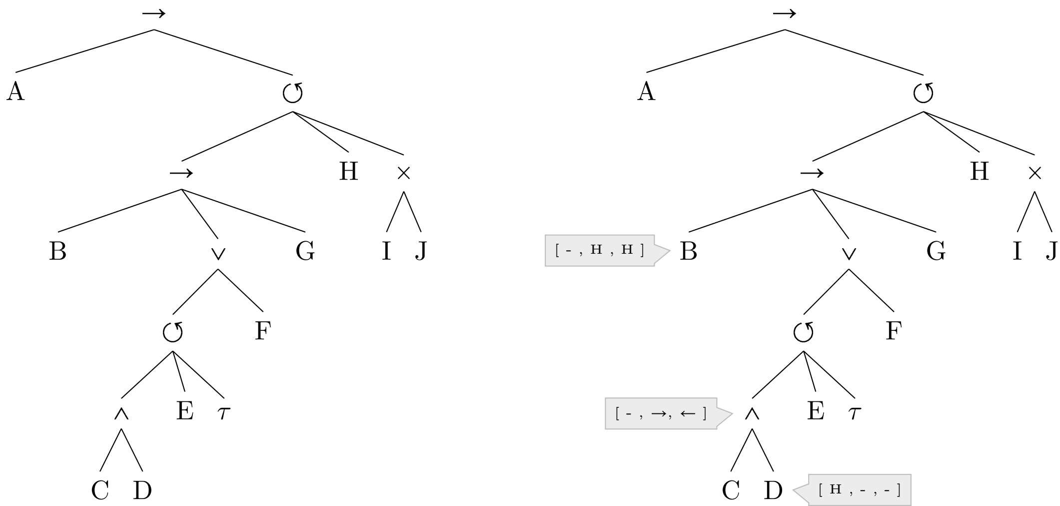
*Configurable process discovery* is directly related to our third research question. In Chapter 4 we explained how we annotate a hierarchical clustering with the data of traces and events. Preferred clusters can be hand-picked from a hierarchical clustering or automatically determined. Every preferred cluster is a group of traces for which we try to find an optimal configuration. Figure 5.1a shows three groups of traces: $C_1$, $C_2$ and $C_3$, which are executions of the running example. Figure 5.1b shows a process tree that is used as input for a configurable process discovery algorithm. We would now like to find a configurable process tree like the one presented in Figure 5.1c. The callouts in Figure 5.1c show the configurations options per configuration, e.g., `B` is annotated with: `[ - , H , H ]`, indicating that for $C_1$ the node should be allowed, and for $C_2$ and $C_3$ the node should be hidden.

In some situations we might not have a single reference model but instead multiple reference models. Discovering a process tree using an evolutionary process discovery algorithm like the ETMd can produce many models which specialize in different aspects. Another reason could be that due to acquisitions, take-overs or mergers we obtain multiple process models for similar processes [19]. Merging these process models into a single reference model is not trivial and can lead to several merged models. Using our configurable process discovery approach we can create a configurable variant for any of these models. However it is not trivial which of these models is the best. By answering the fourth research question we solve this issue by providing a means to *evaluate* the configurable process tree on different quality characteristics.

Section 5.1 discusses existing work on discovering a configurable process model and their common issues. In Section 5.2 we propose our configurable process discovery approach that uses observed execution orderings and execution frequencies to find configuration options. Then in Section 5.3 we propose a method for evaluating the quality of configurable process models. Finally we conclude this chapter in Section 5.4.

| $t_{\#}$ | Trace | Cluster |
|---|---|---|
| $t_1$ | $\langle A, B, C, E, C, E, C, F, E, C, G, I \rangle$ | $C_1$ |
| $t_2$ | $\langle A, B, F, C, E, C, E, C, G, I \rangle$ | $C_1$ |
| $t_3$ | $\langle A, C, D, F, E, C, D, E, C, D, E, C, D, E, C, D, G, I \rangle$ | $C_2$ |
| $t_4$ | $\langle A, C, D, E, C, D, F, E, C, D, G, I \rangle$ | $C_2$ |
| $t_5$ | $\langle A, D, C, E, D, C, F, E, D, C, G, H, D, C, E, D, C, F, E, D, C, G \rangle$ | $C_3$ |
| $t_6$ | $\langle A, D, F, C, E, D, C, G, H, D, C, E, D, C, E, D, C, G \rangle$ | $C_3$ |
| $t_7$ | $\langle A, D, C, E, D, C, F, E, D, C, E, D, C, G \rangle$ | $C_3$ |

(a) Traces that are clustered into three clusters $C_1$, $C_2$ and $C_3$, which represent the variants of the running example.




(b) Reference process tree, which is the unconfigured running example.

(c) The discovered configurable process tree.

Figure 5.1: Table of traces clustered into three clusters (a), a reference process tree (b), and the resulting configurable process tree (c).

## 5.1 Related work

A configurable process model can be discovered in many different ways. Buijs et al. [12] proposes four different approaches and evaluates these. Every approach starts with a collection of event logs. We can relate to this since any preferred group of traces, can be seen as an individual event log. The four approaches are:

**Approach 1:** The first approach, shown in Figure 5.2, discovers a process model for every input event log. Then these models are merged to obtain a single configurable process model. This approach was first proposed by Gottshalk et al. [20]. Merging process models is however no trivial task. Many different merge methods were proposed [12, 19, 27, 36]. A major problem of this approach is that when process models cannot be related properly, the merging algorithm simply produces a choice between the input models.

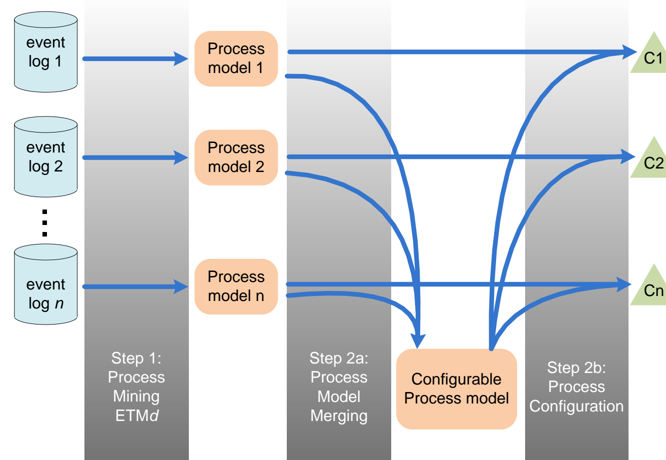



Figure 5.2: Merge individually discovered process models [12].

**Approach 2:** The second approach, shown in Figure 5.3, tries to improve the first approach by first merging the event log. Next a common process model is discovered from this merged event log. This common process model is then individualized per input event log. After which these individual models are merged into a configurable process model. The idea is that by first making a common process model and deriving the individual models from this, it becomes easier to merge them into a configurable process model [12].

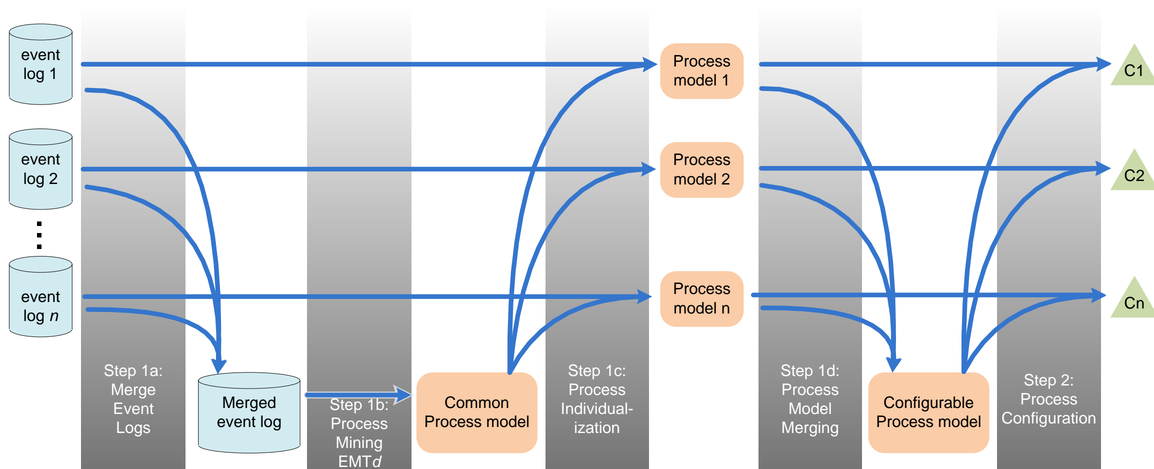



Figure 5.3: Merge similar discovered process models [12].

**Approach 3:** The third approach, shown in Figure 5.4, works by discovering a single process model that describes the behavior of all event logs. Then for each individual event log a configuration is discovered for this process model such that it fits this event log best. In this approach the single process model should be less precise since it is only restricted by the configurations but never extended [12].

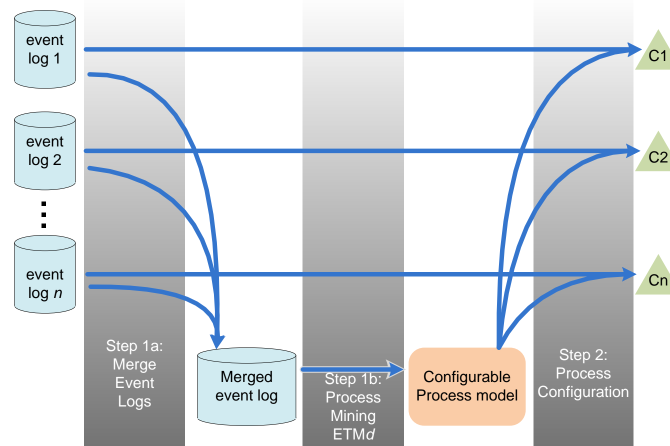



Figure 5.4: First discover a single process model then discover configurations [12].

**Approach 4:** The fourth approach, shown in Figure 5.5, is an approach wherein the discovery of the process model and the configurations is combined. By doing this it might overcome the disadvantages of the previous three approaches. By providing this integration, a better trade-off between process model structure and configuration options can be made [12].

The ETM [9] is evolutionary process mining framework that uses mutators. Every generation mutations are made by the mutators in the process models. After each generation the best (according to the specified quality characteristics) models are kept. Buijs et al. [12] extends the ETM with a mutator that changes configuration options. They also add a metric representing the quality of the configuration perspective. The ETM can therefore automatically make a trade-off between the process model quality and the quality of the configuration perspective.

Assy et al. [8] proposes an interactive discovery approach. Their approach tries to assist the end-user in creating a final configurable process model. They first mine configurable process fragments from a collection of event logs. For these fragments guidelines are mined that assist in choosing a configuration of such a fragment. After every configuration step these guidelines are dynamically updated in order to take the already chosen configurations into consideration.

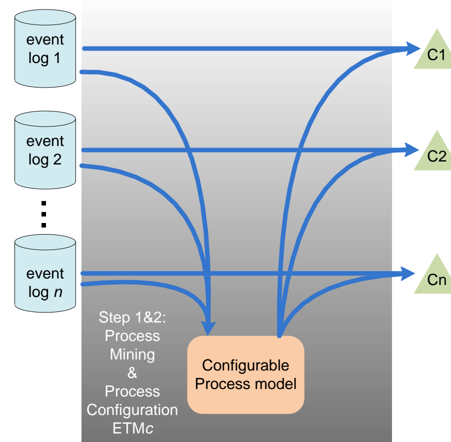



Figure 5.5: Discover process model and configurations at the same time [12].

Buijs et al. [12] evaluated these four approaches and found that approach one and two struggle with merging process models based on their behavior. The individual models that need to be merged may not be similar at all, making it very hard to merge them appropriately. Also these approaches only consider the model structure but lack knowledge of the frequencies that parts of a model are executed. The second approach performs slightly better, although the resulting configurable process model may still remain complex. The third and fourth approach seem to be able to better generalize the behavior into a configurable process model. Finally the fourth approach seems to provide the most flexibility. In Section 5.2 we propose an approach that can find configurations for any event log, given a process tree. Our approach is therefore applicable in step two of approach three but can also be incorporated in approach four.

## 5.2 Discovering process tree configurations

Section 5.1 discussed four different approaches towards configurable process discovery. Approach three and four seem to provide the best results in general. For these approaches Buijs et al. [12] extended the ETM framework with a mutator that randomly selects a node in the process tree and randomly changes the node its configuration options to one of the allowed options for that node. This is however a brute force approach that in the worst case covers the full search space.

We propose a new method that uses the execution frequencies and observed execution orderings (of parts of the process tree) to discover a configuration for a given process tree and event log. The final configurable model is obtained by combining the configurations and use this as the final set of configuration options. Our method is directly applicable in step two of approach three (shown in Figure 5.4). It can however easily be modified into a mutator for the ETM framework. By doing this the ETMc algorithm can directly make a trade-off between the process model structure and the configuration options. Inclusion in the ETM framework will make it applicable in approach four (shown in Figure 5.5).

We start by explaining the use of alignments and their mapping to process trees in Section 5.2.1. Section 5.2.2 explains how we discover parts of the process tree that need to be blocked or hidden. Section 5.2.3 explains how we discover operator downgrades. Finally in Section 5.2.4 we explain how we deal with loops with respect to operator downgrading.

### 5.2.1 Mapping alignments

Alignments provide a mapping between the observed behavior (as present in the event logs) and the modeled behavior (process tree). We use this mapping to identify points in the process tree that require configuration options for an event log. We start by discarding any move on log only, these moves describe behavior that was observed in the event log, but not allowed by the process tree. Since we are only restricting the process tree and not adding any behavior, these moves cannot be used in discovering configuration options. Next any move in the alignment is mapped to its related node (since only synchronous moves and model moves remain, we always have a mapped node in the tree) and all parents of the related node. We do however keep a segregation of traces in this mapping per node.

Figure 5.6 shows the creation of a mapping for leaf `C` and all of its parents. The full mapping should however always consist of a mapping between every node in the process tree and the corresponding alignment moves. The root node of the process tree always contains every move of every alignment which is not a move on log only.

### 5.2.2 Hiding and blocking

After mapping the alignments our method starts by investigating the strongest configuration options: *hide* and *block*. The idea is that if within an event log a part of the process tree is not executed frequently we should either hide or block it. The hide option indicates that a node can be skipped over and the block options indicates that a node cannot be traversed (and also not skipped over). If all children of a node are blocked, a single child of an $\wedge$, $\rightarrow$, $\leftarrow$-node is blocked, or the do or exit-child child of a $\circlearrowleft$-node is blocked, it indicates that the node itself is blocked as well.

To find the hide and block options we use a top-down approach, starting at the root of the process tree. For every node in the process tree, we compare the fraction of the traces related to this node with

| $l_1$ | A | $\gg$ | C | D | $\gg$ | F | G | I |
|---|---|---|---|---|---|---|---|---|
| $\sigma_1$ | A | B | C | D | $\tau$ | F | G | I |

| $l_2$ | A | B | C | $\gg$ | F | G | I |
|---|---|---|---|---|---|---|---|
| $\sigma_2$ | A | B | C | $\tau$ | F | G | I |

Figure 5.6: The boxes around the process tree show the mapping between the moves of the alignments (as shown in the top for a trace $t_1$ and a trace $t_2$ in which all move on log only are already discarded) and the process tree for leaf `C` and all of its parents.

respect to the full event log. If this fraction is below a certain threshold we should perform hiding or blocking. When we have to hide or block we no longer visit the children of the node and skip to the first node that is not part of the sub-tree. By default we assume that a node should be blocked. However to avoid propagation of the block to the parent(s), we should instead hide the node if the parent of the node is an operator of type: $\rightarrow$, $\leftarrow$, $\wedge$, or if the node is the do or exit-child of a $\circlearrowleft$-node, or if all other children of the parent are already blocked. We should also hide the node if it is the root-node. Hiding and blocking is defined as follows:

**Definition 5.1** *(Hiding and blocking)*
*Let $L$ be the full event log and $L_n \subseteq L$ the collection of traces related to a node $n$ of a process tree. Let $t_{hb}$ be a given threshold, with $0 \leqslant t_{hb} \leqslant 1$. Let $p(n)$ denote the parent of node $n$ or undefined if $n$ is the root-node. Let $c(n)$ denote the children of node $n$. Let $m(n)$ denote the type of node $n$. We should hide or block if and only if:*

$$\frac{|L_n|}{|L|} < 1 - t_{hb} \tag{5.1}$$

*We choose an appropriate downgrade option as follows:*

- *If $p(n)$ is undefined, then hide*
- *If $m(p(n)) \in \{\rightarrow, \leftarrow, \wedge\}$, then hide*
- *If $m(p(n)) = \circlearrowleft$ and $n$ is the do or exit-child of $p(n)$, then hide*
- *Let $c_{other} = \{c_i \in c(p(n)) : c_i \neq n\}$, if $\forall c_i \in c_{other} : c_i$ is blocked, then hide*
- *Else, block*

According to this definition a threshold of $t_{hb} = 0.95$ would indicate that if less than 5% of the traces execute the node, it should be hidden or blocked. After visiting the full tree we have obtained the initial configuration with only hide and block options. In the next step we determine possible downgrade options of operators.

### 5.2.3 Downgrading operators

After determining the hide and block options we should find the *downgrade* options for the operator-nodes (i.e., the non-leaf nodes). Operators can only be made more-restrictive. Figure 2.2 shows the downgrade hierarchy. In this hierarchy the operators on the top are less restrictive than the ones further below. We again take a top-down approach and visit any $\wedge$, $\vee$ or $\circlearrowleft$-node which is not yet configured. To downgrade these types of nodes we use the following policies:

**Parallel execution $\wedge$:** The $\wedge$-operator can be downgraded to a $\rightarrow$ or $\leftarrow$-operator. Every trace related to the $\wedge$-node should be evaluated whether it executes the children: left-to-right, right-to-left or in another order. If the fraction of traces that execute left-to-right with respect to the total number of traces related to the $\wedge$-node is above a certain threshold we should downgrade to the $\rightarrow$-operator. In a similar fashion the downgrade to the $\leftarrow$-operator is determined. We formally define the downgrade of the $\wedge$-operator as follows:

**Definition 5.2** *(Downgrade parallel execution)*
*Let $n$ be an unconfigured $\wedge$-node, or an $\vee$-node configured as an $\wedge$-operator, of a process tree. Let $L$ be the full event log and $L_n \subseteq L$ the collection of traces related to node $n$. Let $L_{ltr} \subseteq L_n$ be the collection of traces that execute the children of $n$ in a left-to-right way, and $L_{rtl} \subseteq L_n$ be the collection of traces that execute the children of $n$ in a right-to-left way. No trace can be both left-to-right and right-to-left, therefore: $L_{ltr} \cap L_{rtl} = \emptyset$. Let $t_\wedge$ be a given threshold, with $0.5 < t_\wedge \leqslant 1$.*

*We should downgrade to a $\rightarrow$-operator if and only if:*

$$\frac{|L_{ltr}|}{|L_n|} \geqslant t_\wedge \tag{5.2}$$

*We should downgrade to a $\leftarrow$-operator if and only if:*

$$\frac{|L_{rtl}|}{|L_n|} \geqslant t_\wedge \tag{5.3}$$

According to this definition a threshold of $t_\wedge = 0.95$ would indicate that if at least 95% of the traces related to the node are executed left-to-right, we should downgrade to the $\rightarrow$-operator, and if at least 95% of these traces are executed right-to-left, we should downgrade to the $\leftarrow$ operator.

**Choice $\vee$:** The $\vee$-operator can be downgraded to a $\times$ or $\wedge$-operator. Initially we assume that we can downgrade to both of these operators. We should now compare every child of the $\vee$-node to every other child of this node. If the fraction of traces that executes both children with respect to the traces that execute at least one of the two children is at least a certain threshold, we should downgrade to an $\wedge$-operator. If this fraction is below the inverse of this same threshold, we should downgrade to a $\times$-operator. If we should downgrade to an $\wedge$-operator, we should further investigate a downgrade to a $\rightarrow$ or $\leftarrow$-operator as defined by definition 5.2. If the $\vee$-operator is downgraded to an $\wedge$, $\rightarrow$ or $\leftarrow$-operator, all of its children with a block option should get the hide option instead. We formally define the downgrade of the $\vee$-operator as follows:

**Definition 5.3** *(Downgrade choice)*
*Let $n$ be an unconfigured $\vee$-node of a process tree. Let $c(n)$ denote the children of node $n$ and define the pairs of different children as $cp(n) = \{(c_1, c_2) \in c(n) \times c(n) : c_1 \neq c_2\}$. Let $L$ be the full event log and $L_n \subseteq L$ the collection of traces related to node $n$. Let $t_\vee$ be a given threshold, with $0 \leqslant t_\vee \leqslant 1$.*

*We should downgrade to an $\wedge$-operator if and only if:*

$$\forall (c_1, c_2) \in cp(n) : \frac{|L_{c_1} \cap L_{c_2}|}{|L_{c_1} \cup L_{c_2}|} \geqslant t_\vee \tag{5.4}$$

*We should downgrade to a $\times$-operator if and only if:*

$$\forall (c_1, c_2) \in cp(n) : \frac{|L_{c_1} \cap L_{c_2}|}{|L_{c_1} \cup L_{c_2}|} < 1 - t_\vee \tag{5.5}$$

According to this definition a threshold of $t_\vee = 0.95$ would indicate that if 95% of the traces related to the node execute all children, we should downgrade to the $\wedge$-operator, and if less than 5% of the traces execute more than one child we should downgrade to the $\times$-operator.

**Repeated execution $\circlearrowleft$:** The $\circlearrowleft$-operator should be downgraded to the $\rightarrow$-operator if the fraction of traces that execute the *do* and *redo* part of the loop with respect to the traces that only execute the *do* part is below a certain threshold. We formally define the downgrade of the $\circlearrowleft$-operator as follows:

**Definition 5.4** *(Downgrade repeated execution)*
*Let $n$ be an unconfigured $\circlearrowleft$-node of a process tree. Let the do-child of $n$ be denoted by $c_{do}$ and the redo-child by $c_{redo}$. Let $L$ be the full event log and $L_n \subseteq L$ the collection of traces related to node $n$. Let $t_\circlearrowleft$ be a given threshold, with $0 \leqslant t_\circlearrowleft \leqslant 1$.*

*We should downgrade to the $\rightarrow$-operator if and only if:*

$$\frac{|L_{c_{do}} \cap L_{c_{redo}}|}{L_n} < 1 - t_\circlearrowleft \tag{5.6}$$

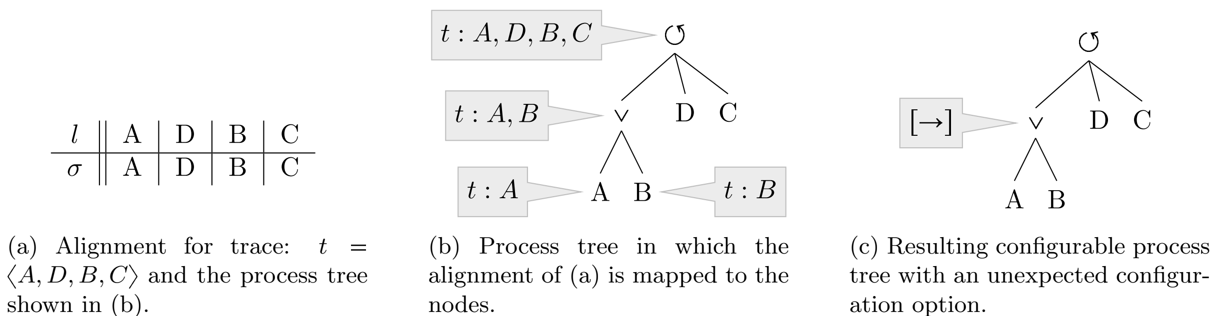
According to this definition a threshold of $t_\circlearrowleft = 0.95$ would indicate that if less than 5% of the traces related to the node execute both the *do* and *redo* child of the loop, we should downgrade to the $\rightarrow$-operator.

After visiting the full process tree we obtain the final configuration. However a problem arises when calculating downgrades for operators that are in the *do* or *redo* part of a $\circlearrowleft$-operator. In the next section we explain the problem in more detail and propose a solution.

### 5.2.4 Loop iteration partitioning

Activities within the *do* or *redo* parts of a $\circlearrowleft$-node may be executed multiple times. From this arises a problem when applying the policies for operator downgrading. Figure 5.7 illustrates this problem. For the trace: $t = \langle A, D, B, C \rangle$ and a process tree we can calculate an alignment (shown in Figure 5.7a). To determine the configurations we need to map this alignment onto the process tree (shown in Figure 5.7b). We should now start our top-down approach to find configurations. Every node is executed by trace $t$, therefore we quickly observe that no hiding or blocking is necessary. The $\circlearrowleft$-node does not require a downgrade since trace $t$ executes both the *do* and *redo* part. The $\vee$-node should however be downgraded. Trace $t$ executes all children of the $\vee$-node, since it is the only trace it means we should downgrade to an $\wedge$-operator. Next is to determine whether we can downgrade to a $\rightarrow$ or $\leftarrow$-operator, since the only trace is $t$, we only have left-to-right traces related to this node and we should therefore downgrade it to a $\rightarrow$-operator. The end-result is shown in Figure 5.7c.

This is obviously not the most favorable result. The newly configured model can no longer replay the original trace $t$ even though this was the only trace. To solve this issue we propose *loop iteration partitioning.* Before processing any child of a $\circlearrowleft$-node we first partition the traces related to the child.

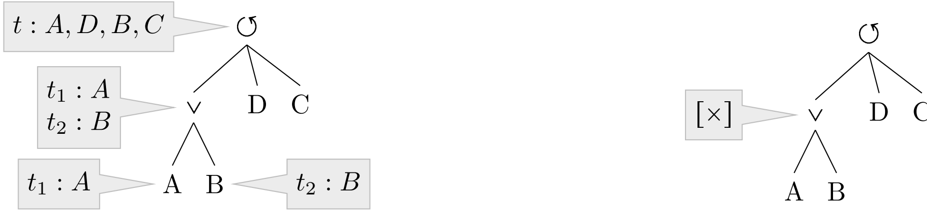



(a) Alignment for trace: $t = \langle A, D, B, C \rangle$ and the process tree shown in (b).

(b) Process tree in which the alignment of (a) is mapped to the nodes.

(c) Resulting configurable process tree with an unexpected configuration option.

Figure 5.7: Mapping of an alignment (a) onto a process tree with a loop construct (b) and how the final configurable tree would look (c). The $\vee$-node is unexpectedly downgraded to a $\rightarrow$-operator.




(a) Process tree in which the alignment of Figure 5.7a is mapped onto the nodes and partitioned for the children of the loop construct.

(b) Resulting configurable process tree with the expected configuration option.

Figure 5.8: The partitioned mapping for the loop construct (a) and the resulting configurable process tree (b).

For every trace we introduce new artificial traces that only contain a single iteration of the loop and we remove the original trace. Figure 5.8a shows how such an updated mapping would look for the original mapping shown in Figure 5.7b. When we now investigate the ∨-node we observe that the two related traces both only execute one of the children. Therefore we should downgrade the ∨-node to a ×-operator. This seems like the most favorable result since downgrading the ∨-node to a ×-operator only increases the overall quality of the process model and it is still able to replay all observed behavior (which was only trace $t$).

## 5.3 Evaluating configurable process trees

Situations exist where we do not have a single reference process tree but instead multiple reference process trees. In the previous section we proposed a new method towards finding a configurable process tree for a collection of event logs by adding configuration options to a single process tree. This method can be applied on each of these reference process trees. In this section we explain a method to evaluate which of the resulting configurable process trees is the best one.

Buijs et al. [12] proposes a method to measure the quality of a configurable process tree in a certain quality dimension. They define the score of a quality dimension for a configurable process tree as the weighted average of the scores for the configured variants of this configurable process tree. This averaged quality dimension is defined as follows:

**Definition 5.5** *(Averaged quality dimension)*
*Let $LC$ denote the collection of input event logs and $PT^c$ the configurable process tree. Let the configured variant of $PT^C$ for an input event log $L \in LC$ be denoted by $PT^c_L$.*

*We now define the score of an quality dimension of $PT^c$ as follows:*

$$Q(PT^c, LC) = \frac{\sum_{L \in LC} |L| \times Q(PT^c_L, L)}{\sum_{L \in LC} |L|} \tag{5.7}$$

Using this definition we can evaluate the configurable process tree on the four well known model quality dimensions: *replay fitness*, *simplicity*, *precision* and *generalization*. These however only take into account the size of the process tree (simplicity), how the process tree reflects the observed behavior (replay fitness and precision) and to what extend it allows for non-observed behavior (generalization). The quality of the configuration options is not considered. The idea is that a configurable process model with fewer configuration options is better. To capture this Buijs et al. [12] proposes a fifth quality dimension: *configuration fitness*, which is defined as follows:

**Definition 5.6** *(Configuration fitness)*
*Let $PT^c$ be a configurable process tree. We define configuration fitness as follows:*

$$Q_c(PT^c) = \frac{\textit{number of configurable nodes in } PT^c}{\textit{number of nodes in } PT^c} \tag{5.8}$$

Using definition 5.5 and 5.6 we can now define an overall quality score for a configurable process tree. We formally define two different ways of calculating this score:

**Definition 5.7** *(Overall configurable process tree quality)*
*Let* $Q_{rf}(PT^c, LC)$, $Q_s(PT^c, LC)$, $Q_p(PT^c, LC)$, *and* $Q_g(PT^c, LC)$ *denote instantiations of* $Q(PT^c, LC)$ *for respectively the model quality dimensions: replay fitness, simplicity, precision, and generalization. Let* $w_{rf}$, $w_s$, $w_p$, *and* $w_g$ *respectively denote weights for the same quality dimensions.*

*We now define two options for calculating an overall configurable process tree quality:*

1. ***Weighted sum (Total):*** *Let* $w_c$ *denote the weight of the configuration fitness quality dimension. We now define the weighted sum of the five quality dimensions as follows:*

$$\begin{aligned} Q_{wt}(PT^c, LC) = {} & w_{rf}Q_{rf}(PT^c, LC) + w_sQ_s(PT^c, LC) \\ & + w_pQ_p(PT^c, LC) + w_gQ_g(PT^c, LC) + w_cQ_c(PT^c) \end{aligned} \tag{5.9}$$

2. ***Weighted sum (Split):*** *Let* $\alpha$, *with* $0 \leqslant \alpha \leqslant 1$, *denote the fraction in which configuration fitness should be taken into account. We now define the weighted sum that separates configuration fitness as follows:*

$$\begin{aligned} Q_{ws}(PT^c, LC) = {} & (1 - \alpha)(w_{rf}Q_{rf}(PT^c, LC) + w_sQ_s(PT^c, LC) \\ & \qquad + w_pQ_p(PT^c, LC) + w_gQ_g(PT^c, LC)) \\ & + \alpha\, Q_c(PT^c) \end{aligned} \tag{5.10}$$

For both options holds that a higher score indicates a better process tree. Therefore the configurable process tree with the highest score is considered to be the best configurable process tree. The first option provides a fully integrated measure of the model and configuration quality dimensions. The second option separates these aspects. Often we first wish to consider the model quality dimensions and only in case of draws prefer a model with a better configuration fitness. We can mimic this behavior by using the 'Weighted sum (Split)' option with a very low value for $\alpha$.

## 5.4 Conclusion

In Chapter 4 we explained how we obtain an annotated hierarchical clustering. From such a hierarchy the end-user can pick one or more groups of traces (clusters). Every group of traces can be considered an individual event log. In this chapter we discussed four different approaches in finding a configurable process tree for a collection of events logs. Evaluation of previous work shows that configuration discovery approaches (approach three and four) overcome the disadvantages of merging approaches (approach one and two). Configuration discovery approaches are implemented in the ETM framework, however in a brute force way. Therefore we have proposed a new configuration discovery approach that can directly be used in approach three and is easily adoptable for approach four. Our configuration discovery approach uses execution frequencies and observed execution orderings to discover a configurable process tree and configurations for every input event log. Finally, in some situations we do not have a single reference process tree but instead multiple reference process trees. We can discover a configurable process tree for each of these reference process trees. It is however not trivial which of these configurable process trees is the best. Therefore we have presented two ways of evaluating which is the best.

# Chapter 6

# ProM implementation

The full approach and many of the discussed techniques are implemented and available as a plug-in of the *ProM 6 framework*[1]. The **Trace Clustering & Configuration Mining** plug-in allows interactive trace clustering with data annotation and configurable process tree discovery. Our plug-in is setup to be easily extendable with new techniques. First in Section 6.1 we provide a user guide. Second in Section 6.2 we explain how the plug-in can be extended.

## 6.1 User guide

Our plug-in provides an interactive platform to perform trace clustering with data annotation, and configurable process tree discovery for a selection of these clusters. As input the plug-in requires a single event log and one or more process trees. The plug-in starts with a wizard consisting of five steps:

1. **Introduction:** This step shows an introductory text which explains the purpose of the plug-in.
2. **General:** Selection of the maximal number of CPU cores and the event classifier. An event classifier defines which of the event data attributes define an event class (i.e., type of activity).
3. **Clustering:** Configuration of the dissimilarity measure, distance aggregation function, and selection and configuration of the clustering algorithm.
4. **Annotation:** This step allows configuration of the annotation algorithm, selection of the data attributes in the event log that may be used for annotation, enable/disable reduction of the hierarchical clustering based on data annotations, and enable/disable of data enrichment with the occurrence of activities.
5. **Evaluation:** In this last step the weight and alpha values are set for the evaluation of the configurable process trees. Our implementation uses the 'Weighted sum (Split)' equation of definition 5.7 to evaluate configurable process trees.

All settings, except the event classifier, can be changed at any time in the plug-in. After finishing the wizard, initial results are calculated and a visualization of the data is shown. Figure 6.1 shows an example for a simulated event log of the running example. The visualization consists of five panes:

1. **Settings:** In this pane the buttons: 'General settings', 'Clustering settings', 'Annotation settings' and 'Evaluation settings' open the corresponding steps of the wizard. All settings except those made in the 'Evaluation settings' are applied immediately and the visualization is updated with the new results. Evaluation is a CPU intensive process and is therefore not performed until the user clicks on 'Evaluate preferred clusters'.
2. **Hierarchy overview:** This pane shows the annotated hierarchical clustering. Every cluster is a circle node in the graph. If a node is colored green it indicates that this cluster is marked as preferred, i.e., it should be considered as an event log in the configurable process discovery. The

[1] ProM 6 is available via: `http://www.promtools.org/prom6/`.

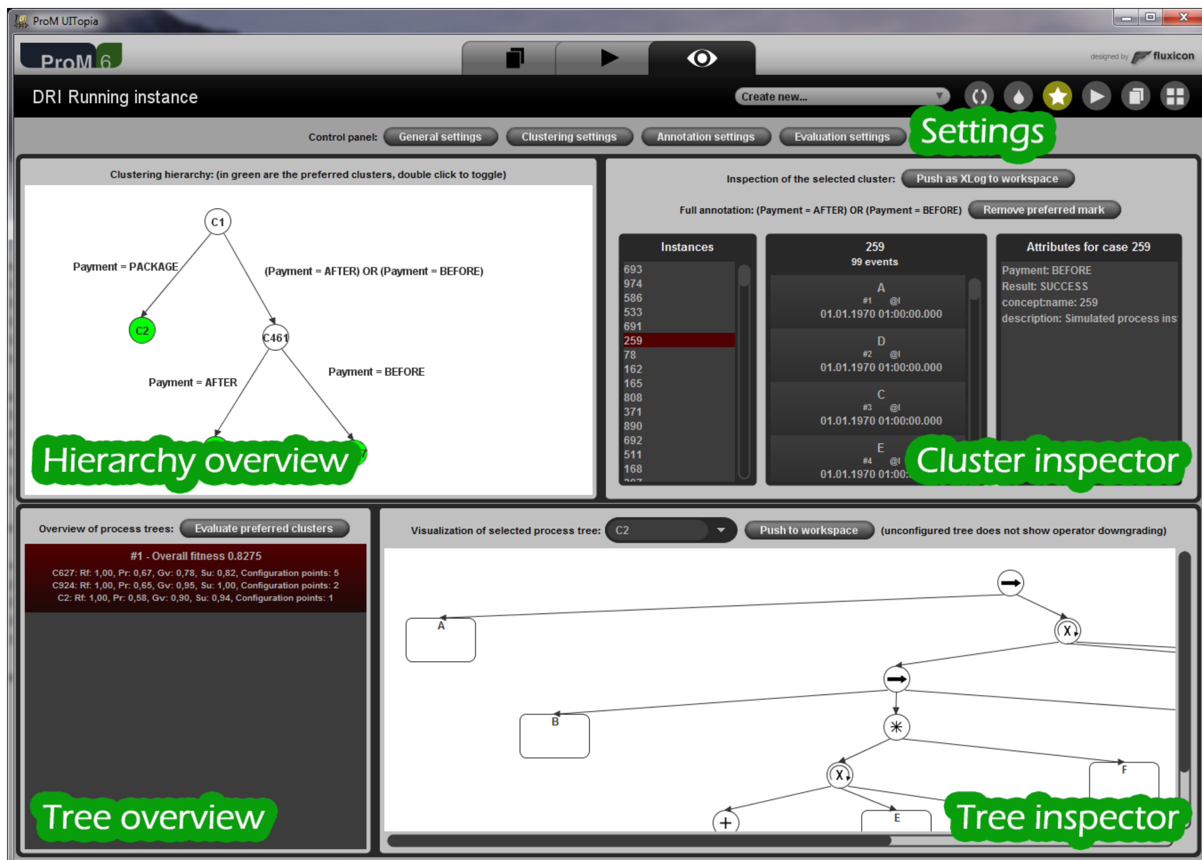



Figure 6.1: Interactive visualization of the plug-in for a simulated log of the running example. The overview consists of five panes: 'Settings', 'Hierarchy overview', 'Cluster inspector', 'Tree overview' and 'Tree inspector'.

preferred status of a cluster can be toggled by double clicking on the corresponding node. A single click on a node shows the contents of the corresponding cluster in the 'Cluster inspector' pane.

3. **Cluster inspector:** After selecting a node in the hierarchical clustering an overview of all traces and corresponding events is provided in this pane. A cluster can be pushed as an `XLog` object to the ProM workspace by clicking on 'Push as XLog to workspace'. `XLog` is the implementation of an event log in the ProM framework.

4. **Tree overview:** After clicking on 'Evaluate preferred clusters' configurations are discovered for the input process trees and the quality characteristics are calculated according to the evaluation settings. This pane shows the configurable process trees in an ordered way (best overall fitness first). The list shows an overall fitness score of the configurable process tree and the scores per preferred cluster. Upon clicking on one of the configurable process trees it is visualized in the 'Tree inspector' pane.

5. **Tree inspector:** This pane shows the selected configurable process tree. With the selection box in the header, the process tree can be configured for each of the preferred clusters. A process tree can be pushed to the ProM workspace by clicking on 'Push to workspace'.

## 6.2 Extendable framework

Our plug-in is setup to be easily extendable with new: dissimilarity measures, clustering algorithms, and annotation algorithms. An extension is added by creating a class which extends one of the base

| *Extension* | *Base class* | *UI Annotation* |
|---|---|---|
| Dissimilarity measure | `AbstractTraceDistanceFunction` | `UIDistanceFunction` |
| Clustering algorithm | `AbstractClusteringAlgorithm` | `UIClusteringAlgorithm` |
| Annotation algorithm | `AbstractAnnotatorAlgorithm` | `UIAnnotatorAlgorithm` |

Table 6.1: Overview of the possible extensions, their corresponding base class, and the annotation that should be added to enable the extension in the GUI.

classes. To enable an extension in the GUI an additional annotation should be added. Table 6.1 shows the possible extensions with the corresponding base class and UI annotation. The GUI automatically searches the root package of the plug-in for extensions. To enable an extension in the GUI it should therefore be placed in a subpackage of `org.processmining.plugins.yvo`. Every UI annotation contains a list of `UISetting` annotations which allow the developer to define the settings that may be set in the GUI. We now explain each of the extensions in more detail:

**Dissimilarity measure:** To add a new dissimilarity measure the developer should create a new class that extends `AbstractTraceDistanceFunction`. This abstract class enforces the implementation of the method `distance(t1, t2)`, which should calculate the distance between two traces. Please note that distances are calculated in parallel and therefore any dissimilarity measure is required to be thread safe. The framework also assumes that every dissimilarity measure is symmetric.

Instead of using `AbstractTraceDistanceFunction`, feature-set approaches are encouraged to extend the class `AbstractFeatureSetTraceDistanceFunction`. This class implements the distance function as the Euclidean distance of two feature-sets. When extending this class the developer is only required to implement the method `createFeatures(t1)`, which should return the features for the given trace.

**Clustering algorithm:** In order to add a new clustering algorithm, a new class should be created which extends `AbstractClusteringAlgorithm`. This class enforces the implementation of a method `clusterHierarchical()`, which should perform the actual clustering and return a hierarchy. The method `clusterHierarchical()` does not have any parameters but instead has access to a `Central Repository` object which provides the event log and trace distances.

One of the most well known open-source data mining frameworks is ELKI[2]. ELKI contains implementations for most of the well known clustering algorithms. To allow quick use of ELKI the developer can instead extend from `AbstractElkiClusteringAlgorithm` which provides basic wrapping functionality.

**Annotation algorithm:** A new annotation algorithm is added by extending the class `AbstractAnnotatorAlgorithm`. This abstract class enforces the implementation of the method `annotate(clustering)`, which should add annotation expressions to a given collection of clusters.

Weka is another well known open-source data mining framework[3]. Weka has implementations for most of the well known classification algorithms. To allow quick use of Weka the developer can instead extend from `AbstractWekaAnnotatorAlgorithm` which provides basic wrapping functionality.

Every extension has access to a `CentralRepository` object. This object acts as a central storage for information used in the plug-in, i.e., it stores the configuration, the event log and basic information of this event log, intermediate results of the different steps, and the process tree contexts. The configuration is a key-value store in which string keys map to objects of arbitrary types, e.g., the number of usable CPU cores is represented by the key `general.cpu.cores` and an integer value. The process tree contexts are represented by `ModelRepository` objects which store a process tree, the (sequential) alignments and the partially ordered alignments.

[2]The website of ELKI is: `http://elki.dbs.ifi.lmu.de/`.
[3]The website of Weka is: `http://www.cs.waikato.ac.nz/ml/weka/`.

# Chapter 7

# Experimental evaluation

In this chapter we present and discuss the results of executed experiments which test the effectiveness of our newly proposed approaches and the effectiveness of our full approach. Most of the experiments are executed on randomized data since event logs with a known clustering are rare. First in Section 7.1 we explain the creation of random process trees and event logs that contain behaviorally similar groups of traces. Section 7.2 compares the Syntactic reasoning approach with existing dissimilarity measures for trace clustering. Section 7.3 compares our configuration discovery approach with existing approaches. Finally in Section 7.4, we apply our full approach on various data sets.

## 7.1 Generating process trees and clustered event logs

For our experimental setup we implemented a random process tree generator and a clustered event log simulator. The random process tree generator requires the following parameters: minimum nodes, maximum nodes, the range of the number of children for an operator node, the number of event classes (i.e., activity types), and probabilities for: a $\tau$-activity, a leaf and probabilities for the operator types. Using these parameters the generator returns fully random process trees that satisfy these conditions.

Given a process tree the clustered event log simulator generates an event log that contains groups of traces which are behaviorally similar. The parameters of the simulator are: the number of traces in the event log, the number of required clusters, the noise probability, and the allowed configuration types. To generate the event log the simulator first generates a configuration (for the process tree) for every cluster that is required. In order to obtain these configurations the simulator first determines configuration possibilities per node in the tree. Three types of configuration possibilities are distinguished:

1. **Choice:** At certain points in the process tree traces can make a choice, i.e., between the children of a $\times$ or $\vee$-operator. At these points the simulator creates a configuration possibility for every child of the $\times$ or $\vee$-operator wherein the active child is allowed and all other children are blocked.

2. **Order:** Process trees do not always imply a strict ordering on activities, i.e., between the children of an $\vee$ or $\wedge$-operator. In this case the simulator creates two possibilities, either downgrade the $\vee$ or $\wedge$-node to a $\rightarrow$-operator or to a $\leftarrow$-operator.

3. **Loop:** At every $\circlearrowleft$-node a decision between just a single iteration or multiple iterations can be made. Again the simulator creates two possibilities, either downgrade to a $\rightarrow$-operator or not downgrade at all.

Only if a type is allowed the corresponding possibilities are added to the possibilities of a given node. In the second step the simulator tries to obtain an initial set of configurations. The simulator does this by randomly taking a node and the corresponding configuration possibilities. First we investigate whether the current node is always reachable. For example, if one of its parents is a choice, and the branch of the current node was not chosen, this node can never be reached. Introducing configurations for this node would then not make sense. Every configuration possibility can become an individual configuration. If we have less configuration possibilities than configurations that are still required we can immediately add the configuration possibilities to the set of configurations. However if only one more

configuration is required, we randomly pick two of the possibilities and combine these with a randomly picked configuration which we previously obtained. Such a combination is however only possible if the existing configuration does not block the path to the new possibilities.

We have now obtained an initial collection of configurations. The simulator now tries to randomly expand this initial collection with new configuration possibilities. This might introduce so-called long-term dependencies, i.e., a choice in the beginning of the execution may influence a choice or the order of execution at the end of the execution. We do this by combining a collection of existing configurations with a collection of possibilities. Every existing configuration is combined with one of the possibilities. This combination is however only possible if the existing configuration does not block the path to the new possibility.

Finally, we obtain a configurable process tree with configurations for every required cluster. The next step is to make a division of the required traces among the required clusters. We do this by first dividing it in a fair way. Second, we introduce some random deviation. Next is to configure the process tree per required cluster and randomly execute the configured process tree from the initial state to an arbitrary final state. Every of such executions is considered a trace. We keep randomly executing the configured process tree until we have found the required number of traces for the cluster we are currently creating. Finally this yields: a combined event log, event logs per cluster, and a process tree with configurations per cluster.

## 7.2 Comparison of dissimilarity measures

In Chapter 3 we discussed trace clustering and the importance of a dissimilarity measure. Many approaches exist towards defining such a measure. We explained the problems of existing approaches and proposed a new approach in Section 3.3, i.e., the Syntactic reasoning approach. First in Section 7.2.1 we explain our experimental setup. Second in Section 7.2.2 we present and discuss the results.

### 7.2.1 Experimental setup

The Syntactic reasoning approach is a syntactical approach that tries to overcome the common problems of existing approaches by adding model-awareness. If not mentioned otherwise we use the default cost settings of the Syntactic reasoning approach[1]. To show this advantage we compare our approach with four existing approaches:

1. **Levenshtein distance:** A fundamental way to calculate the dissimilarity between two sequences. The Levenshtein distance counts the number of insertions, deletions or substitutions required to change one sequence into another.
2. **Set-Of-Activities** Every activity observed in the full event log is considered to be a feature (or dimension in a vector). If the trace contains an activity, the feature is given the value 1 and if not 0. The final dissimilarity of two traces is defined as the Euclidean distance of the vectors of features of these traces.
3. **Bag-Of-Activities:** Similar to the 'Set-Of-Activities' approach, however different in that a feature is given the number of occurrences of an activity as the value.
4. **3-Grams:** The k-grams approach with $k = 3$. Every subsequence of a trace of length 3 is considered to be a feature. The number of times this subsequence occurs in the trace is the value of the feature. The final dissimilarity of two traces is defined as the Euclidean distance of the vectors of features of these traces.

We evaluate these approaches on randomly generated process trees and clustered event logs. Which are generated using the techniques as explained in Section 7.1.

As explained in Section 3.3 existing feature-set approaches have trouble capturing the order of execution, while existing syntactical approaches suffer from issues with respect to loop identification and that

[1]The default costs of the Syntactic reasoning approach are: $p_{pd} = 1.25$, $p_{lms} = 0.5$, $p_{lru} = 1.25$, $p_{nt} = 0.75$, $p_{id} = 1.0$, $p_{c1} = 1.5$, $p_{c2} = 0.5$ and $p_{c3} = 1.0$.

| *Case* | *Leaf* | *Operator* | $\tau$-leaf | $\rightarrow$ | $\leftarrow$ | $\wedge$ | $\circlearrowleft$ | $\times$ | $\vee$ |
|---|---|---|---|---|---|---|---|---|---|
| Normal | 0.50 | 0.50 | 0.05 | 0.25 | 0.05 | 0.25 | 0.10 | 0.200 | 0.150 |
| Loop iterations | 0.50 | 0.50 | 0.05 | 0.05 | 0.05 | 0.05 | 0.70 | 0.075 | 0.075 |
| Order of execution | 0.50 | 0.50 | 0.05 | 0.05 | 0.05 | 0.70 | 0.05 | 0.075 | 0.075 |
| Parallelism punishment | 0.50 | 0.50 | 0.05 | 0.05 | 0.05 | 0.70 | 0.05 | 0.075 | 0.075 |

Table 7.1: Per case the probability of an operator or leaf node. Followed by the probability of an $\tau$-leaf and the probability of each operator type. Any leaf that is not a $\tau$ describes an actual activity.

| *Case* | *Tree Size* | *Operator node children #* | *Duplicate activities* | *Event classes #* |
|---|---|---|---|---|
| Normal | *Variable* | 2 - 5 | No | *Variable* |
| Loop iterations | *Variable* | 2 - 5 | No | *Variable* |
| Order of execution | *Variable* | 2 - 5 | No | *Variable* |
| Parallelism punishment | *Variable* | 2 - 5 | No | *Variable* |

Table 7.2: Per case the tree size, number of children per operator node, whether duplicate activities are allowed, and the number of event classes.

mismatches in the order of execution might be punished very heavily where this would not be appropriate. To illustrate these problems we introduce four cases, one normal case and three cases that illustrate extreme case wherein these problems are visible. Table 7.1 shows per case the probability settings and Table 7.2 shows the other settings of the random process tree generator. Table 7.3 shows the settings of the clustered event log simulator. The four cases are setup as follows:

**Normal:** We compare our approach against others on normal trees. We chose probabilities that we think are close to realistic process trees. The clustered event log generator is allowed to use any kind of configuration possibility.

We hypothesize that if our approach adequately deals with common problems it should profit from this in the normal situation as well and therefore perform better than other dissimilarity measures in general.

**Loop iterations:** In this case the probability of a $\circlearrowleft$-node is very big, this can cause a great variability in the traces of the simulated event logs. Some traces iterate loops very often while others very few, even while the traces belong to the same cluster. In this case we wish to highlight problems that occur if an approach fails to deal with loops. We include a context-aware version of the Syntactic reasoning approach which is setup to ignore the cost of additional loop iterations ($p_{lru} = 0$).

We hypothesize that approaches that are capable of dealing with repeating activities should perform better. Examples of such approaches are the Syntactic reasoning approach and the Set-Of-Activities approach. When setting the cost of additional loop iterations in the Syntactic reasoning approach to zero, both approaches do not penalize an extra loop iteration and are therefore able to accurately identify the different choices that are made.

**Order of execution:** In this case the probability of an $\wedge$-node is very big, therefore process trees have a lot of parallelism. In this case we only allow for the order configuration type. By doing this we can show that feature-set approaches have trouble capturing the order of execution aspect. We include a context-aware version of the Syntactic reasoning approach which penalizes the cost of a difference in parallel execution very heavily ($p_{pd} = 20$).

| *Case* | *Configuration types* | *Trace #* | *Cluster #* | *Noise probability* |
|---|---|---|---|---|
| Normal | Choice, Order, Loop | 1,000 | *Variable* | 0.0 |
| Loop iterations | Choice | 1,000 | *Variable* | 0.0 |
| Order of execution | Order | 1,000 | *Variable* | 0.0 |
| Parallelism punishment | Choice | 1,000 | *Variable* | 0.0 |

Table 7.3: Per case the allowed configuration types, the number of traces per experiment, the number of clusters per event, and the probability of noise.

Syntactical approaches can fully observe the order of execution aspect while the 3-Grams approach can only partially observe this. Other feature-set approaches like the Set-Of-Activities and Bag-Of-Activities can not observe this at all. We therefore hypothesize that syntactical approaches perform best, 3-Grams worse and the Set-Of-Activities and Bag-Of-Activities approaches worst.

**Parallelism punishment:** Like the previous case, the probability of an $\wedge$-node is very big, therefore process trees have a lot of parallelism. In this case we only allow for the choice configuration type. This means that the traces can execute the process trees in various orders, even while they belong to the same cluster. In this case we include a context-aware version of the Syntactic reasoning approach which is setup to ignore the cost of the parallel execution difference ($p_{pd} = 0$).

We hypothesize that approaches that cannot ignore parallelism behavior have difficulties finding the clusters. Examples of such approaches are the Levenshtein distance and 3-Grams.

Each of these cases is investigated for different tree sizes. We investigate the process tree sizes: 2 - 5, 5 - 10, 10 - 20, 20 - 40, 40 - 80 and 80 - 160 nodes, with respectively the number of event classes: 5, 10, 20, 40, 80, 160 and the number of clusters in the log: 2, 2, 3, 4, 5 and 5. The clustering problem becomes increasingly difficult with an increasing process tree size and number of clusters. For every process tree size we run 100 experiments, i.e., 100 different process trees and clustered event logs are generated on which we perform single linkage hierarchical clustering [16] using the five dissimilarity measures as distance functions.

The result for a single experiment and a dissimilarity measure is a hierarchical clustering. From our clustered event log simulator we however get a flat clustering of traces. Comparing the hierarchical clustering with the flat reference clustering is not trivial. To extract a flat clustering from the hierarchy we *cut* the hierarchy at the required number of clusters. We do this by starting at the root and traversing the most distant splits until we have found the required number of clusters.

It may however occur that a draw occurs, i.e., a split consists of more than two clusters or multiple splits have the same distance between the clusters. If a split consists of more than two clusters we should add all of these clusters at the same time. If we have multiple splits we should traverse all of these splits at the same time. It may now occur that we obtain more clusters than we required. To solve this we should take the *base set* of clusters (i.e., the clusters we had before traversing the splits without the splits we are about to traverse) and the *candidate set* of clusters (i.e., clusters we obtain after traversing the splits). The base set should now be combined with every combination of the candidate clusters of such length that we obtain the required number of clusters. Every such combination of the base and candidates is called a cut of the hierarchy.

Another issue arises when the hierarchy does not contain the required number of clusters. In this case we should add empty clusters to the final set of clusters such that we obtain the required number of clusters. This final clustering is called a cut of the hierarchy.

For every cut of the hierarchy we calculate the *precision*, *recall* and *$F_1$ score*, which is the harmonic mean of recall and precision. In our results we distinguish between the *best cut* (i.e., the cut for which the $F_1$ score is highest) and the *worst cut* (i.e., the cut for which the $F_1$ score is lowest). We formally define the recall, precision and $F_1$ scores for a single cut of the hierarchy as follows:

**Definition 7.1** *(Recall, Precision and the $F_1$-score)*
*Let $C_{ref} = <C_{r1}, ..., C_{rn}>$ denote a list of reference clusters. Let $C_{cut}$ be a cut of the hierarchical clustering with $|C_{cut}| = |C_{ref}|$. Let $pmt(S)$ be a function that returns all permutations for a set $S$. Let $<C_{p1}, ..., C_{pn}>$ be a single permutation of $C_{cut}$. We now define precision for a single permutation as follows:*

$$P_{pmt}(<C_{r1}, ..., C_{rn}>, <C_{p1}, ..., C_{pn}>) = \frac{|C_{p1} \cap C_{r1}| + ... + |C_{pn} \cap C_{rn}|}{|C_{p1}| + ... + |C_{pn}|} \tag{7.1}$$

*We now define recall for a single permutation as follows:*

$$R_{pmt}(<C_{r1}, ..., C_{rn}>, <C_{p1}, ..., C_{pn}>) = \frac{|C_{p1} \cap C_{r1}| + ... + |C_{pn} \cap C_{rn}|}{|C_{r1}| + ... + |C_{rn}|} \tag{7.2}$$

*Using these measures we define the $F_1$-score for the given cut of the hierarchy as follows:*

$$F_1(C_{ref}, C_{cut}) = \max_{C_{pmt} \in pmt(C_{cut})} \frac{2P_{pmt}(C_{ref}, C_{pmt})R_{pmt}(C_{ref}, C_{pmt})}{P_{pmt}(C_{ref}, C_{pmt}) + R_{pmt}(C_{ref}, C_{pmt})} \tag{7.3}$$

*Let $C_{pmt,max}$ be the permutation of $C_{cut}$ for which the $F_1$-score is maximized. We now define precision as follows:*

$$P(C_{ref}, C_{cut}) = P_{pmt}(C_{ref}, C_{pmt,max}) \tag{7.4}$$

*We now define recall as follows:*

$$R(C_{ref}, C_{cut}) = R_{pmt}(C_{ref}, C_{pmt,max}) \tag{7.5}$$

### 7.2.2 Results and discussion

Figure 7.1, Figure 7.2, Figure 7.3 and Figure 7.4 respectively show the results of the cases: normal, loop iterations, order of execution and parallelism punishment. In each of these figures the left graph shows the $F_1$ score of the worst cut and the right graph shows the $F_1$ score of the best cut. Appendix B.1 includes the corresponding recall and precision scores for each of the cases. Please note that the context-specific version of the Syntactic reasoning approach has different tweaks of the cost function per case. For all cases we observe that for very small process trees (2 - 5 and 5 - 10 nodes) every approach performs very good and often provides a perfect clustering. In these process trees the clustered event log simulator is unable to build more complex configurations. This results in very simple and clearly distinguishable clusters, e.g., a choice between `A` and `B`. With larger process trees, starting from 10 - 20 nodes, the clustered event log simulator is able to generate more complex configurations, and therefore clusters that are harder to recognize. For most cases, except the loop iterations, we observe that at 10 - 20 nodes the results start to stabilize. We now discuss every case individually:

**Normal:** (Figure 7.1) For normal process trees we observe that the Syntactic reasoning approach is one of the top scoring approaches. We observe a relatively stable result, i.e., there is not much difference between picking the worst or the best cut. The Set-Of-Activities has a similar score but is less stable, i.e., there is a bigger difference between picking the worst or the best cut.

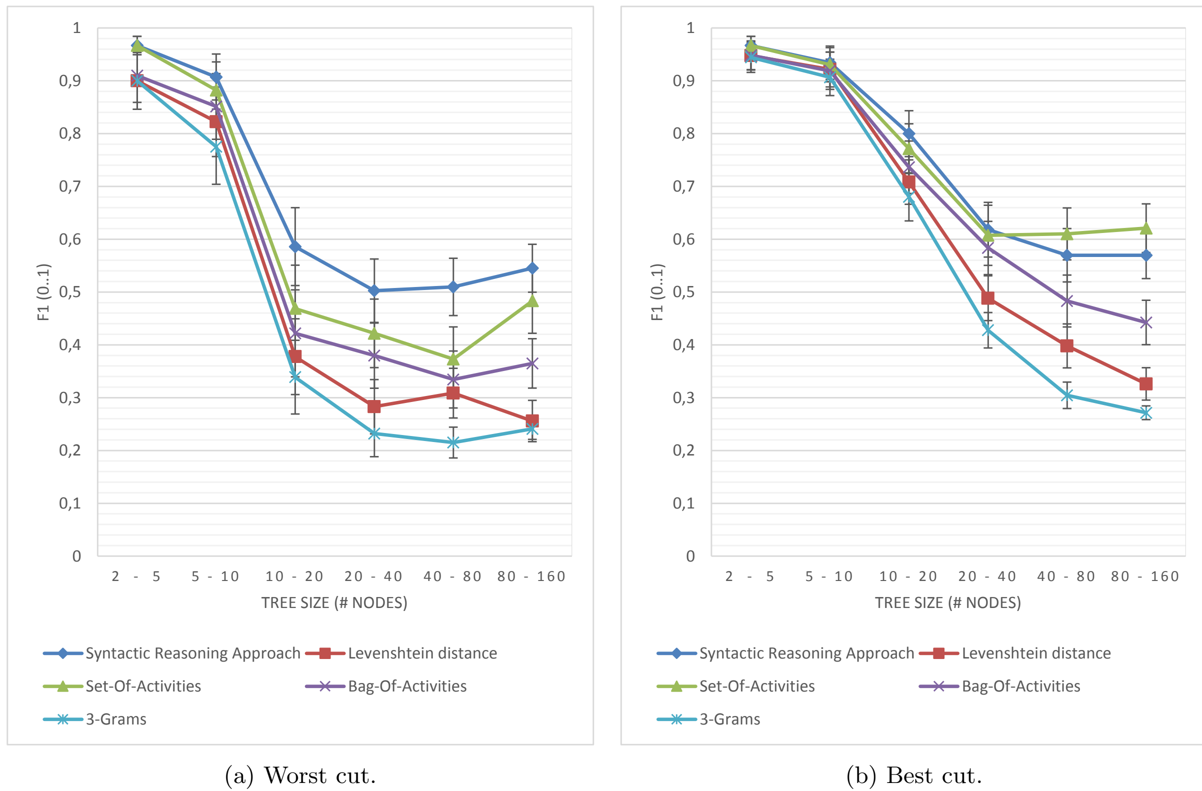
We should also take into account that the random process tree generator was not allowed to use duplicate activities which is an advantage for the Set-Of-Activities approach. When duplicate activities are allowed, clusters may be formed on the execution of the same activity but at a different state in the process tree. By providing a process tree to the Syntactic reasoning approach it can distinguish these different states. For the Set-Of-Activities it is however impossible to identify these kinds of clusters.

**Loop iterations:** (Figure 7.2) We observe that the Syntactic reasoning approach can adequately deal with loops by using the input process tree. The Set-Of-Activities approach was able to deal with loops since it only counts activities once, meaning that adding another iteration of a loop adds no distance. The Levenshtein distance, 3-Grams and Bag-Of-Activities were not able to deal with loops and therefore score much worse.

With a context-specific tweak of the cost parameters we greatly improved the performance of the Syntactic reasoning approach such that it can compete with the Set-Of-Activities approach. Like in the normal case we observe that the Set-Of-Activities approach is less stable than the Syntactic reasoning approach.

**Order of execution:** (Figure 7.3) In this experiment we observe the advantage of a syntactical approach over a basic feature-set approach. Because of the syntactic nature of our approach it fully captures the order of execution aspect. The Syntactic reasoning approach scores similar to the Levenshtein distance and the 3-Grams approach. 3-Grams is a feature-set approach that partially observes the order of execution aspect, this seems to be sufficient to score similarly to the syntactical approaches in this experiment. The Set-Of-Activities and Bag-Of-Activities approaches are unable to distinguish any order of execution and therefore score worst. Again we observe that a context-specific tweak of the cost parameters of the Syntactic reasoning approach improves the result. In this experiment this is however less visible.

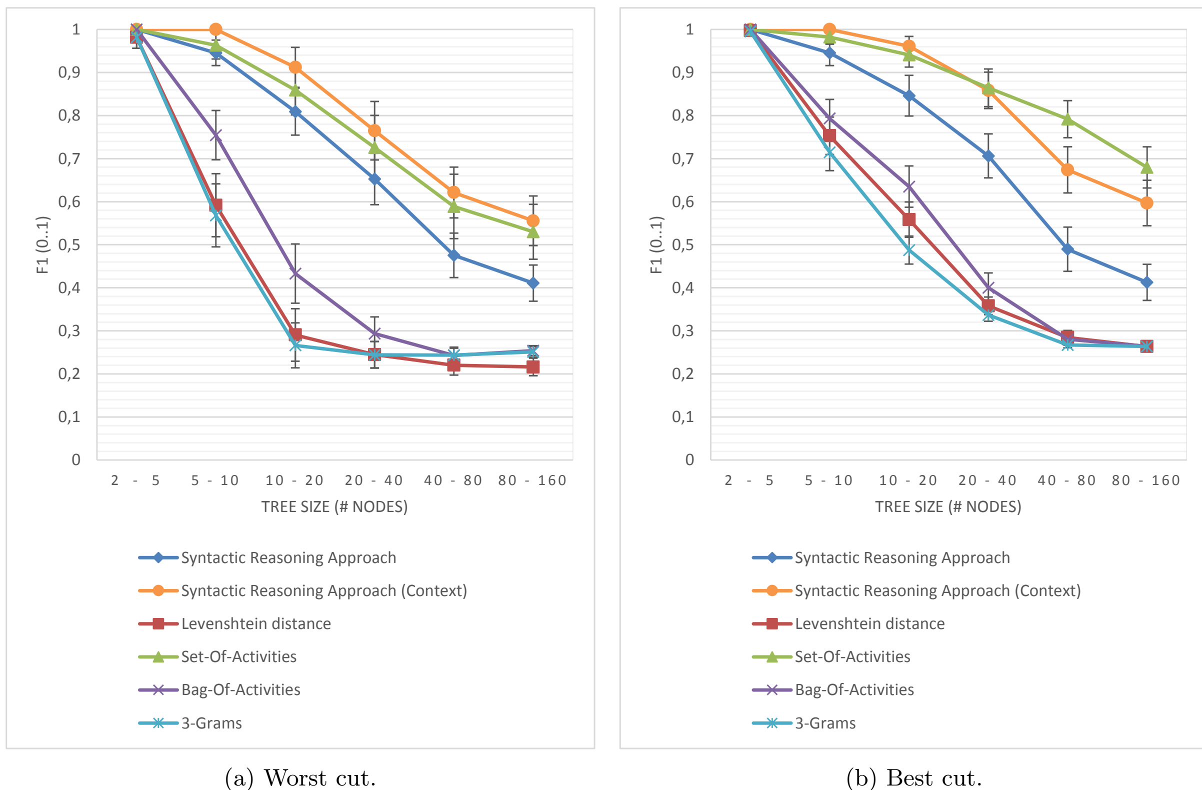
In general it seems that all of the approaches have difficulties recognizing the clusters in this experiment. As the process trees become bigger more nested $\wedge$ or $\vee$-nodes occur. In some cases the clustered event log simulator configures the shallow nodes but not the deeper nodes. Because




(a) Worst cut.

(b) Best cut.

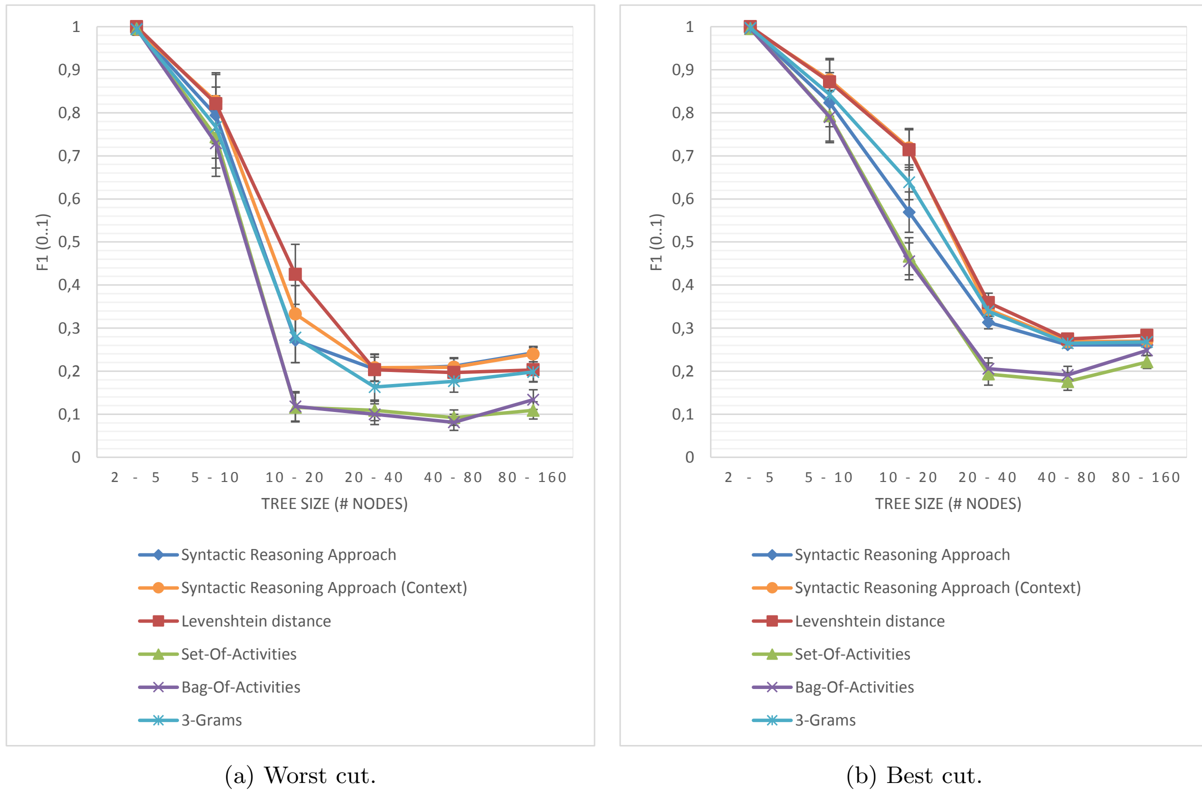
Figure 7.1: The F1 scores with 95% confidence intervals of the best and worst cut in the normal case.




(a) Worst cut.

(b) Best cut.

Figure 7.2: The F1 scores with 95% confidence intervals of the best and worst cut in the loop iterations case.




(a) Worst cut.

(b) Best cut.

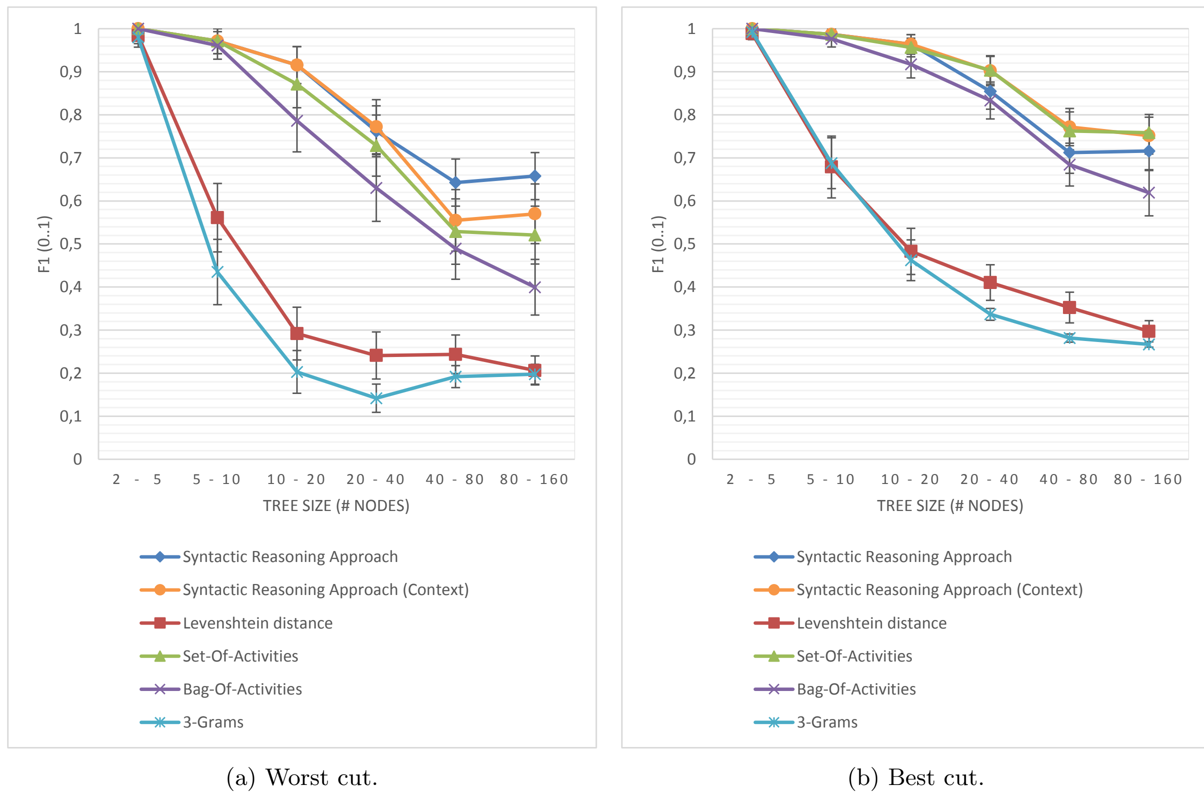
Figure 7.3: The F1 scores with 95% confidence intervals of the best and worst cut in the order of execution case.

of this still much variability in the order of execution among the clusters may exist. Meaning that it is still very difficult to observe the clustering.

**Parallelism punishment:** (Figure 7.4) The Syntactic reasoning approach is again one of the top scoring approaches and provides the most stable result. The Set-Of-Activities and Bag-Of-Activities do not observe order of execution and therefore score similarly to the Syntactic reasoning approach. The context-specific tweak of the cost parameters does improve the result in the best cut, but decreases the result in the worst cut, therefore the context-specific variant is less stable. The Levenshtein distance and 3-Grams approach penalize difference in the order of execution very heavily and therefore score worst.

We observe that the best approaches differ per case. However our approach, the Syntactic reasoning approach, with a simple context-specific tweak always scores among these best approaches. Without any tweaks our approach already scores among the best scoring approaches for the normal, order of execution and parallelism punishment cases. The Syntactic reasoning approach supports context-specific tweaking of the cost function. We have shown that a simple tweak can greatly improve the performance of our approach. With such a tweak we improved the scores in the loop iterations approach such that it can compete with the top scoring approach of this case, i.e., Set-Of-Activities. Furthermore, it seems that the Syntactic reasoning approach produces a more stable result than other approaches, i.e., there is not much difference between the best and worst cut of the resulting hierarchical clustering.

Our approach adequately deals with loops and difference in parallelism execution but is still able to penalize differences in order of execution when required in a context. Therefore our approach combines the best of two worlds (feature-set and syntactical). Opposed to the compared approaches, the Syntactic reasoning approach allows tweaking of the cost function to a specific context which can greatly improve performance.




(a) Worst cut.

(b) Best cut.

Figure 7.4: The F1 scores with 95% confidence intervals of the best and worst cut in the parallelism punishment case.

## 7.3 Comparison of configurable process discovery approaches

In Chapter 5 we discussed different approaches towards configurable process discovery. We discussed the ETMc algorithm which is part of the ETM framework. The ETMc algorithm explores the full search space to find the optimal set of configurations, it is therefore a brute force approach [12]. We have proposed a new structured approach that uses execution frequencies and observed execution orderings to discover configurations in Section 5.2. This section compares our method with the ETMc algorithm. First in Section 7.3.1 we explain the experimental setup. Second in Section 7.3.2 we present and discuss the results.

### 7.3.1 Experimental setup

In this experiment we compare our configuration discovery approach with the ETMc algorithm. The ETMc algorithm is currently the only other configuration discovery approach for process trees. We split our experiment into two parts:

**Comparison with paper:** Buijs et al. [12] evaluated the ETMc algorithm on a running example. We compare their results of approach three with the results of our algorithm for the same event logs and input process tree. They ran the ETMc algorithm for 80,000 generations to obtain a stable result. The used running example is a simple loan application process of a financial institute. Four different variants of this process exist for which they created event logs using simulation. We first discuss the *time complexity* of both algorithms and second compare both algorithms in the following quality dimensions: *the number of configuration options*, *generalization*, *precision*, *replay fitness*, and *simplicity*.

**Random trees and clustered event logs:** In the second part of this experiment we randomly generate process trees and clustered event logs using the techniques as discussed in Section 7.1. The random tree generator uses the same settings as the normal case of the previous experiment (shown

in Table 7.1 and Table 7.2). We investigate the tree sizes: 2 - 5, 5 - 10, 10 - 20, and 20 - 40 with respectively the number of event classes: 5, 10, 20 and 40. For every tree size 30 different process trees and clustered event logs are generated. The clustered event log simulator generates logs of 500 traces without noise. It is allowed to use all types of configuration possibilities and is setup to generate event logs with two clusters. The clustered event log simulator internally uses a configurable process tree to simulate the event log. We consider this *reference configurable process tree* to be the optimal result. In this experiment we consider the *time aspect* and the following quality dimensions: *replay fitness*, *precision*, *simplicity*, *generalization* and *the number of configuration options*.

To evaluate the ETMc algorithm we run it for 1,000 generations with a population-size of 20 and an elite-size of 6. After 1,000 generations we evaluate the best configurable process tree from the population. The ETMc is setup to evaluate the process trees using the 'Weighted sum (Split)' equation of definition 5.7, with parameters: $w_{rf} = 10$, $w_p = 5$, $w_s = 1$, $w_g = 1$, and $\alpha = 0.00001$.

All process model quality dimensions are calculated according to definition 5.5. In the results we denote our approach as the VisFreq algorithm. In all experiments we run our approach using 95% threshold settings, i.e., $t_{hb} = 0.95$, $t_{\wedge} = 0.95$, $t_{\vee} = 0.95$ and $t_{\circlearrowleft} = 0.95$.

### 7.3.2 Results and discussion

**Comparison with paper**

We start by comparing our approach with the paper by Buijs et al. [12]. Figure 7.5 shows the resulting configurable process trees of the ETMc and the VisFreq algorithm. In order to obtain a stable result the ETMc was ran for 80,000 generations. In every generation the ETMc needs to evaluate every newly created or modified process tree using the techniques as discussed in Section 5.3. In order to calculate replay fitness, the ETMc calculates alignments between a process tree and every unique trace in the event log. The time complexity of a random mutation of the configuration options in a process tree is negligible with respect to the time complexity of calculating alignments. Therefore most of the time is spend on calculating alignments. The same holds for our algorithm, determining the configuration options is negligible with respect to the time complexity of calculating alignments. However our approach calculates the alignments only once for the input process tree. Therefore the time complexity of our algorithm is approximately the time complexity of a single process tree evaluation of the ETMc. We further support this claim in the second part of this experiment.

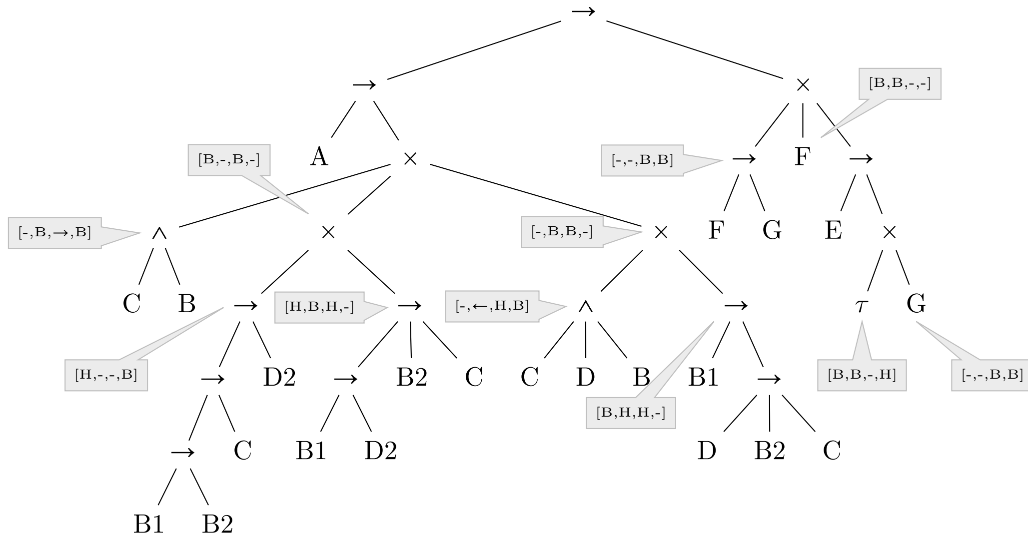
When manually investigating the configurable process trees (shown in Figure 7.5) we observe that the configuration options for both algorithms are very similar. However since the ETMc randomly adds configuration options to nodes in the process tree, the resulting configurable process tree contains configuration options for nodes that are no longer reachable since one of its parents is hidden or blocked for that configuration.

Table 7.4 shows the combined scores of the quality dimensions and the scores per event log and corresponding configuration. The best scores among the two approaches are in bold. We now discuss every quality dimension individually:

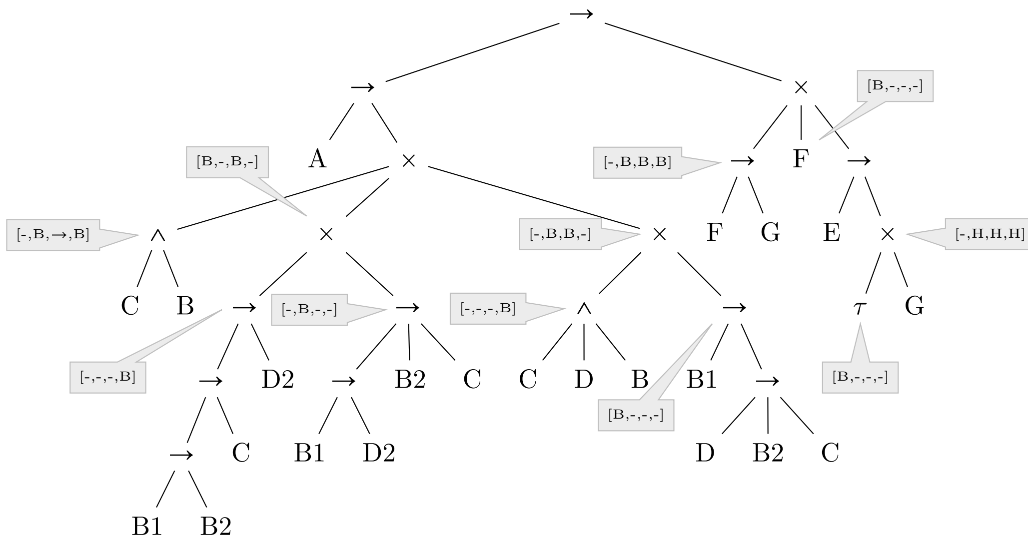
**Configuration options:** For every variant the VisFreq algorithm generates fewer configuration options than the ETMc algorithm. The ETMc algorithm randomly configures nodes in the process tree. Because of this the ETMc might add a configuration option to a node after which in further generations it hides or blocks one of the parents of this node. The configuration option is not removed from this child node, causing the final result to have redundant configuration options. The VisFreq algorithm investigates the configuration options top-down and therefore does not suffer from this issue.

**Generalization:** In the generalization dimension our approach scores slightly worse. We argue that generalization is the least important quality dimension for any configured process tree. A configurable process tree should generalize very well since it should support many variants. However upon configuring a configurable process tree we want to restrict the process tree such that it only supports the preferred variant. By restricting a process tree it should generalize less.

**Precision:** Both algorithms score identical in the precision dimension.




(a) Result of the ETMc algorithm.




(b) Result of the VisFreq algorithm.

Figure 7.5: Configurable process tree discovered by the ETMc algorithm (a) and the configurable process tree discovered by the VisFreq algorithm (b).

| | *Configuration options* | *Generalization* | *Precision* | *Replay Fitness* | *Simplicity* |
|---|---|---|---|---|---|
| Combined | 6.9895 | **0.6591** | **0.9775** | 0.9933 | **0.7626** |
| Variant 0 | 6.0000 | **0.7112** | **0.9898** | **1.0000** | **0.8500** |
| Variant 1 | 7.0000 | 0.5157 | **1.0000** | 0.9545 | **0.7000** |
| Variant 2 | 8.0000 | **0.7069** | **1.0000** | **1.0000** | **0.7692** |
| Variant 3 | 6.0000 | **0.6142** | **0.9079** | **1.0000** | **0.7083** |

(a) Results ETMc algorithm.

| | *Configuration options* | *Generalization* | *Precision* | *Replay Fitness* | *Simplicity* |
|---|---|---|---|---|---|
| Combined | **4.7895** | 0.6557 | **0.9775** | **1.0000** | 0.7439 |
| Variant 0 | **4.0000** | **0.7112** | **0.9898** | **1.0000** | **0.8500** |
| Variant 1 | **5.0000** | **0.5562** | **1.0000** | **1.0000** | 0.6471 |
| Variant 2 | **5.0000** | 0.6901 | **1.0000** | **1.0000** | 0.7500 |
| Variant 3 | **5.0000** | 0.6039 | **0.9079** | **1.0000** | 0.6957 |

(b) Results VisFreq algorithm.

Table 7.4: Quality criteria of the two trees shown in Figure 7.5. The best scores among the two approaches are in bold. Fewer configuration options is considered to be better. For the other quality dimensions holds that a higher score is considered to be better.

**Replay fitness:** The VisFreq algorithm performs slightly better with respect to replay fitness. The VisFreq algorithm finds a configuration for variant one that supports perfect replay fitness while the ETMc did not. Therefore the combined score is higher as well.

**Simplicity:** The ETMc algorithm scores better for variant one, two and three. The most significant difference is observed for variant one. The VisFreq algorithm however scores better in replay fitness for variant one. We argue that in order to achieve perfect replay fitness a bigger process tree was required.

We cannot conclude that one of the algorithms is better. Both algorithms provide a similar solution. Only with respect to the time complexity of both algorithms we see an obvious difference. With 80,000 generations the brute force ETMc algorithm covered a large portion of the full search space. Opposed to the VisFreq algorithm which is a structured approach towards finding configurations and therefore only needs to calculate alignments once for the input process tree. This is similar to a single tree evaluation of the ETMc with respect to time complexity.

### Random trees and clustered event logs

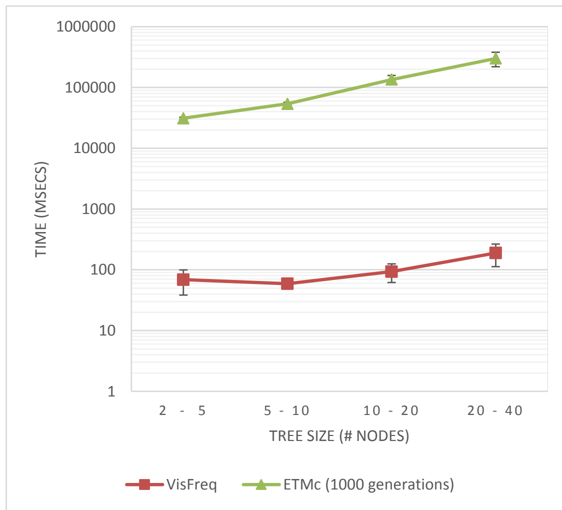
In the second part we tested both algorithms on randomly generated process trees with randomly generated event logs. Figure 7.6 shows the timing results and quality dimension scores. For the quality dimensions we included the scores of the reference configurable process tree used by the clustered event log simulator to generate the logs. We consider this reference configurable process tree to be the optimal result.

Figure 7.6a shows the timing results of both algorithms on a logarithmic scale. We can now clearly see the difference in time complexity (which we also discussed in the previous part of this experiment). For the tree sizes: 2 - 5, 5 - 10, 10 - 20 and 20 - 40 the VisFreq algorithm is approximately 450, 900, 1,000 and 1,600 times faster than the ETMc algorithm. The timings of both algorithms include a start-up time. Because of this start-up time we cannot directly observe that the VisFreq algorithm is similar to a single tree evaluation of the ETMc. However this seems a reasonable assumption since the time spent by the ETMc algorithm increases more rapidly than that of the VisFreq algorithm.

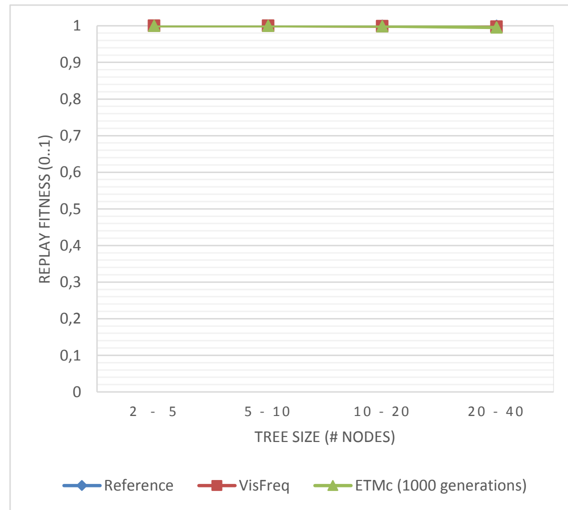
For all quality dimensions the result of the VisFreq algorithm closely approximates the reference configurable process tree. The ETMc however fails to achieve similar scores in the quality dimensions: precision and generalization. It does however improve in the number of configuration options. Having fewer configuration options seems to be better but we argue that the fewer configuration options directly relate to the worse score in the precision quality dimension.

Both approaches achieve perfect replay fitness and simplicity. This can be explained by the fact that any input process tree already has perfect replay fitness. This is inherent to the way the clustered event log simulator creates the event logs. Since replay fitness is considered to be the most important quality dimension the ETMc does not quickly prefer a solution with a worse replay fitness.

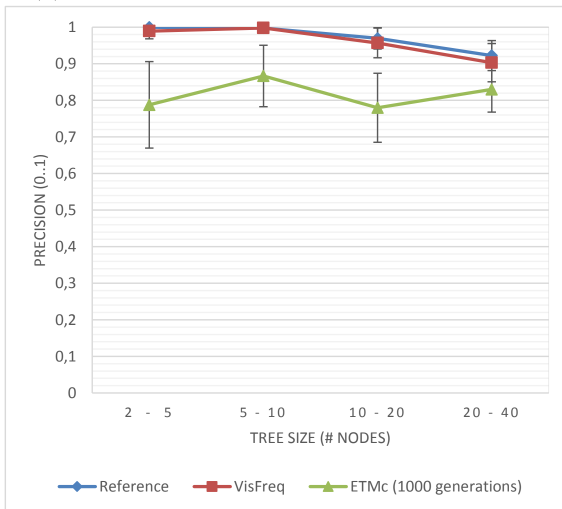
Running the ETMc for more generations will probably improve the results with respect to the quality dimensions. With enough generations the ETMc has traversed the full search space to such extend that it always finds a solution that closely approximates the reference configurable process tree. This however takes significantly longer than a run of the VisFreq algorithm. We can conclude that our approach is significantly faster than the ETMc while still providing close to optimal solutions.



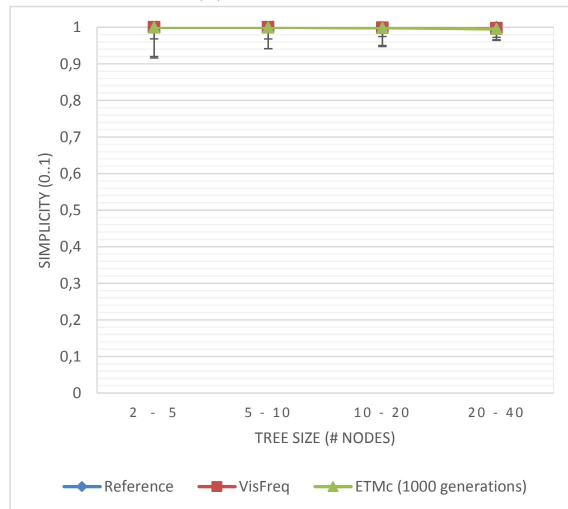

(a) Time in milliseconds on a logarithmic scale.



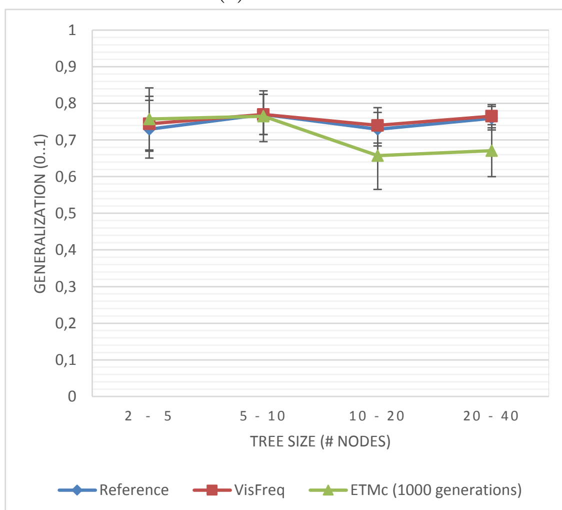

(b) Replay fitness.



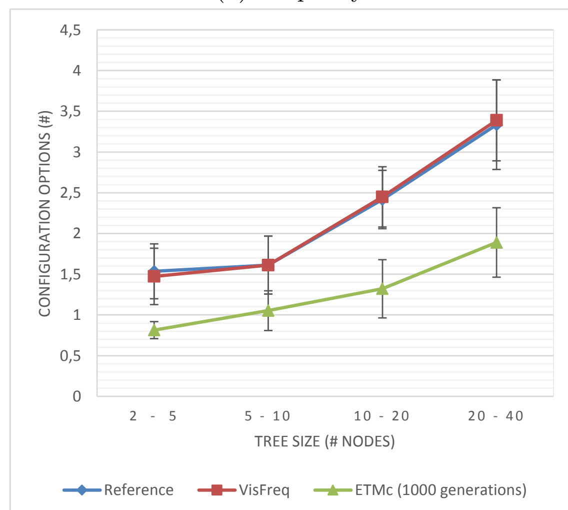

(c) Precision.




(d) Simplicity.




(e) Generalization.




(f) Number of configuration options.

Figure 7.6: Comparison of the different quality criteria calculated on the results of the VisFreq and ETMc approaches and quality criteria that were calculated on the reference configurable process tree. The error bars denote the 95% confidence intervals.

## 7.4 Evaluation of the full approach

To test the full approach we manually evaluate data sets and discuss the results of different approaches. First in Section 7.4.1 we evaluate the full approach on a simulated event log of the running example. Second in Section 7.4.2 we evaluate the full approach using the BPI challenge logs of 2012.

### 7.4.1 Running example

To evaluate the full approach on the running example (as proposed in Section 1.1), we build a simulation model which randomly executes one of the three variants of the running example. The simulation model records the payment method (`Payment`) and whether the student finished the process with a driving license or not (`Result`). The simulated event log contains 323 traces of variant one, 363 traces of variant two, and 314 traces of variant three. We performed clustering on this event log using the single linkage hierarchical clustering algorithm [16] and annotated the hierarchical clustering using the C4.5 decision tree algorithm [33]. Next we reduced the resulting hierarchical clustering based on data annotations. Using the Syntactic reasoning approach with default settings we find the annotated hierarchical clustering as shown in Figure 7.7. We ran the same procedure using the Levenshtein distance, Set-Of-Activities, Bag-Of-Activities and 3-Grams approaches (explained before in Section 7.2.1) and included the resulting hierarchical clusterings in Appendix B.2.1.

Table 7.5 summarizes the results of the different approaches. It shows the size of the hierarchical clusterings in number of clusters and marks the variants of the running example for which we could identify a cluster. Both Set-Of-Activities and Bag-Of-Activities are feature-set approaches that cannot observe the order of execution aspect. The difference between variant two and three of the running example is the order of execution of activities *Driving Class* (C) and *Pay For Class* (D). Therefore it is impossible for these approaches to observe the difference between these variants.

The Set-Of-Activities approach was also unable to find a cluster representing variant one. The Set-Of-Activities approach does not take into account the number of times an activity occurred. Therefore the execution of activity *Pay For Class* (D) for variant two and three is only taken into account once in the final distance. When comparing a trace of variant two or three with a trace of variant one, the execution of activity D is therefore not penalized heavier than e.g., when a student fails once, which adds the execution of of activity *Failed* (H).

All other approaches are able to identify the three variants of the running example. The Syntactic reasoning approach however finds the smallest and simplest hierarchical clustering. For the approaches that identify the three variants of the running example, we used the clusters identifying the variants to discover configurations using our configuration discovery algorithm. This resulted in the configurable process tree as shown in Figure 1.2.

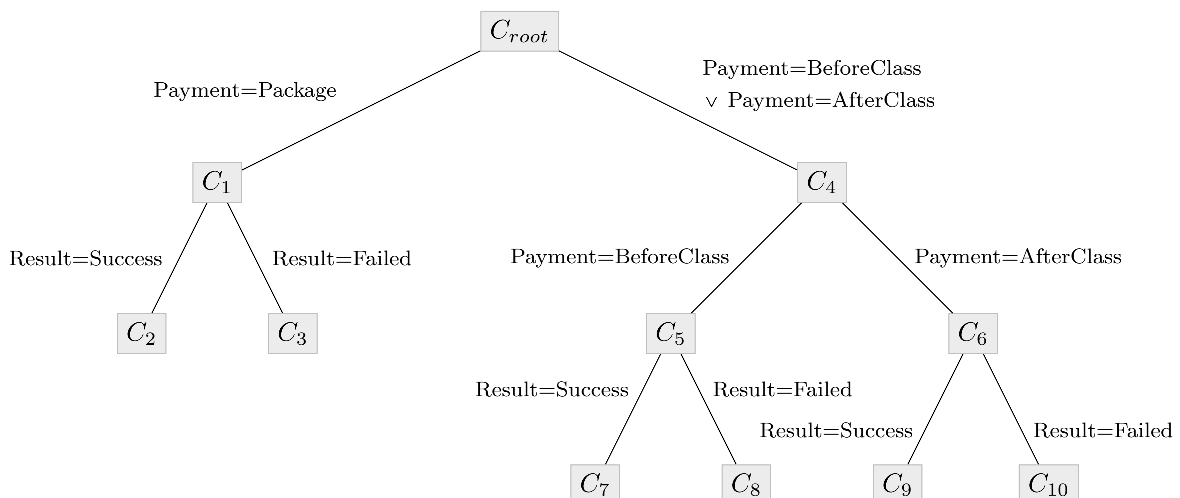



Figure 7.7: Hierarchical clustering of the running example using the Syntactic reasoning approach.

| *Dissimilarity measure* | *Clusters* | *Identifies variants* | | |
|---|---|---|---|---|
| | | *1. Package* | *2. Payment After* | *3. Payment Before* |
| Syntactic reasoning | 11 | ✓ | ✓ | ✓ |
| Levenshtein distance | 41 | ✓ | ✓ | ✓ |
| Set-Of-Activities | 11 | - | - | - |
| Bag-Of-Activities | 56 | ✓ | - | - |
| 3-Grams | 19 | ✓ | ✓ | ✓ |

Table 7.5: Per dissimilarity measure the size of the hierarchical clustering in number of clusters and whether a variant of the running example can be distinguished in this hierarchical clustering.

### 7.4.2 BPI challenge logs 2012

The BPI challenge 2012[2] is a competition in which the participants are asked to analyze a real-life event log using whatever techniques available and report on this. The provided event log describes the process for a personal loan or overdraft within a Dutch Financial Institute. The event log contains 262,000 events in 13,087 traces. The event log contains three data attributes per trace: `concept:name`, `AMOUNT_REQ` and `REG_DATE`. These data attributes respectively denote a unique name of the trace, the loan amount, and the time of registration. We immediately discarded the data attribute: `concept:name` since it is unique per trace and therefore cannot be used to explain groups of traces (clusters). In the event log the requested loan amount is marked as a string, we modified this to an integer to allow range conditions by the data annotation algorithm. The goal of this experiment is to identify whether there is a relation between the trace data and the observed behavior and whether we can make a classification of the traces.

We first opened the full event log and discovered a process tree using the Inductive Miner [28]. This gave us a huge and complex process tree, indicating that the process contains many deviations. Using this discovered process tree and the event log, we performed trace clustering using the single linkage hierarchical clustering algorithm [16] using as dissimilarity measure the Syntactic reasoning approach with default settings. Next we annotated the hierarchical clustering using the C4.5 decision tree algorithm [33]. Finally we reduced the hierarchical clustering based on data annotations. This resulted in a huge hierarchical clustering wherein the root branches out to many clusters with each only a few traces. Each of these clusters is annotated with a long chain of conditions. It seemed like the annotation algorithm tried to explain every trace in the cluster individually. We can therefore conclude that both data attributes: `AMOUNT_REQ` and `REG_DATE` do not relate to behavior. We also tried to exclude one of the two data attributes, this however did not improve the results. In the remainder of this experiment we discarded both data attributes.

Since the event log does not contain data attributes that describe behavior, we enriched it with the occurrence of events using the techniques proposed in Section 4.3. We ran the same procedure as before to obtain an annotated hierarchical clustering. This resulted in a huge hierarchical clustering from which we could not make much sense. We could however identify an initial distinction between *approved* and other traces, and recognize that the event log probably contains uncompleted traces.

Next we investigated the meaning of the activities and identified interesting activities for our investigation. The BPI challenge website explains that the activities prefixed with *A_* indicate the states of the loan application. We first filtered the event log to only contain the application states. Next we discovered a process tree using the Inductive Miner [28]. Inspection of this process tree showed that the process always starts with the activity: *A_SUBMITTED*, and that the process ends with at least one of the activities: *A_APPROVED*, *A_REGISTERED*, *A_ACTIVATED*, *A_DECLINED* or *A_CANCELLED*. We further filtered the event log to only include events of these six activities. Again we discovered a process tree using the Inductive Miner [28], this resulted in the process tree shown in Figure 7.8. Using this process tree and the filtered event log we ran the same procedure as before to obtain an annotated hierarchical clustering.

Figure 7.9 shows the resulting hierarchical clustering. Data attributes prefixed with `EO:` indicate the occurrence of activities. If the activity occurred it has the value `True`, else it has the value `False`. The hierarchical clustering shows a clear segregation between *approved*, *canceled* and *declined* traces. It is reasonable to assume that these are the end-states of the traces. Next to these clusters we also have a

[2]The official website is: `http://www.win.tue.nl/bpi/2012/challenge`.

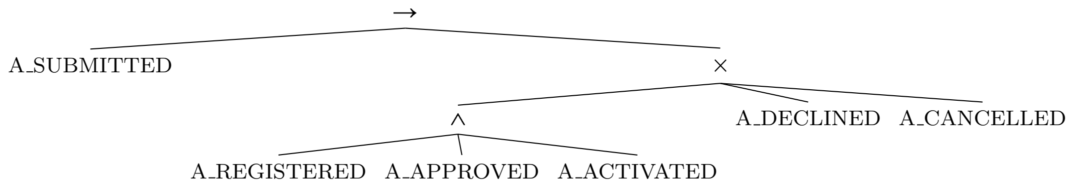



Figure 7.8: Process tree discovered using the Inductive Miner for the initial filtered event log of the BPI challenge 2012.

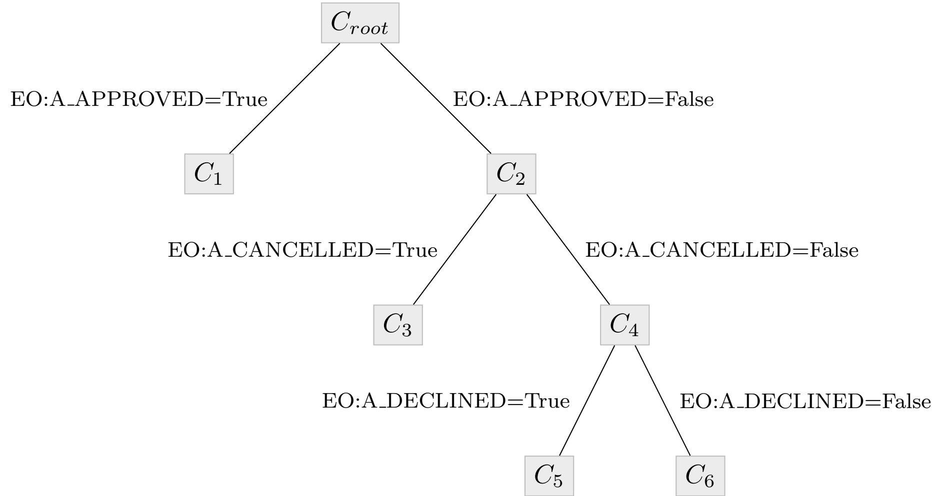



Figure 7.9: Resulting hierarchical clustering of the initial filtered event log of the BPI challenge 2012.

cluster with traces that do not belong to any of these end-states, we consider this to be the *uncompleted* traces. Table 7.6 shows the number of traces and the state of the traces for each of the leaf clusters. We compared our result with a submission of the BPI challenge 2012 [25] and can conclude that we have found a clustering that is identical to their first-level classification of the traces (shown in Figure 7.10). We however found this classification in a semi-automatic way.

Next to this initial classification in the end-states, the report [25] includes sub-classifications into whether an offer was sent and/or there was a fraud check. Our previous result could not identify these sub-classifications since we removed the events representing these types of activities. Therefore we again manually inspected the event log and found two activities: *O_SENT* and *W_Beoordelen fraude*, which respectively represent sending an offer and accessing for fraud. We filtered the original event log to only include the previous six activities and the two new activities. Again we discovered a process tree using the Inductive Miner [28], this resulted in the process tree shown in Figure 7.11. We ran the same procedure as before to obtain an annotated hierarchical clustering. Figure 7.9 partially, i.e., some of the deeper levels are removed, shows the hierarchical clustering (the full hierarchical clustering is included in Appendix B.2.2). The hierarchical clustering now also segregates whether an offer was sent or not. The fraud check is however less visible and only for clusters $C_1$ and $C_7$ the traces were segregated. Table 7.7 shows per cluster the number of traces, whether an offer was sent in these traces, and whether a fraud

| *Cluster* | *Number of traces* | *Trace state* |
|---|---|---|
| $C_1$ | 2,246 | Approved |
| $C_3$ | 2,807 | Canceled |
| $C_5$ | 7,635 | Declined |
| $C_6$ | 399 | Uncompleted |

Table 7.6: Summarizes the leaf clusters of Figure 7.9. It shows the number of traces and the state of the traces per cluster.

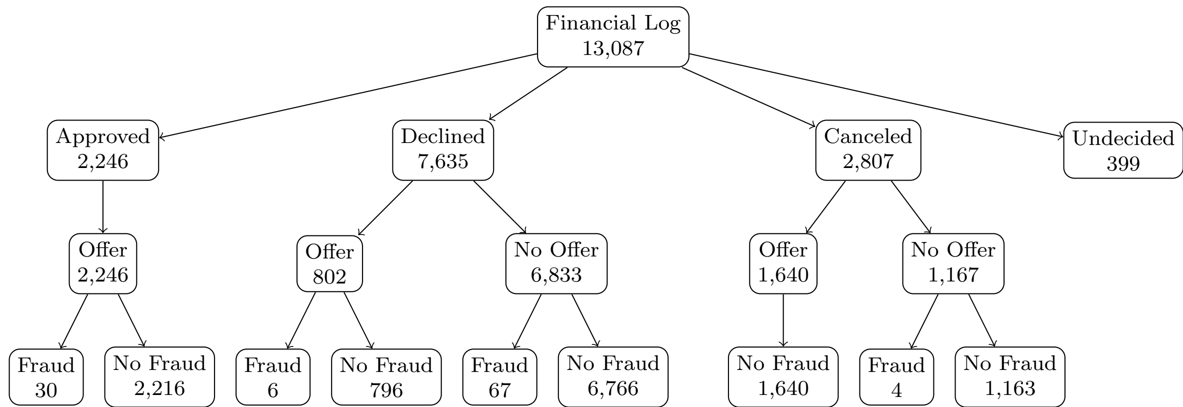



Figure 7.10: Classification of the BPI challenge 2012 traces [25].

check was performed. Our trace clustering is very similar to the classification shown in Figure 7.10. We were however unable to detect the very small fraud check clusters in the case of a canceled trace for which no offer was sent and in the case of a declined trace for which an offer was sent.

Classifying traces in large real-life event logs like the BPI challenge 2012 log is a cumbersome task. With manual inspection and filtering it is possible to classify the traces (as shown by one of the submissions [25]). However more complex dependencies between activities would be nearly impossible to identify in this way. Trace clustering with data annotations is no holy grail in that it always provides a perfect clustering, not to mention that the perfect clustering is context-specific and probably impossible to define, it does however provide much insight to an end-user. In this experiment we immediately observed the segregation of *approved* and other traces and observed that the event log probably contains uncompleted traces. With these insights the user can further pre-process the event log by filtering noisy or irrelevant events. A few of these manual steps made it possible to find a clustering that provides much information to the user. We initially found a segregation of the traces based on its end states. Further manual steps made it possible to immediately find a trace clustering of the traces in which we could also identify whether an order was sent and whether there was a check for fraud.

In this experiment we did not have a reference process tree and had to discover a process tree from the event log which was then used as input for the Syntactic reasoning approach. For the unfiltered event log the Inductive Miner [28] discovered a very big and complex process tree because of the many deviations in the process. We argue that such a process tree provides less information to the Syntactic reasoning approach opposed to what a reference process model could have provided. A reference process model could have made it possible to find a good classification of the traces without any filtering. This reference process model was however not supplied and could therefore not be used. We do however think that with some manual steps a discovered process tree does help the trace clustering. The Syntactic reasoning approach could be extended with process discovery algorithms such that it no longer needs an input process tree. Any process discovery algorithm that outputs one or more process trees would be applicable. This mean that by improving process discovery algorithms, which is a very active field of research, we directly improve upon trace clustering.

We had to enrich the BPI challenge 2012 event log with the occurrence of events since the traces itself did not contain any data which support the choices that were made during execution. This however adds attributes for every activity, which allows the data annotation algorithm to find an appropriate data expression for every cluster. Because of this the reduction algorithm based on data annotations is a lot less effective. It might be easier to find a good classification for event logs that contain data attributes which describe the choices made during the execution of the traces.

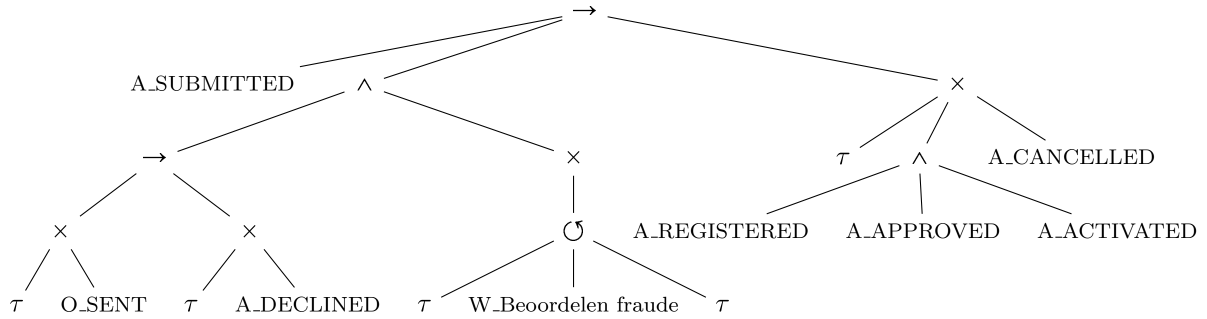
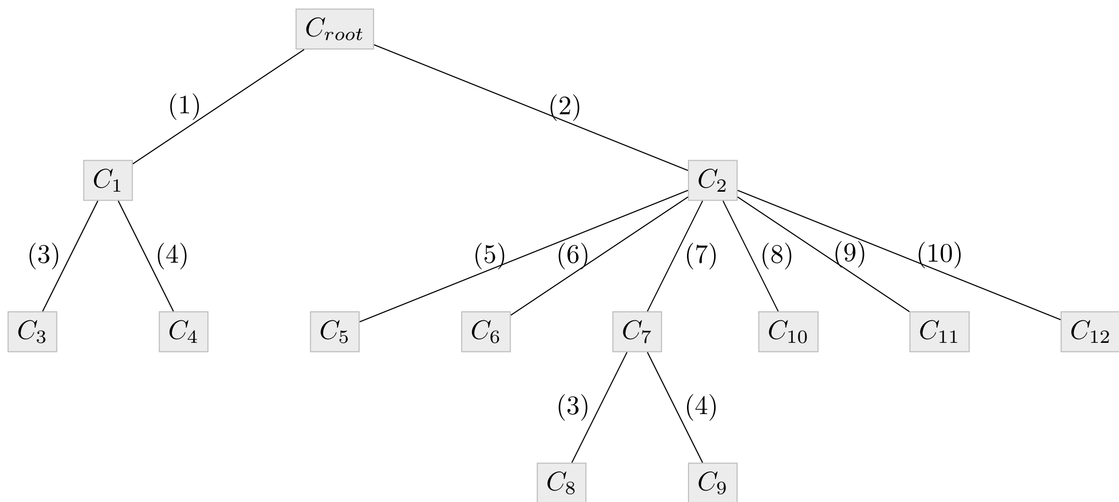



Figure 7.11: Process tree discovered using the Inductive Miner for the filtered event log of the BPI challenge 2012 which includes sending of offers and checks for fraud.




| | | | | | |
|---|---|---|---|---|---|
| (1) | EO_APPROVED=True | | | | |
| (2) | EO_APPROVED=False | | | | |
| (3) | EO:W_Beoordelen fraude=True | | | | |
| (4) | EO:W_Beoordelen fraude=False | | | | |
| (5) | EO:A_DECLINED=False | ∧ | EO:A_CANCELLED=True | ∧ | EO:O_SENT=False |
| (6) | EO:A_DECLINED=False | ∧ | EO:A_CANCELLED=True | ∧ | EO:O_SENT=True |
| (7) | EO:A_DECLINED=True | ∧ | EO:A_CANCELLED=False | ∧ | EO:O_SENT=False |
| (8) | EO:A_DECLINED=True | ∧ | EO:A_CANCELLED=False | ∧ | EO:O_SENT=True |
| (9) | EO:A_DECLINED=False | ∧ | EO:A_CANCELLED=False | ∧ | EO:O_SENT=False |
| (10) | EO:A_DECLINED=False | ∧ | EO:A_CANCELLED=False | ∧ | EO:O_SENT=True |

Figure 7.12: Partial trace clustering of the filtered event log of the BPI challenge 2012 which includes sending of offers and checks for fraud.

| *Cluster* | *Number of traces* | *Trace state* | *Offer sent* | *Fraud check* |
|---|---|---|---|---|
| $C_1$ | 2,246 | Approved | Yes | - |
| $C_2$ | 10,841 | Canceled/Declined/Uncompleted | - | - |
| $C_3$ | 30 | Approved | Yes | Yes |
| $C_4$ | 2,216 | Approved | Yes | No |
| $C_5$ | 1,167 | Canceled | No | - |
| $C_6$ | 1,640 | Canceled | Yes | - |
| $C_7$ | 6,833 | Declined | No | - |
| $C_8$ | 67 | Declined | No | Yes |
| $C_9$ | 6,766 | Declined | No | No |
| $C_{10}$ | 801 | Declined | Yes | - |
| $C_{11}$ | 72 | Uncompleted | No | - |
| $C_{12}$ | 326 | Uncompleted | Yes | - |

Table 7.7: Summarizes the clusters of Figure 7.12. Per cluster it shows the number of traces, the state of the traces, whether an offer was sent, and whether a fraud check was performed.

# Chapter 8

# Conclusion and future work

In this chapter we conclude this thesis, discuss the limitations of our work, and propose ideas for future work. First in Section 8.1 we conclude our results and briefly answer our research questions. Second in Section 8.2 we discuss the limitations of our work and propose ideas for future work.

## 8.1 Conclusion

The development and maintenance cost greatly increases with the increased use of information systems. This causes more and more organizations to start using shared business process management infrastructures, Software-as-a-Service (SaaS) and Cloud Computing. This shows that companies are willing to adapt themselves to information systems in order to reduce costs. Configurable process models provide a means for organizations to keep some freedom in the use of an information system. They can configure these models in such way that it best suits their way of working.

Related work proposes many different approaches to discover a configurable process model from a collection of event logs. These approaches however discover the model at an organizational level. In this thesis we proposed discovery at a behavioral level. Rather than looking at the traces of different organizations we directly consider the behavior of the traces and group those that describe similar behavior. Grouping behaviorally similar traces of an event log directly relates to the field of trace clustering. Clustering algorithms often use a dissimilarity measure to identify objects that are similar to each other. Existing dissimilarity measures for trace clustering often have difficulties with parallel executable behavior and loops. We proposed a new dissimilarity measure that solves these issues by using an input process tree. In this thesis we also explained how we can annotate groups of traces with the data in the event log. By doing this we provide insights to the end-user in how data implicates the observed behavior. Each group of traces can be considered to be an event log. From such a collection of event logs it is possible to discover a configurable process model. Before this work the ETMc was the only approach to discover a configurable process tree. This is however a brute force approach. In this thesis we proposed a new structured approach that uses execution frequencies and observed execution orderings to find configurations for a process tree and a collection of event logs, resulting in a configurable process tree.

In order to reach our goal we answered the four research questions as follows:

1. **How can we identify groups of behaviorally similar traces in an event log?** (Chapter 3)

   This question directly relates to the field of trace clustering. Before this work the usage of trace clustering was primarily in process discovery, by finding homogeneous groups of traces the complexity of discovered models may be reduced. In this thesis we use trace clustering to find groups of traces that imply configurations on a process model. Most clustering algorithms use a dissimilarity measure. Related work proposes different approaches towards defining this measure, however most of these approaches suffer problem problems because of their lack of model-awareness. Therefore we have proposed a new dissimilarity measure: the *Syntactic reasoning approach.* We have explained how our dissimilarity can be applied in clustering and how *preferred clusters* can be suggested to the end-user. The final result is a *hierarchical clustering.* This hierarchical clustering is a hierarchical segregation of the traces in the input event log.

2. **How can we explain these groups based on data?** (Chapter 4)

   Decision mining can be applied to explain every choice during the execution of a process model by the data of the traces. The problem stated in this research question is very similar. The groups of traces in the hierarchical clustering are grouped since they describe similar behavior, i.e., they make similar choices. To explain the groups we *annotate* the hierarchical clustering with the data of the traces such that the end-user can pick a set of preferred clusters. We have shown how this problem can be transformed to a classification problem and discussed the use of two classification algorithms, the *naive Bayes* classifier and *C4.5 decision tree* classifier.

   After annotation of the hierarchical clustering it may still be huge and incomprehensible for an end-user. Existing techniques to reduce a hierarchical clustering do not take into account the data annotations. We have proposed an algorithm that reduces the hierarchical clustering based on the data annotations. Another issue is that sometimes we cannot explain a hierarchical clustering with the data of the event log. For these cases we explained how event logs can be enriched. By adding trace-features as data to the traces it may become possible to find an explanation for the clusters. By answering this research question we obtained an *(reduced) annotated hierarchical clustering*. From this hierarchy the end-user can select their preferred clusters,

3. **How can we create a configuration for a given process tree and a group of traces?** (Chapter 5)

   We answered this research question by first discussing four different approaches towards configurable process discovery. Evaluation of previous work shows that discovery approaches overcome the disadvantages of merging approaches. We have explained that for any group of traces we can find a configuration for a given process tree using the ETMc algorithm. This algorithm is however a brute force approach. Therefore we proposed a new *structured configuration discovery approach*. Our configuration discovery approach uses execution frequencies and observed execution orderings to discover the configurations.

4. **How can we select a good configurable process tree for a selection of groups of traces?** (Chapter 5)

   Situations may arise where we do not have a single reference process tree but instead multiple reference process trees. For example, merging process trees is not trivial and may result in multiple trees, or algorithms like the ETMd can discover many different process trees which each specialize in different aspects. It is not trivial which configurable process tree is the best. We presented two ways of calculating an *overall configurable process tree quality score*. Using this score an end-user can evaluate which of the configurable process trees is the best.

Our full approach is implemented as a plug-in in the ProM 6 framework. In Chapter 6 we provided a user guide and explained the extensible framework of our plug-in. Our plug-in provides the user an interactive way to perform trace clustering with data explanations, and discovery of a configurable process tree for a selection of preferred clusters. The plug-in is setup in such way that it is easily extensible with new dissimilarity measures, clustering algorithms, and classification algorithms.

We have performed extensive experimental evaluation on the newly proposed approaches and our full approach. In Chapter 7 we explained our experimental setup and discussed the results. For different types of process trees and event logs the most suitable dissimilarity measure for trace clustering seems to be different. However our Syntactic reasoning approach is always among the best scoring approaches, or the best scoring approach, even without tweaking. Next to this, our approach produces the most stable results, i.e., when cutting the hierarchy there is not much difference between the worst and best possible cut. Furthermore, our approach supports context-specific tweaking which other approaches do not. The experiments showed that tweaking improves the clustering results. Our configuration discovery approach was compared with the ETMc on the running example of the ETMc paper [12] and we experimented with random process trees and event logs. In both comparisons we can conclude that our algorithm is much faster and produces solutions that are close to a reference solution. The ETMc is only able to produce good solutions after sufficient generations. Finally, we tested our full approach on our running example and the BPI challenge 2012 event log. We have explained how our full approach is applicable on such event logs and have shown that our full approach is effective in identifying different behavior and in finding a classification of traces.

## 8.2 Future work

In this section we discuss the limitations of our work and discuss possible improvements of the proposed approaches. In Section 8.2.1 we discuss possible improvements of the Syntactic reasoning approach and the implementation of a trace clustering plug-in. Section 8.2.2 discusses improvements in the annotation of the hierarchical clustering. Section 8.2.3 discusses how our configuration discovery algorithm can be further improved and applied in different contexts. Finally in Section 8.2.4 we propose additional experimental evaluation which was not included due to time constraints.

### 8.2.1 Trace clustering

In Chapter 3 we discussed trace clustering. Clustering algorithms often require a dissimilarity measure. We proposed a new dissimilarity measure in Section 3.3, the Syntactic reasoning approach. Our approach improves over existing approaches by the use of one or more reference process trees. We think that the following ideas might still be noteworthy to look into:

**Loop roll-up improvement:** The Loop roll-up error-transformation currently simply adds additional loop iterations to one partially ordered alignment (POA) such that both POAs have the same number of iterations for a loop instance. This is followed by a second step that repairs repeating mismatches. In this second step only the first occurrence of a mismatch is kept, other mismatches are removed. In this way a repeating mismatch is only penalized once. In some cases this may not be appropriate. An alternative is to use the insert/delete/substitution error-transformations to make sure that both POAs have the same number of iterations, while decreasing the costs of the error-transformations with every iteration of the loop.

**Generic cost function:** The current setup of the cost function of the Syntactic reasoning approach does allow some context-specific tweaking but this is however limited. Currently we can only change the weights of the error-transformations. The different cases in the substitution error-transformation provide some freedom in how particular types of substitutions should be penalized but we think that this can be greatly improved. In Section 3.1.2 we explained the Generic Edit Distance [23, p.86] which is a generalization of the Levenshtein and allows for a fully customized distance function. The same principle could be applied in our approach. The current costs could be used as defaults, from which the end-user could tweak specific error-transformations per type of activity and/or corresponding type of alignment move.

**Weighted distance aggregation:** In definition 3.1 of Section 3.3.5 we define four basic different aggregation functions. Initially we performed tests with a fifth aggregation function that weights the distances based on the quality dimensions of the corresponding process model. The idea is that distances calculated for better process trees should be weighted heavier than distances calculated for worse process trees. Due to time constraints we were however unable to look deeper into this. The main issue is that it is not trivial which quality dimensions indicate that a process tree is better and which quality dimensions indicate that a process tree is worse for the Syntactic reasoning approach.

**Trace clustering plug-in:** Chapter 6 explains the implementation of our plug-in of the full approach. The components used in this plug-in are implemented in such way that it should be relatively easy to implement a plug-in purely focused at trace clustering. We were however unable to implement this due to time constraints.

### 8.2.2 Explaining clusters by data

In Chapter 4 we explained how a hierarchical clustering can be explained by the data of an event log. We also proposed a new reduction algorithm and proposed enrichment of the event log. We still have the following ideas for improvement:

**Naive Bayes classifier:** In Section 4.1 we proposed how the naive Bayes classifier can be used to annotate the hierarchical clustering, we however did not implement this algorithm. We proposed to add the probabilities to the edges. An alternative is to add the absolute occurrences to the

edges. The probabilities and absolute occurrences are directly interchangeable but the absolute occurrences might provide more insight to the end-user.

**C4.5 contradictions:** The C4.5 decision tree algorithm can result in contradictions in the final annotated hierarchical clustering (as discussed in Section 4.1). We provided two solutions towards this problem. Both solutions are however not implemented at this time. Future work could look into when a solution is appropriate and how these should be applied.

**Data implications:** Some data attributes might implicate values on others. This is especially visible when enriching trace data with the occurrence of events, e.g., all events under an $\wedge$-node are always executed and therefore directly imply each other (not considering noise). An analysis of these implications might provide the user with insights and may simplify the classification problem.

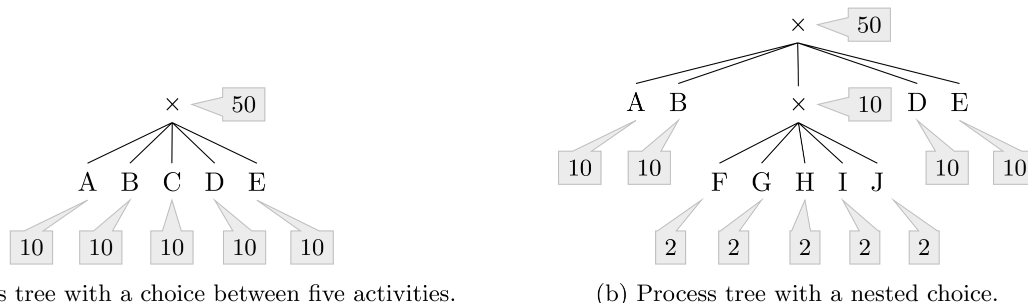
**Data annotation subsets:** In Section 4.2 we proposed a reduction algorithm based on data annotation. The current version only merges clusters with identical data annotations. However we could also merge clusters when the data annotation of one cluster describes a subset of the data annotation of an other cluster, e.g., in the case of range conditions on numbers.

### 8.2.3 Configurable process discovery

In Chapter 5 we proposed a new structured configuration discovery approach for process trees. We effectively tackled a problem with respect to loops, there is however still a limitation in the current version of our approach with respect to decision points. We also discussed that our algorithm can be applied in an evolutionary way, in this section we propose two ideas. We now discuss our ideas in more detail:

**Decision points:** The decision points in a process tree are represented by $\times$, $\vee$ and $\circlearrowleft$-nodes. We use Figure 8.1 to illustrate the problem. Assume we have an event log with 50 traces that each execute the $\times$-node of this process tree. The traces uniformly execute the children of the $\times$-node. Figure 8.1a shows a distribution of the 50 traces among the children. Any threshold $t_{hb} < 0.8$ causes the configuration discovery algorithm to block or hide all of the children. This is expected behavior since if we choose a threshold below 80%, we expect the children to be blocked or hidden if they are executed less than 20% of the time. However a problem occurs in the cases where we have nested choices. Figure 8.1b shows a process tree with again a uniform distribution among the children. For this process tree all children of the nested choice are blocked or hidden for any threshold $t_{hb} < 0.96$. We can easily see how this problem grows worse with more nested choices. Please note that this problem only occurs when hiding or blocking since all operator downgrading equations are already context-specific.

The problem occurs since we always compare the number of traces related to a node against the number of traces in the full event log. Currently the following equation is used to determine whether




(a) Process tree with a choice between five activities.

(b) Process tree with a nested choice.

Figure 8.1: A process tree with a choice between five activities (a) and a process tree with a nested choice. Both process trees are annotated with callouts that indicate the number of traces associated to the node.

blocking or hiding should be applied:

$$\frac{|L_n|}{|L|} < 1 - t_{hb} \tag{8.1}$$

We can modify this equation to compare against a context specific number of traces, as follows:

$$\frac{|L_n|}{s} < 1 - t_{hb} \tag{8.2}$$

Where $s \geqslant 0$ is the context-specific number that initially has the value of $|L|$. $s$ may only decrease when visiting deeper choice nodes, i.e., a shallow node may influence the value of $s$ for a deeper node, but never the other way around.

A trivial solution is to set $s$ to the number of traces related to the decision node for all children of this decision node. This however does not take into account the size of the full event log at all. Another solution would be to assume a uniform distribution of traces among the children of the choice node. In this case we divide $s$ by the number of children related to the decision node. However assuming a uniform distribution may not be appropriate, especially for $\vee$ and $\circlearrowleft$-nodes since for these operators multiple children may be executed. We can raise the assumption by taking into account the execution frequencies of the children, i.e., the number of times a child is executed.

We have presented multiple options to solve this limitation. Due to time constraints we were however unable to implement a solution and use it in our experimental evaluation.

**Evolutionary threshold optimization:** It is not trivial which values should be taken for the thresholds of our configuration discovery approach. We can implement a new algorithm in the ETM framework that tries to optimize the thresholds. The most basic approach would be to randomly change the thresholds and create new candidate configurable process trees. Every generation we keep the best configurable process tree and modify all others. After sufficient generations this gives us the best possible (according to the specified quality criteria) configurable process tree for our algorithm.

**Evolutionary discovery:** As mentioned in Section 5.2 our configuration discovery approach can easily modified into a mutator for the ETM framework. This makes our algorithm applicable in the ETMc algorithm which discovers the process model and configurations at the same time. Currently our algorithm always generates configurations for the full process tree. The algorithm can be modified to only determine the configuration options for parts of the process tree, which the mutator randomly picks. Another idea is to mix our approach with the random mutator of the ETMc, this might yield surprising results. Like proposed in the previous item, the mutator could also randomly change the thresholds.

### 8.2.4 Additional experimental evaluation

In Chapter 7 we provided the results of our experimental evaluation. We now propose additional experiments that were not performed due to time constraints:

**Noisy event logs:** In none of our experiments on artificial or simulated data we added noise to the event logs. We did some preliminary tests in the comparison of dissimilarity measures with 10% noise and the initial results were very promising, i.e., our Syntactic reasoning approach seemed to outperform all other approaches. Due to time constraints we were however unable to include the findings in this thesis.

The thresholds of our configuration discovery approach provide a means to deal with noise. In the experiments of part two of Section 7.3, we set all thresholds to 0.95 but we might as well have used 1.0 since no noise was present in the simulated event logs (not taking into account the issue with nested choices). Further experiments are required to validate whether the thresholds work as expected.

**Imperfect process trees:** The input process trees of the configuration discovery approaches in the experiments of Section 7.3 always had perfect replay fitness. Results might be different when noise is present in the process trees. Therefore further experiments are required that introduce random

flaws in the process trees and then try to discover the configurations for the process tree and the simulated event log.

**Duplicate activities:** In Section 7.2 we compared multiple dissimilarity measures. In none of the investigated cases we allowed for duplicate activities in the process trees. One of the best scoring approaches is the Set-Of-Activities approach. This approach is however unable to deal with clusters that are identified by the same type of activity. Our Syntactic reasoning approach does not suffer from this issue since it uses process trees. Therefore we expect that our approach outperforms the Set-Of-Activities approach in such situations.

**Comparison of one vs. multiple process trees:** Section 3.3.5 provides equations to aggregate the calculated distances of the Syntactic reasoning approach for multiple models. It seems like a reasonable assumption that many process models which each specialize in different aspects approach an ideal process model for the Syntactic reasoning approach. However experiments are required to verify this assumption.

# Bibliography

[1] W.M.P. van der Aalst. Formalization and verification of event-driven process chains. *Computing Science Reports 98/01*, 1998. 8

[2] W.M.P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer Publishing Company, Incorporated, 1st edition, 2011. 1, 7, 8, 11, 13

[3] W.M.P. van der Aalst, A. Adriansyah, and B.F. van Dongen. Replaying history on process models for conformance checking and performance analysis. *Wiley Int. Rev. Data Min. and Knowl. Disc.*, 2(2):182–192, March 2012. 10

[4] W.M.P. van der Aalst, K.M. Hee, A.H.M. ter Hofstede, N. Sidorova, H.M.W. Verbeek, M. Voorhoeve, and M.T. Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363, 2011. 8

[5] R. Accorsi and T. Stocker. Discovering workflow changes with time-based trace clustering. In K. Aberer, E. Damiani, and T. Dillon, editors, *Data-Driven Process Discovery and Analysis*, volume 116 of *Lecture Notes in Business Information Processing*, pages 154–168. Springer Berlin Heidelberg, 2012. 16

[6] A. Adriansyah, B.F. van Dongen, and W.M.P. van der Aalst. Conformance checking using cost-based fitness analysis. In *Proceedings of the 2011 IEEE 15th International Enterprise Distributed Object Computing Conference*, EDOC '11, pages 55–64, Washington, DC, USA, 2011. IEEE Computer Society. 10

[7] M. Ankerst, M.M. Breunig, H. Kriegel, and J. Sander. Optics: Ordering points to identify the clustering structure. pages 49–60. ACM Press, 1999. 23

[8] N. Assy, W. Gaaloul, and B. Defude. Mining configurable process fragments for business process design. In M.C. Tremblay, D. VanderMeer, M. Rothenberger, A. Gupta, and V. Yoon, editors, *Advancing the Impact of Design Science: Moving from Theory to Practice*, volume 8463 of *Lecture Notes in Computer Science*, pages 209–224. Springer International Publishing, 2014. 34

[9] J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. A genetic algorithm for discovering process trees. pages 1–8, June 2012. 12, 34

[10] J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. On the role of fitness, precision, generalization and simplicity in process discovery. In R. Meersman, H. Panetto, T. Dillon, S. Rinderle-Ma, P. Dadam, X. Zhou, S. Pearson, A. Ferscha, S. Bergamaschi, and I.F. Cruz, editors, *On the Move to Meaningful Internet Systems: OTM 2012*, volume 7565 of *Lecture Notes in Computer Science*, pages 305–322. Springer Berlin Heidelberg, 2012. 8, 11

[11] J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. Towards cross-organizational process mining in collections of process models and their executions. In F. Daniel, K. Barkaoui, and S. Dustdar, editors, *Business Process Management Workshops*, volume 100 of *Lecture Notes in Business Information Processing*, pages 2–13. Springer Berlin Heidelberg, 2012. 1

[12] J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. Mining configurable process models from collections of event logs. In F. Daniel, J. Wang, and B. Weber, editors, *Business Process Management*, volume 8094 of *Lecture Notes in Computer Science*, pages 33–48. Springer Berlin Heidelberg, 2013. 1, 9, 12, 33, 34, 35, 39, 52, 53, 66

[13] R.J.G.B. Campello, D. Moulavi, and J. Sander. Density-based clustering based on hierarchical density estimates. In J. Pei, V.S. Tseng, L. Cao, H. Motoda, and G. Xu, editors, *Advances in Knowledge Discovery and Data Mining*, volume 7819 of *Lecture Notes in Computer Science*, pages 160–172. Springer Berlin Heidelberg, 2013. 23, 24

[14] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995. 26

[15] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Trans. Inf. Theor.*, 13(1):21–27, September 2006. 26

[16] W.H.E. Day and H. Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification*, 1(1):7–24, 1984. 23, 48, 58, 59

[17] P. Domingos and M. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997. 26

[18] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996. 23

[19] F. Gottschalk, W.M.P. van der Aalst, and M.H. Jansen-Vullers. Merging event-driven process chains. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems: OTM 2008*, volume 5331 of *Lecture Notes in Computer Science*, pages 418–426. Springer Berlin Heidelberg, 2008. 31, 33

[20] F. Gottschalk, W.M.P. van der Aalst, and M.H. Jansen-Vullers. Mining reference process models and their configurations. In R. Meersman, Z. Tari, and P. Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2008 Workshops*, volume 5333 of *Lecture Notes in Computer Science*, pages 263–272. Springer Berlin Heidelberg, 2008. 33

[21] F. Gottschalk, W.M.P. van der Aalst, M.H. Jansen-Vullers, and M. La Rosa. Configurable workflow models. *International Journal of Cooperative Information Systems*, 17(02):177–221, 2008. 1, 9, 31

[22] A. Hallerbach, T. Bauer, and M. Reichert. Capturing variability in business process models: The provop approach. *Journal of Software Maintenance*, 22(6-7):519–546, October 2010. 9

[23] R.P. Jagadeesh Chandra Bose. *Process mining in the large: preprocessing, discovery, and diagnostics.* PhD thesis, Technische Universiteit Eindhoven, Eindhoven, Netherlands, 2012. 16, 17, 30, 67

[24] R.P. Jagadeesh Chandra Bose and W.M.P. van der Aalst. Context aware trace clustering: Towards improving process mining results. In *Proceedings of the SIAM International Conference on Data Mining (SDM 2009)*, pages 401–412. Society for Industrial and Applied Mathematics, 2009. 16, 30

[25] R.P. Jagadeesh Chandra Bose and W.M.P. van der Aalst. Process mining applied to the bpi challenge 2012: Divide and conquer while discerning resources. In M. La Rosa and P. Soffer, editors, *Business Process Management Workshops*, volume 132 of *Lecture Notes in Business Information Processing*, pages 221–222. Springer Berlin Heidelberg, 2013. 60, 61

[26] A.B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962. 19

[27] M. La Rosa, M. Dumas, R. Uba, and R. Dijkman. Business process model merging: An approach to business process consolidation. *ACM Trans. Softw. Eng. Methodol.*, 22(2):11:1–11:42, March 2013. 33

[28] S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst. Discovering block-structured process models from event logs - a constructive approach. In J. Colom and J. Desel, editors, *Application and Theory of Petri Nets and Concurrency*, volume 7927 of *Lecture Notes in Computer Science*, pages 311–329. Springer Berlin Heidelberg, 2013. 8, 59, 60, 61

[29] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, C., 1967. University of California Press. 23

[30] A.K.A. de Medeiros, A. Guzzo, G. Greco, W.M.P. van der Aalst, A.J.M.M. Weijters, B.F. van Dongen, and D. Sacc. Process mining based on clustering: A quest for precision. In A.H.M. ter Hofstede, B. Benatallah, and H.Y. Paik, editors, *Business Process Management Workshops*, volume 4928 of *Lecture Notes in Computer Science*, pages 17–29. Springer Berlin Heidelberg, 2008. 15, 16

[31] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, mar 1970. 20

[32] OMG. *Business Process Model and Notation (BPMN)*. Object Management Group, dtc/2010-06-05 edition, 2010. 8

[33] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. 26, 28, 58, 59

[34] M. Rosemann and W.M.P. van der Aalst. A configurable reference modelling language. *Information Systems*, 32(1):1 – 23, 2007. 9

[35] A. Rozinat and W.M.P. van der Aalst. Decision mining in business processes. *BPMNCenter Report BPM-06-10*, 2006. 25

[36] D.M.M. Schunselaar, E. Verbeek, W.M.P. van der Aalst, and H.A. Raijers. Creating sound and reversible configurable process models using cosenets. In W. Abramowicz, D. Kriksciuniene, and V. Sakalauskas, editors, *Business Information Systems*, volume 117 of *Lecture Notes in Business Information Processing*, pages 24–35. Springer Berlin Heidelberg, 2012. 9, 33

[37] M. Song, C.W. Günther, and W.M.P. van der Aalst. Trace clustering in process mining. In D. Ardagna, M. Mecella, and J. Yang, editors, *Business Process Management Workshops*, volume 17 of *Lecture Notes in Business Information Processing*, pages 109–120. Springer Berlin Heidelberg, 2009. 16

# Appendices

# Appendix A

# Partially ordered alignment construction algorithm

Algorithm 1 shows the construction algorithm for partially ordered alignments. The algorithm requires as input a sequential alignment and a process tree.

```
Data: Sequential Alignment a, Process Tree t
Result: Partially ordered alignment
Store parent nodes in the tree for every non-log move
for move ← non-move on log only ∈ a do
    parentNodes[move] ← t.parents(move.treeNode);
end
Estimate an insertion point for the log-moves. Always prefer insertion under a ∧-operator or
∨-operator. This makes sure that our assumption on the insertion point is the weakest possible
(when only considering the parents of the first non-log left and right move).
for move ← move on log only ∈ a do
    nextMove ← a.nextNonLog(move);
    previousMove ← a.prevNonLog(move);
    if nextMove exists then
        Retrieve the first parent in which we can insert the move
        commonParent ← t.firstInsertableParent(nextMove.treeNode);
        if t.nodeType(commonParent) is ∧ or ∨ then
            Store commonParent as a parent and all parents of commonParent
            parentNodes[move] t.parentTillIncluding(commonParent);
            continue;
        end
    end
    if previousMove exists then
        commonParent ← t.firstInsertableParent(previousMove.treeNode);
        parentNodes[move] ← t.parentTillIncluding(commonParent);
        continue;
    end
    if t.numChildren(t.root) == 0 then
        parentNodes[move] ← [];
    else
        parentNodes[move] ← [t.root];
    end
end
Now create a successor relation that represents the POA.
for i ← a.length; i >= 0; i ← i - 1 do
    successors[i] ← [];
    for succ ← a.length; succ > i; succ ← succ - 1 do
        commonParent ← seekFirstCommonParent(i, succ, parentNodes);
        if commonParent exists then
            if t.nodeType(commonParent) is ∧ or ∨ then
                break;
            end
            successors[i].push(succ);
            successors[i].removeAll(impliedSuccessors(succ, successors));
        end
    end
end
Result ← POA(successors);
```

**Algorithm 1:** Algorithm to construct a partially ordered alignment

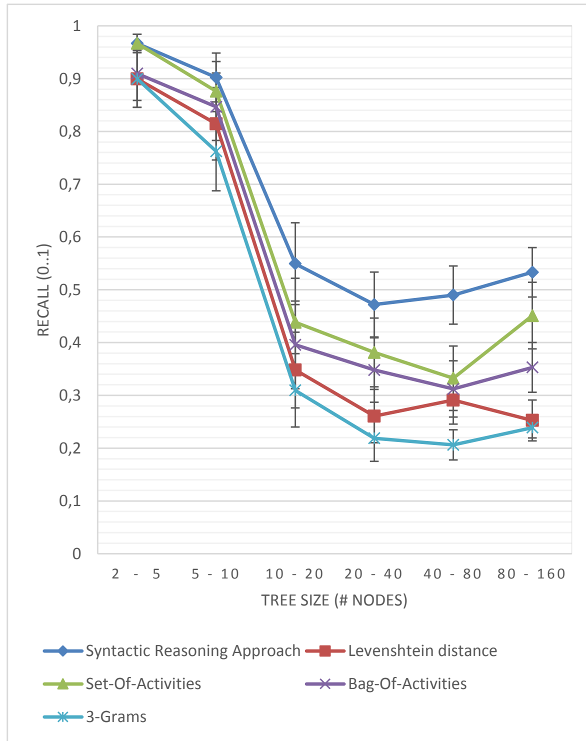
# Appendix B

# Additional experimental evaluation results

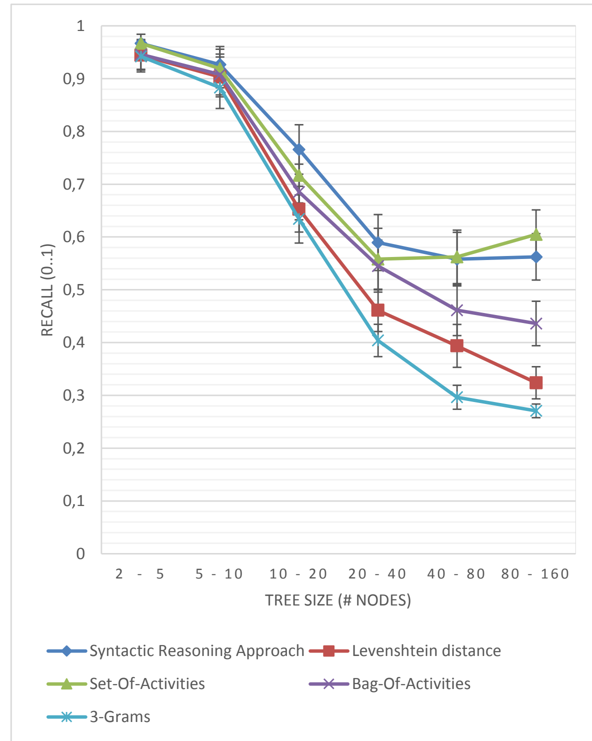
In this appendix we present additional experimental evaluation results. In Section B.1 we show the recall and precision results for the $F_1$ scores as discussed in Section 7.2. In Section B.2 we present additional results for the full approach experiments as discussed in Section 7.4.

## B.1 Comparison of dissimilarity measures

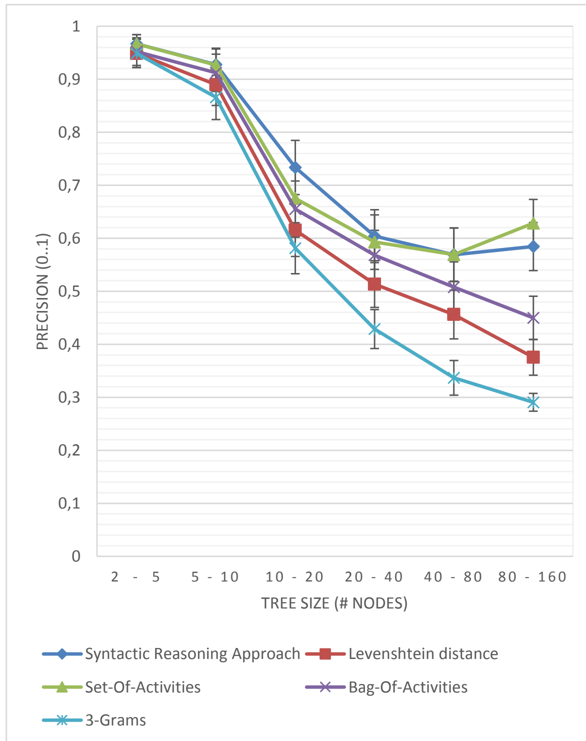
In this section we present the recall and precision results for the dissimilarity measure experiments as discussed in Section 7.2. Figures B.1, B.2, B.3 and B.4 respectively show the recall and precision scores, for the worst and best cut, of the cases: normal, loop iterations, order of execution and parallelism punishment.



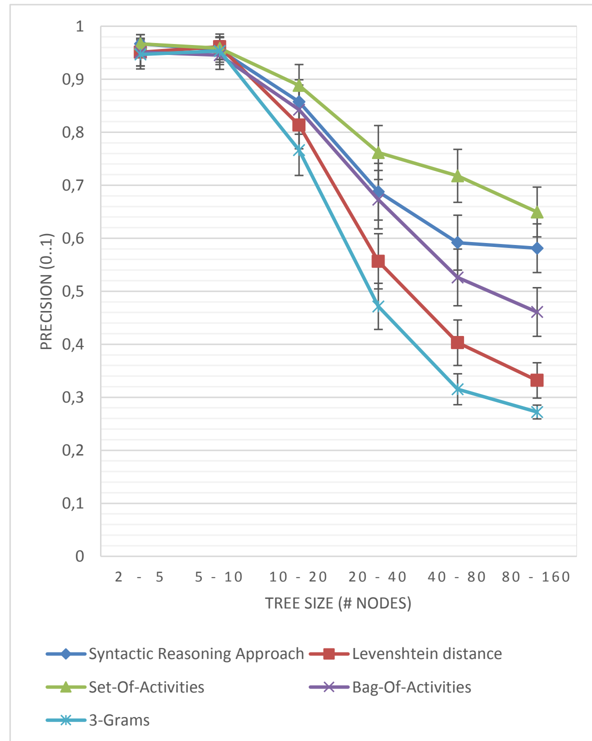

(a) Worst cut - Recall.




(b) Best cut - Recall.

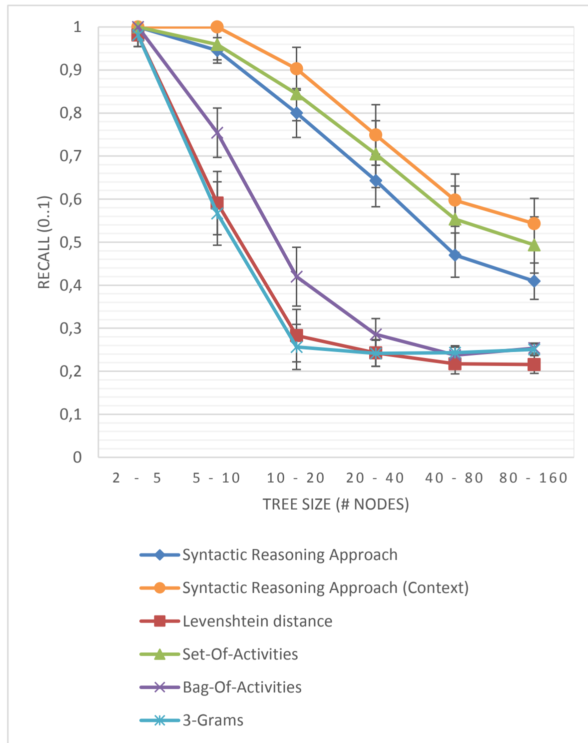



(c) Worst cut - Precision.

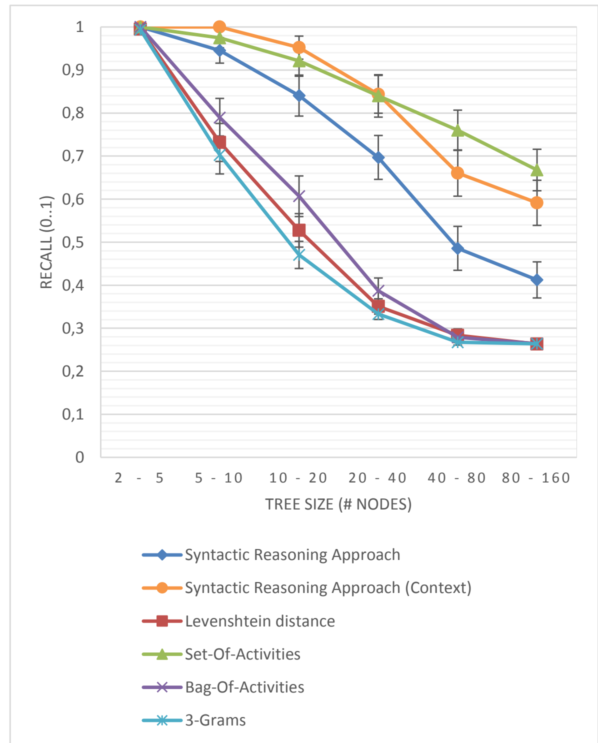



(d) Best cut - Precision.

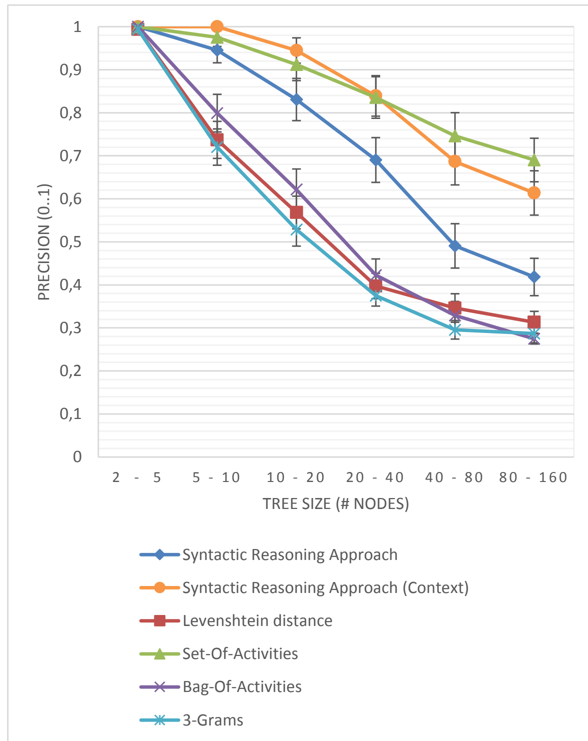
Figure B.1: The recall and precision scores with 95% confidence intervals of the best and worst cut in the normal case.



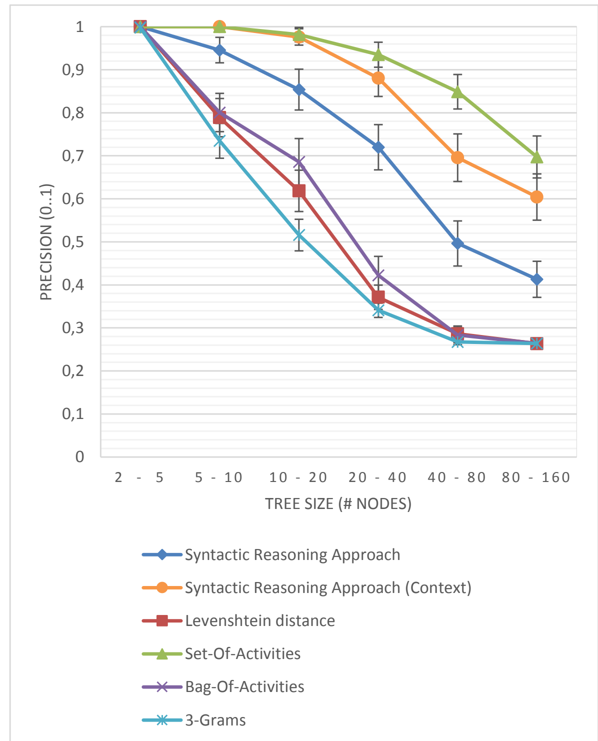

(a) Worst cut - Recall.




(b) Best cut - Recall.

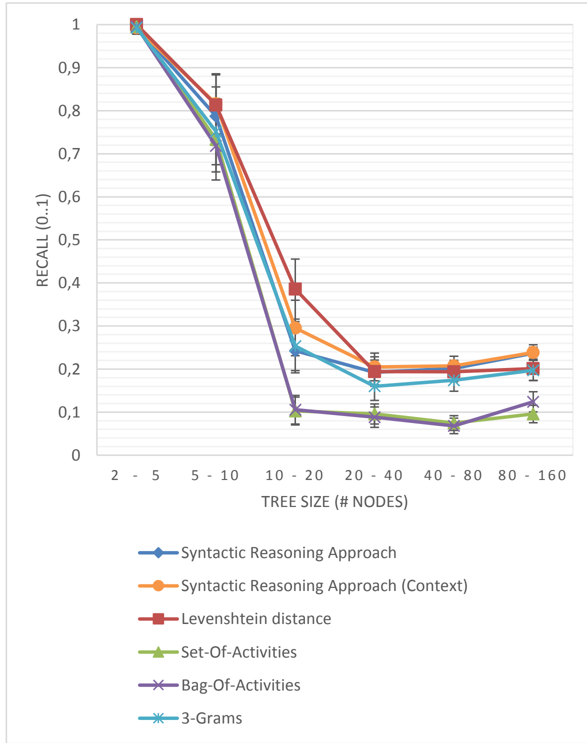



(c) Worst cut - Precision.

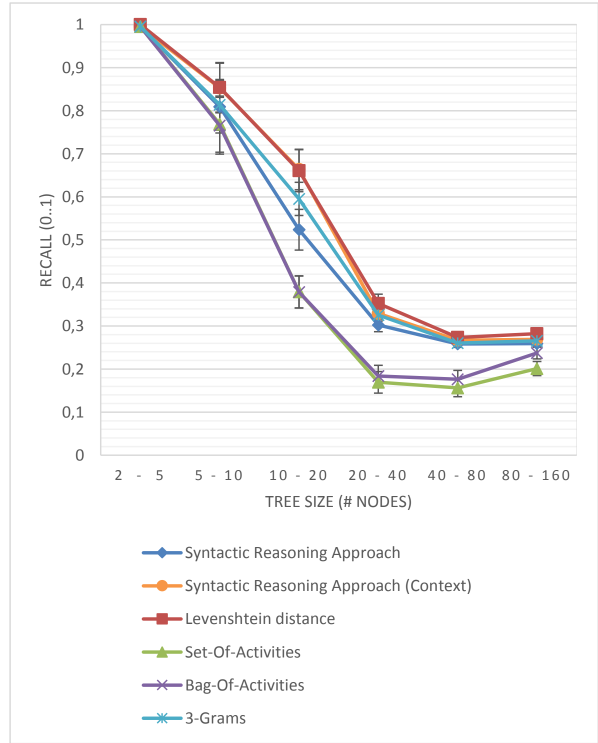



(d) Best cut - Precision.

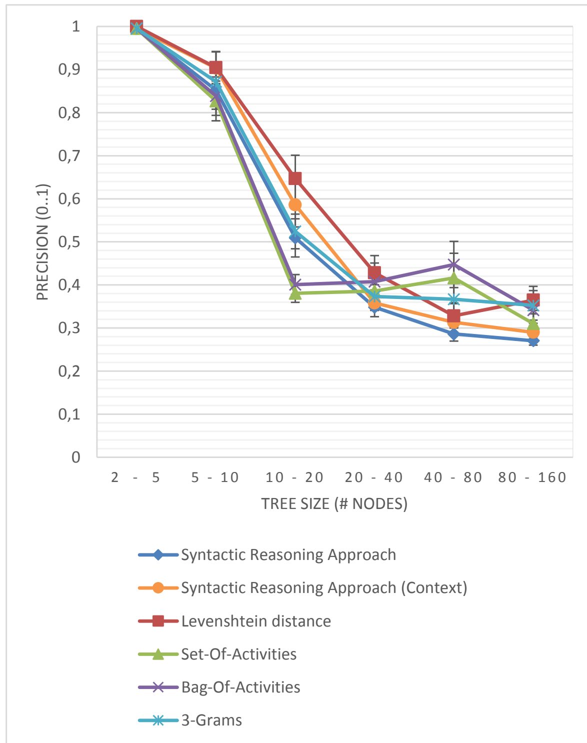
Figure B.2: The recall and precision scores with 95% confidence intervals of the best and worst cut in the loop iterations case.



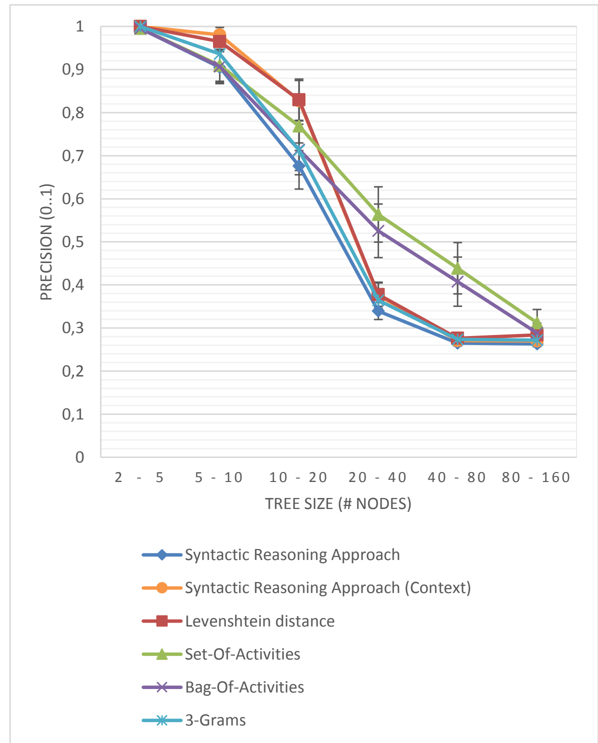

(a) Worst cut - Recall.




(b) Best cut - Recall.

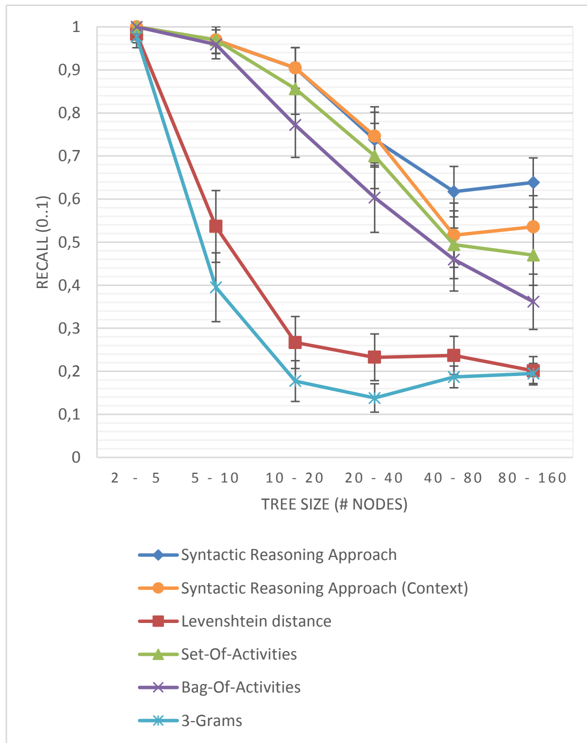



(c) Worst cut - Precision.

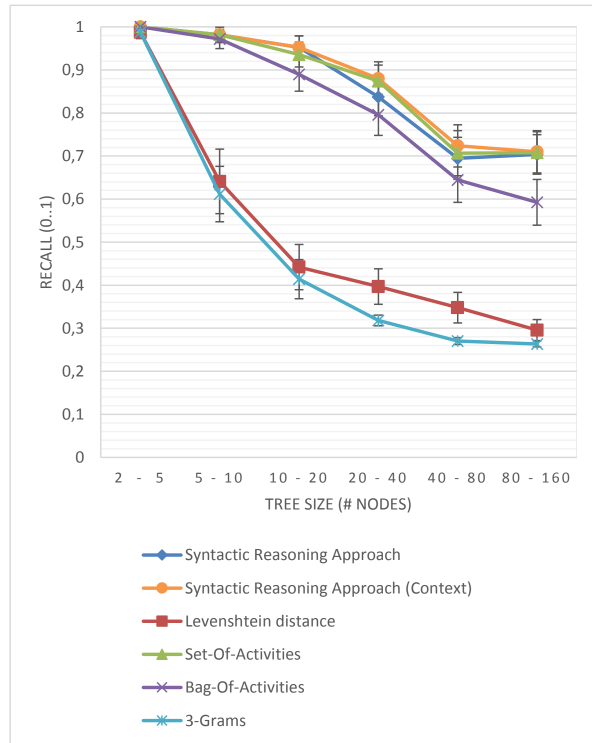



(d) Best cut - Precision.

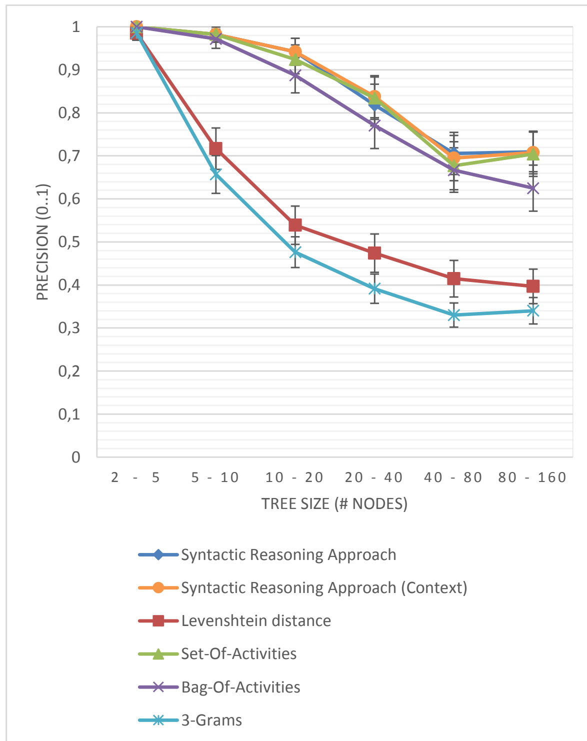
Figure B.3: The recall and precision scores with 95% confidence intervals of the best and worst cut in the order of execution case.



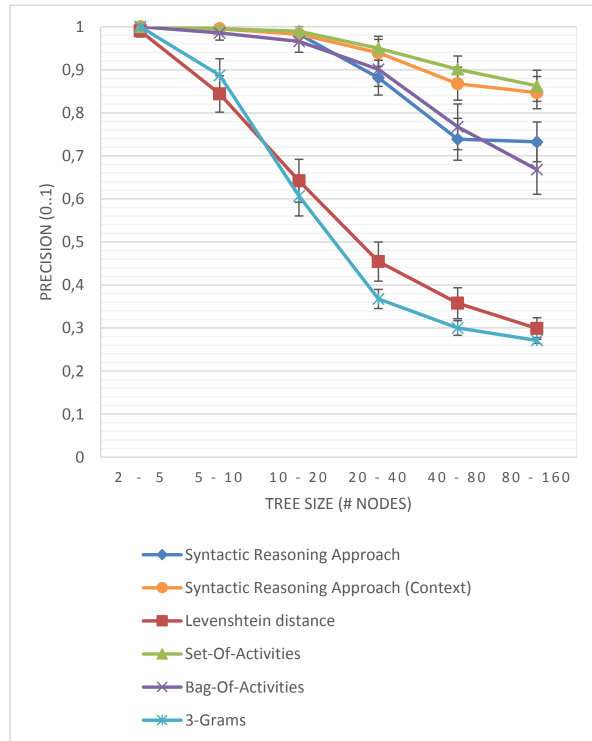

(a) Worst cut - Recall.




(b) Best cut - Recall.




(c) Worst cut - Precision.




(d) Best cut - Precision.

Figure B.4: The recall and precision scores with 95% confidence intervals of the best and worst cut in the parallelism punishment case.

## B.2 Evaluation of the full approach

In this section we present additional results for the experiments discussed in Section 7.4. Section B.2.1 shows additional results for the full approach experiment of the running example (Section 7.4.1) and Section B.2.2 shows additional results for the full setup experiment of the BPI challenge logs 2012 (Section 7.4.2).

### B.2.1 Running example

Figures B.5, B.6, B.7, B.8 and B.9 respectively show the hierarchical clusterings of the Syntactic reasoning, Levenshtein distance, Set-Of-Activities, Bag-Of-Activities and 3-Grams approaches.

Figure B.5: Hierarchical clustering of the running example using the Syntactic reasoning approach.

Figure B.6: Hierarchical clustering of the running example using the Levenshtein distance.

### B.2.2 BPI challenge logs 2012

Figure B.10 shows the full hierarchical clustering of the filtered BPI challenge log of 2012 which includes sending of offers and checks for fraud.

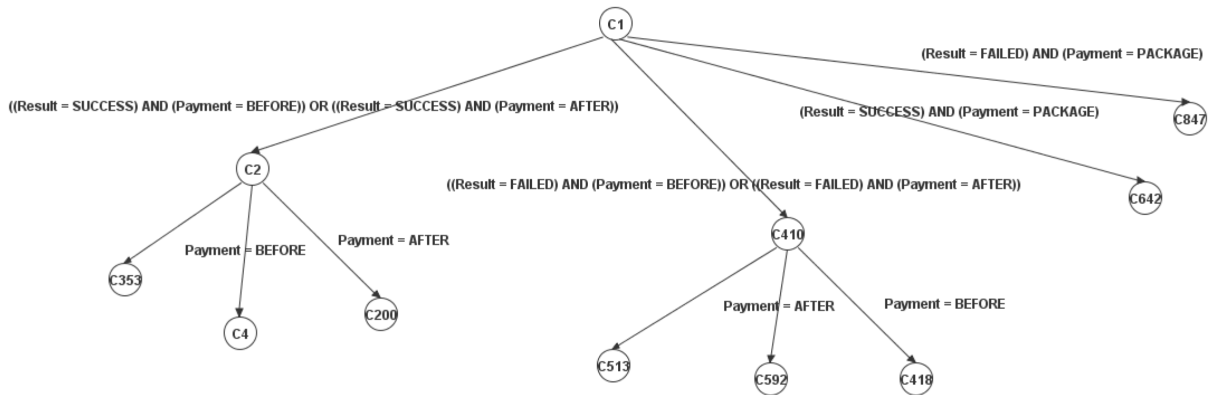



Figure B.7: Hierarchical clustering of the running example using Set-Of-Activities.

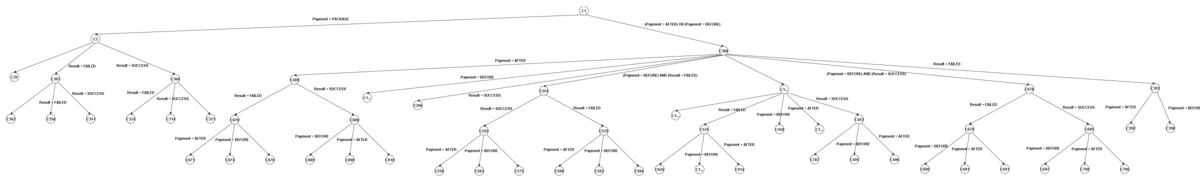


Figure B.8: Hierarchical clustering of the running example using Bag-Of-Activities.

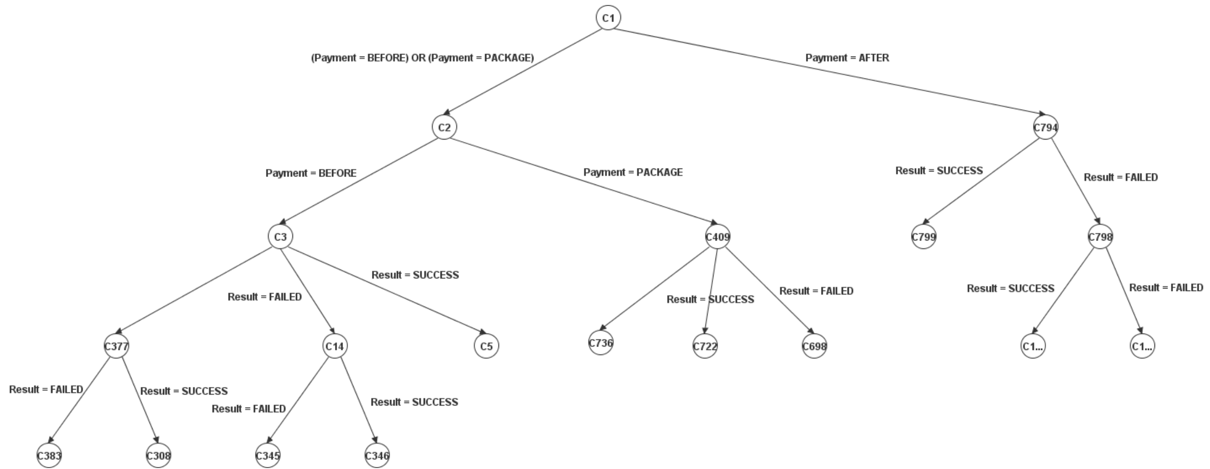



Figure B.9: Hierarchical clustering of the running example using 3-Grams.

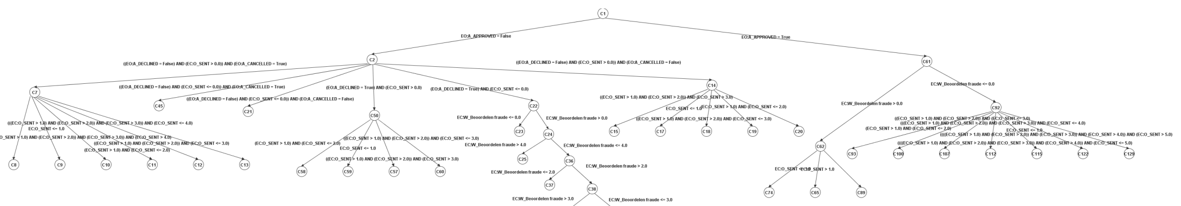


Figure B.10: The full hierarchical clustering of the filtered BPI challenge log of 2012 which includes sending of offers and checks for fraud.