Alignment-based process model repair and its application to the evolutionary tree miner

van Eck, M.L.

Award date:
2013
Alignment-based Process Model Repair and its Application to the Evolutionary Tree Miner

Master Thesis

Maikel van Eck, BSc

Supervisors:
dr. ir. B.F. van Dongen
dr. G.H.L. Fletcher
dr. A.M. Wilbik

Eindhoven, December 2013
Abstract

Process mining is an emerging research discipline that provides techniques that can be used to discover, monitor and improve real processes using event data. In this thesis we present several approaches and extensions that improve the effectiveness of the Evolutionary Tree Miner, a genetic process mining algorithm. These approaches and extensions enable the Evolutionary Tree Miner to make smart changes to process models, in order to obtain models of a higher quality in less time than the original implementation, while taking into account the four process model quality dimensions of replay fitness, simplicity, precision and generalisation. The approaches and extensions are based on concepts and ideas from process model repair, which have been applied in the context of the Evolutionary Tree Miner. We show, through experiments on both artificial and randomly generated event logs, that our approach is superior to the original implementation of the Evolutionary Tree Miner in its ability to quickly produce high quality models.
Preface

First of all, I would like to thank my supervisor Boudewijn van Dongen for his guidance and feedback during my graduation project, and for his ideas that inspired the work I present in this thesis. Furthermore, I would like to thank Joos Buijs for his help with the practical side of working on the Evolutionary Tree Miner, for his willingness to adapt the original ETM (and for fixing the occasional bug) that allowed me to implement my own extensions, and for the \LaTeX{} template that was used for this thesis.

I would also like to thank all the people that worked with me on projects and courses over the years. Special thanks go out to Xixi, for all the conversations and discussions we had, for the amazingly detailed feedback on my thesis, and for all the good times. A big thank you also goes out to my best friends, Ralph and Werner, and to the gentlemen from my Tuesday night gaming group, and Rom in particular for also giving me feedback on my thesis. Thanks for keeping me sane during my graduation project!

Finally, I would like to thank my parents and my brother for all their support during my studies. Without you, I would not have made it this far.

Maikel van Eck
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Preface</td>
<td>v</td>
</tr>
<tr>
<td>Contents</td>
<td>vii</td>
</tr>
</tbody>
</table>

### 1 Introduction
1.1 Process Model Quality | 1
1.2 Repairing Process Models | 2
1.3 Research Goal | 3
1.4 Thesis Overview | 3

### 2 Alignment-based Process Tree Repair
2.1 Process Trees | 5
2.2 Measuring the Quality of a Process Tree | 6
2.3 Running Example | 7
2.4 Aligning Process Models and Event Logs | 8
2.5 Identifying Problem Areas within Process Trees | 9
2.6 Repairing Problem Areas within Process Trees | 10
   2.6.1 Removing Behaviour | 10
   2.6.2 Adding Behaviour | 11
   2.6.3 Changing Behaviour | 12

### 3 Extending the Evolutionary Tree Miner
3.1 The Evolutionary Tree Miner | 15
3.2 Alignment-based Crossover | 16
3.3 Alignment-based Mutation | 17
3.4 Initial Population Creation | 18
   3.4.1 Trace-model Creation | 18
   3.4.2 Advanced Trace-model Creation | 19

### 4 Experimental Evaluation
4.1 Testing on Artificial Processes | 25
   4.1.1 Crossover and Mutation Test | 26
   4.1.2 Initial Population Creation Test | 34
4.2 Testing on Random Processes | 46
   4.2.1 Testing on Models of Varying Size | 46
   4.2.2 Testing on Event Logs with Varying Noise Levels | 47
CONTENTS

5 Future Work 53
  5.1 Identifying Problem Areas within Process Trees 53
  5.2 Repairing Problem Areas within Process Trees 54
  5.3 Evolutionary Tree Miner Improvements 55
    5.3.1 Crossover 55
    5.3.2 Mutation 56
    5.3.3 Initial Population Creation 56
  5.4 Additional Experimental Evaluation 57

6 Conclusions 59

Bibliography 61

Appendix 63

A Additional Evaluation Results 63
  A.1 Trace-model creation 63
  A.2 Combining Mutation and Crossover 66
  A.3 Varying Noise Levels 69

B Artificial Models and Event Logs 73
  B.1 Driver’s License Model 73
  B.2 Insurance Claim Model 73
  B.3 Simple Loop-choice Model 73
  B.4 Double Loop-choice Model 73
Chapter 1

Introduction

Over the years, more and more data is being recorded and stored in information systems. However, for many organisations it is still a challenge to extract value from this data. The goal of process mining is to provide this value, by using the recorded data to obtain process-related information. One example of this is the use of process mining to automatically discover a process model by looking at events recorded within an enterprise information system. In general, Process mining is an emerging research discipline that provides techniques that can be used to discover, monitor and improve real processes using event data [2].

The data that is used as the input for process mining is a so-called event log, which is a collection of sequentially recorded events that occurred during the execution of a process [2, 8]. Each event refers to an activity (i.e., a well-defined step in the process) and is related to a particular case (i.e., a process instance). A sequence of events belonging to a single case is called a trace. Event logs may also store additional information, such as the resource (i.e., person or device) executing or initiating an activity, the timestamp of an event or other attributes associated with an event (e.g., the size of an order).

Usually, three different types of process mining are distinguished [2, 8]. The first type is process discovery, where a process model is created using only the behaviour observed in an event log. The second type of process mining is that of conformance checking, where an existing process model is compared with an event log of the same process. This comparison shows where the execution of the process deviates from the process model. The third type is process enhancement, where a process model is extended or improved using information obtained from the event log. Combined, these process mining techniques can be used to check compliance, analyse bottlenecks, predict delays, and recommend actions to minimise the expected flow time.

In this thesis we touch upon all three areas of process mining, but we restrict ourselves to the control-flow perspective (i.e., the ordering of activities within a process [2]).

1.1 Process Model Quality

Process models describe the behaviour and possible executions of a process, and they can be used for various purposes such as verification, performance analysis, specification and documentation, or simply to gain insight into a process and to promote discussions amongst the process stakeholders [2]. Over the past years, many different process discovery algorithms have been developed that can automatically produce such process models for a given event log. These algorithms and techniques have been compared and evaluated through case-studies and by looking at the properties of the models produced [2, 9].

The following four quality dimensions are generally considered when discussing the results of a process discovery technique, and they each cover a different aspect of the quality of a process model [2, 8, 11, 12]:

**Replay fitness** This dimension quantifies the extent to which the discovered process model can
correctly reproduce the cases recorded in the event log. A model that does not have a perfect replay fitness may contain activities that are sometimes skipped in the event log, the log may contain events that are not described by the model, or activities may be executed in a different order in the event log than described by the model.

**Simplicity** The simplicity dimension refers to Occam’s Razor, which in this context means that the best model is the simplest model that can explain the behaviour seen in the event log.

**Precision** This dimension measures whether the model prohibits behaviour that was not seen in the event log. A model is precise if it does not “underfit” (i.e. the model does not allow for a lot of unseen behaviour).

**Generalisation** The generalisation dimension indicates to what extent the model will be able to correctly reproduce future, not yet observed, behaviour of the process. A model is general if it does not “overfit” (i.e. the model does not only explain the cases in the sample event log, but it explains the process that generated the event log).

Unfortunately, no known process mining algorithm always scores perfectly on all quality dimensions for every possible event log. However, conformance checking techniques can be used to detect and diagnose differences between modelled behaviour and behaviour observed in the event log [2]. Due to the close relation between conformance checking and the measuring of the replay fitness quality dimension, and to a lesser extent the measuring of the precision and generalisation quality dimensions, this means that it may be possible to identify the parts of a process model that are of suboptimal quality [13]. Once these parts have been identified it is possible to repair these specific parts of the process model, as a form of process enhancement, to increase the process model quality [2] [13].

### 1.2 Repairing Process Models

Repairing process models based on observed behaviour from an event log is a new area of process mining research. Although there are many conformance checking techniques that detect deviations between an event log and a process model, there are far fewer approaches that can automatically repair a model based on the results of conformance checking techniques in order to conform to an event log and improve the quality of the process model [10] [13].

The work of Fahland et. al. [13] focusses on repairing deviations between the behaviour allowed by a process model and the behaviour observed in an event log. These deviations are detected by the creation of *alignments* between the process model and traces from the event log, which is essentially the computation of the replay fitness of the model [5] [6]. Based on this information, the event log is decomposed into several smaller logs containing the nonfitting subtraces. A subprocess is then mined for each sublog using a process discovery algorithm and these are then added to the original model at the right location in order to repair the model.

A limitation of the approach of Fahland et. al. is that replay fitness is the only quality dimension that is explicitly taken into account when identifying the parts of the process model that can be improved. Furthermore, the assumption is made that a domain expert first filters out the noise from the event log, because the approach cannot distinguish exceptional behaviour that should not be modelled from behaviour that is not modelled while it should be. If the event log does contain noise, then the noisy behaviour will also be added to the process model.

The approach taken by Buijs et. al. [10] is based on the genetic process discovery algorithm presented in [8], the *Evolutionary Tree Miner* (ETM). The ETM algorithm works by creating a *population* of models that are then evaluated according to the four quality dimensions described above in Section 1.1. If none of the stop criteria have been satisfied then the models in the population are changed through random modifications and evaluated again. This is repeated until one of the stop criteria has been satisfied, in which case the best scoring model will be returned.

The ETM algorithm was extended in [10], so that the initial population was seeded with the reference model being repaired and a new quality dimension was added that measures the
structural similarity between the reference model and the new models being evaluated. In this way it is ensured that the reference model is improved with respect to all four quality dimensions, while staying as similar as possible to the original model.

Although the approach by Buijs et. al. does take into account all four quality dimensions when repairing a process model, it is not able to explicitly distinguish exceptional behaviour that should not be modelled from behaviour that is not modelled while it should be. This means that a domain expert is still needed to inspect the improved model in order to validate the results. However, genetic process mining algorithms such as the ETM algorithm are robust to noise [2], so noisy behaviour has a lower probability to be added to the process model while it is being repaired.

A bigger issue is that genetic process mining is not very efficient for large models and logs [2]. As indicated in [13], calculating the conformance of a model takes significantly longer than repairing the actual model once deviations have been identified. The reason for this is that it is computationally complex to calculate the alignments needed to determine the replay fitness. This is not much of an issue for the approach by Fahland et. al., but in the ETM algorithm the replay fitness needs to be recalculated each time that a model changes, and many different changes need to be tested in order to find the best way to repair a model because the changes are made randomly.

### 1.3 Research Goal

The goal of the research described in this thesis is to improve the ETM algorithm and enable it to make smart changes to models, in order to obtain models of a higher quality in less time than before, while taking into account all four quality dimensions. We do not just focus on repairing process models because the ETM algorithm can also be used as a process discovery algorithm. The main difference between discovering a process model with a genetic process mining algorithm like the ETM algorithm and repairing an existing model is the initial population of models, but the operations that change a model can be used for both genetic process mining and repair [2].

To speed up the ETM algorithm for both process discovery and repair, we want to reduce the number of computationally expensive process model quality calculations that are being executed. To achieve this and to increase the quality of the results, we need a good starting point for the initial population when doing process discovery and we need an approach similar to the approach of Fahland et. al. [13] in order to be able to make smart changes to a process model. This means that we answer the following research questions in this thesis:

**Process Model Repair**

- How can we **identify** the parts of a process model that have a low quality, so that we can improve those parts instead of trying to improve parts that are already good?
- How can we **improve** the parts of a process model that have a low quality, so that we can create models of a higher quality?

**Evolutionary Tree Miner**

- How can we **apply** the concepts and ideas from process model repair, so that we can **improve the effectiveness** of the Evolutionary Tree Miner?
- How can we **create** an initial population of process models that already have a **reasonable quality**, so that we need few changes to reach a model with a high quality?

### 1.4 Thesis Overview

In the previous sections we introduced the concepts related to the topic of process mining and stated the research questions that are answered in this thesis. In this section, an overview is presented of the remainder of this thesis.
In Chapter 2, we answer the first two research questions by showing how we identify and subsequently change the parts of a process model that have low quality. We focus here on the replay fitness quality dimension because that is generally considered to be the most important quality dimension.

In Chapter 3, we answer the last two research questions. We discuss how the concepts and ideas from process model repair can be applied to improve the Evolutionary Tree Miner and how these improvements were implemented as crossover and mutation operations. We also show how we create a good initial population of process models for the ETM algorithm when it is used for process discovery.

The experimental evaluations discussed in Chapter 4 were performed to test the effectiveness of the extensions to the ETM algorithm. The results show that the extended ETM algorithm performs significantly better than the original ETM algorithm.

In Chapter 5, an overview is presented of the things that may further improve the effectiveness of the ETM algorithm and potentially interesting areas of future research related to the content of this thesis.

Finally, the thesis is concluded in Chapter 6.
Chapter 2

Alignment-based Process Tree Repair

In this chapter we answer the first two research questions and show how we identify and change the parts of a process model that have a low quality. In Section 2.1 we first explain the process tree modelling notation that is used in this thesis and then in Section 2.2 we show how the quality of a process tree can be calculated. In Section 2.3 we then introduce the running example that is used to explain our approach. Following this, we explain in Section 2.4 what an alignment between a process model and a trace from the event log is. These alignments can be used to identify the areas of a process model that can be improved with respect to the fitness replay quality dimension and we show how this information is extracted from the alignments in Section 2.5. Finally, in Section 2.6 we describe how to change a process model in order to repair the problems that are identified using the alignments.

2.1 Process Trees

There exist many different process model notations that can be used to visualise and represent process models. Some examples of notations that show the control-flow perspective of the processes they describe include the Business Process Model and Notation (BPMN) [10], Petri nets [12] and Event Driven Process Chains (EPC) [1].

As a consequence of using the ETM algorithm, we use process trees in the remainder of this thesis to describe our process models and how to modify them, because this is the representation that is used by the ETM algorithm internally. The reason why process trees are used by the ETM algorithm is that traditional modelling languages allow the creation of models that are not sound, i.e. these models contain deadlocks, livelocks and other anomalies. Process trees, however, are guaranteed to represent sound process models, which reduces the search space of the ETM algorithm because unsound models do not have to be considered [8, 9].

Figure 2.1 shows the possible operators that process trees can be composed of and their translations to BPMN models. Operator nodes specify the relation between their children and the five operator types are: sequential execution (→), parallel execution (∧), exclusive choice (×), non-exclusive choice (∨) and repeated execution (⊇). Children of an operator node can again be operator nodes or they can be leaf nodes that represent the execution of an activity. The order of the children matters for the sequence and loop operators. The order of the children of a sequence operator specifies the order in which the children are executed (from left to right). Loop nodes (⊇) always have three children. The left child is the ‘do’ part of the loop and after its execution either the middle child, the ‘redo’ part, may be executed or the right child, the ‘exit’ part, may be executed. After the execution of the ‘redo’ part the ‘do part is again enabled and the ‘exit’ part is disabled. Process trees can also contain leaf nodes labelled with a τ, which indicates an unobservable activity.
CHAPTER 2. ALIGNMENT-BASED PROCESS TREE REPAIR

Figure 2.1: An overview of the different process tree operators and their relation to BPMN constructs.

2.2 Measuring the Quality of a Process Tree

The metrics used in this thesis that measure the four quality dimensions introduced in Section 1.1 are the same as those presented in [9], except for the simplicity dimension for which Buijs has created a new metric. The general idea behind each metric is presented below, but the precise formulae that compute these metrics are not presented here because they are not very important for the results in this thesis.

Replay fitness This dimension quantifies the extent to which the discovered process model can correctly reproduce the cases recorded in the event log. The alignment-based fitness computation defined in [5] is used to compute the replay fitness of a process tree. Alignments are explained in more detail in Section 2.4. In short, the technique aligns as many events as possible from each trace in the event log with activities in an execution of the model. If necessary, activities are skipped or events from the log are inserted into the execution, and penalties are given for skipping and inserting activities. The replay fitness metric is calculated as follows:

\[
Q_{rf} = 1 - \frac{\text{cost for aligning model and event log}}{\text{Maximal cost to align the event log on the model}}
\]

where the denominator is defined as the minimal cost when no match can be found between the events from the event log and the nodes in the process model, i.e. the worst case scenario.

Simplicity The simplicity dimension quantifies the complexity of the model. The metric that is used for this dimension punishes for useless nodes in the process tree. Useless nodes are for example operator nodes with only a single child and \( \tau \)-nodes (i.e. unobservable activities) that have a \( \rightarrow \)-operator or \( \land \)-operator as their parent. Simplicity is calculated as follows:

\[
Q_s = 1 - \frac{\#\text{useless nodes}}{\#\text{nodes in tree}}
\]
CHAPTER 2. ALIGNMENT-BASED PROCESS TREE REPAIR

**Precision** This dimension measures whether the model prohibits behaviour that was not seen in the event log. This is calculated by counting the so-called escaping edges (i.e. decisions that are possible in the model, but never made in the log) in the state space of the tree execution. If there are no escaping edges then the precision is perfect. We obtain the part of the statespace that is used from information provided by the replay fitness, where we ignore events that are in the log, but do not correspond to an activity according to the alignment. The precision is calculated as follows:

\[
Q_p = 1 - \frac{\sum_{\text{visited markings}} \text{#visits} \times \frac{\text{#outgoing edges} - \text{#used edges}}{\text{#outgoing edges}}}{\text{#total marking visits over all markings}}
\]

**Generalisation** The generalisation dimension indicates to what extent the model will be able to correctly reproduce future, not yet observed, behaviour of the process. The metric that is used to measure this considers the frequency with which each node in the tree needs to be visited if the model is to produce the given log. For this we use the alignments provided by the replay fitness. If a node is visited more often then we are more certain that its behaviour is (in)correct. If some parts of the tree are very infrequently visited, generalisation is bad. Therefore, generalisation is calculated as follows:

\[
Q_g = 1 - \frac{\sum_{\text{nodes}} (\sqrt{\text{#executions}})^{-1}}{\text{#nodes in tree}}
\]

All four metrics are computed on a scale of 0 to 1, where 1 is the optimal value. It is possible to discover models with a score of 1 for replay fitness, precision or simplicity, but generalisation can only reach a score of 1 in the limit of the number of visits to each node. The overall quality of a process tree is determined by a weighted sum of all four metrics, with a higher weight for the replay fitness because that dimension is more important than the other quality dimensions [9, 13].

### 2.3 Running Example

In this thesis we are using a process describing how people buy lunch at a canteen as a running example to explain how to repair process trees. This process is shown in Figure 2.3 as a BPMN process model and in Figure 2.3 as a process tree. The process starts when a person enters the canteen. This person will then, in any possible order, get the drink they want, get a sandwich and make the choice whether to get a snack or not. Following this, the person can choose to pay either with cash or by credit card and the process ends after the person has left the canteen. An example of a trace in an event log generated by this process is: { Enter canteen (A), Get drink (B), Get sandwich (C), Pay cash (E), Exit canteen (G) }.

![Figure 2.2: A simple BPMN process model describing how people buy lunch at a canteen, which is used as a running example in the rest of this thesis.](image-url)
CHAPTER 2. ALIGNMENT-BASED PROCESS TREE REPAIR

Figure 2.3: A process tree describing the same process as Figure 2.2. The activities are labelled as follows: A = Enter canteen, B = Get drink, C = Get sandwich, D = Get snack, E = Pay cash, F = Pay credit card, G = Exit canteen, and τ being an unobservable activity.

2.4 Aligning Process Models and Event Logs

Conformance checking techniques can be used to identify how the process executions that are stored in an event log deviate from the behaviour allowed by a process model [2]. One such conformance checking technique is the starting point in the approach by Fahland et. al., which automatically repairs the deviations found by the conformance checking technique [13]. This conformance checking technique aligns an event log \( L \) and a process model \( M \) (i.e. relates events in the event log to model elements and vice versa), so that a minimal set of changes can be identified such that the deviations are eliminated and the log can be replayed by the model [5].

In order to find the minimal set of changes so that the log \( L \) can be replayed on the model \( M \), we need to find an execution sequence \( \sigma \) of \( M \) for each trace \( l \in L \) such that \( \sigma \) and \( l \) are as similar as possible. When aligning \( \sigma \) and \( l \) we may find activity executions in \( \sigma \) that are not part of \( l \) and we call such executions model moves. We may also find events in \( l \) that are not part of \( \sigma \) and we call the execution of such events log moves. Finally, we may find activity executions in \( \sigma \) that match events found in \( l \) and we call such executions synchronous moves.

For example, assume that we have the trace \( l = \langle BCEFG \rangle \) that we are trying to align to the execution sequence \( \sigma = \langle ABC\tau EG \rangle \) of the model in Figure 2.3. The activity A in \( \sigma \) does not occur in \( l \), which means that if we want to align our trace \( l \) with execution sequence \( \sigma \) then we need to be able to skip the execution of A. Such a skipped activity is a model move. The symbol \( \tau \) is used to indicate the execution of an unobservable activity in the model, which is never found in a trace in the event log by definition. The event F in \( l \) does not occur in \( \sigma \), so we need to insert the execution of F in our alignment. Such an insertion is a log move. Finally, the activities B, C, E and G are included in both \( l \) and \( \sigma \), so the executions of these activities are synchronous moves because they are executed both in the log and on the model. An alignment between \( l \) and \( \sigma \) is also shown graphically in Figure 2.4.

Figure 2.4: An alignment between trace \( l = \langle BCEFG \rangle \) and execution \( \sigma = \langle ABC\tau EG \rangle \). The symbol \( \perp \) is used to indicate that there is no matching move in the other sequence.

An approach was presented in [5] that automatically finds an execution sequence \( \sigma \) of a model \( M \) for each trace \( l \in L \) such that the alignment of \( \sigma \) and \( l \) has a minimal number of skips and insertions, based on a cost function that specifies the cost for log moves, model moves, synchronous moves and \( \tau \)-moves (i.e. executions of unobservable activities). These optimal alignments specify exactly how the traces deviate from the behaviour allowed by the model and what minimal set of changes is needed in order to be able to replay each trace on the model.

8 Alignment-based Process Model Repair and its Application to the Evolutionary Tree Miner
2.5 Identifying Problem Areas within Process Trees

In this section we show how to identify the parts of a process tree that have a negative effect on the quality of the model. We focus here on the replay fitness dimension because it is generally considered to be the most important quality dimension [9, 13] and because it has a well defined metric in the form of alignments between the event log and the process model. The precision quality dimension is addressed in Section 2.6 when we discuss the actual repair of a process tree. The other two quality dimensions of generalisation and simplicity are often in conflict with replay fitness and precision [2, 13], so we do not address these dimensions here and instead rely on the ETM algorithm to balance all four quality dimensions based on the weight function.

The alignments described in Section 2.4 provide local information on the parts of the model that are nonconformant to the event log, because they identify activities that should be skipped or inserted at a specific point during model execution. In Figure 2.5 we show three things: (1) an example of a process tree that we are repairing so that it correctly describes the process shown in Figures 2.2 and 2.3, (2) an optimal alignment between the trace \( l = \langle ABCGE \rangle \) and the incorrect process tree, and (3) a mapping of the moves in the alignment onto the nodes of the process tree such that it becomes clear what the problem areas within the process tree are.

The interpretation of the alignment move mapping shown in Figure 2.5 is that model and log moves indicate problems that need to be repaired and the mapping shows where these problems are located in the process tree. We can repair the process tree by making activities onto which model moves are mapped skippable, while activities onto which log moves are mapped can be repaired by inserting additional activities before or after those activities. The moves in the alignment are mapped onto the activities from the process tree in the following way: both synchronous moves (e.g. \( A \)) and model moves (e.g. \( D \)) are mapped onto their corresponding activities in the process tree, while log moves (e.g. \( E \)) are mapped onto the last activity executed on the model before the log move (e.g. activity \( B \) for log move \( E \)) and onto the first activity executed on the model after the log move (e.g. activity \( F \) for log move \( E \)).

![Figure 2.5](image)

Figure 2.5: An optimal alignment between the trace \( l = \langle ABCGE \rangle \) and a process model in need of repair, together with a mapping of the alignment onto model elements. Synchronous moves are mapped onto leaf nodes in green, while model moves are mapped in blue and log moves are mapped onto the process tree in orange.

The alignment move mapping shown in Figure 2.5 already allows us to repair a process tree using an approach that is similar to the one of Fahland et al. [13]. However, sometimes the cause of a problem within a process model is non-local, as is shown in Figure 2.6, which shows another incorrect process tree that needs to be repaired to match our running example. Here, the real problem is the incorrect \( \rightarrow \)-operator, marked in red, that needs to be changed into a \( \land \)-operator, but the alignment move mapping can only identify problems at the leaf node level.

That is why the alignment move mapping has been extended to include operator nodes. All of the moves mapped to the children of an operator node are also mapped to the operator node itself. This means for example that the synchronous, log and model moves mapped to the leaf nodes \( B \) and \( D \)
CHAPTER 2. ALIGNMENT-BASED PROCESS TREE REPAIR

and C in Figure 2.6 are also mapped to their parent →-operator node, but this also causes them to be mapped to the →-operator node at the root of the process tree. We currently do not have a way to identify which node in the process tree provides the best location to repair a problem, other than trying all possibilities and evaluating the quality of the resulting process trees.

Figure 2.6: Another incorrect process tree and a mapping of the alignment of the trace \( l = \langle ABCEG \rangle \) onto the tree. The problematic part of this tree is the →-operator marked in red, but this cannot be observed directly by looking at the alignment.

The extended alignment move mapping provides us with the information to identify the parts of a process tree that have a negative effect on the replay fitness of the model. It also indicates whether additional activities should be inserted into the process tree or whether the existing activities should be made skippable, which is used to determine how to repair the process tree.

2.6 Repairing Problem Areas within Process Trees

In this section we explain how to repair process trees using the information from the alignment move mapping, while also taking into account the precision quality dimension. In general, there are three ways to repair a part of a process tree: (1) behaviour can be removed from the tree, (2) behaviour can be added to the tree, and (3) the behaviour of the tree can be changed.

To illustrate how to repair a process tree, we will repair the process tree shown in Figure 2.5 until it matches the process of the running example from Section 2.3. Our example event log consists of the following two traces: \( l_1 = \langle ABCEG \rangle \) and \( l_2 = \langle ACDBFG \rangle \). The alignment move mapping for trace \( l_1 \) that is displayed in Figure 2.5 shows us that the process tree needs to be able to skip activities D and F, and that the activities C and E should be inserted into the process tree.

2.6.1 Removing Behaviour

Behaviour can be removed from a process tree in two ways: by making activities skippable or by removing activities. Choosing which of the two options is more appropriate depends on the entire event log. In our example, the alignment move mapping in Figure 2.5 showed us that we need to repair the process tree by removing the behaviour that activity D is always executed in every trace. We choose to make the leaf node D skippable, instead of removing it entirely, because our model only contains a single leaf node able to execute D and the event log contains a trace without D (i.e. trace \( l_1 \)), but also a trace with the execution of activity D (i.e. trace \( l_2 \)).

Making an activity skippable can be done by replacing its leaf node in the process tree with a choice between that activity and an unobservable activity. To allow activity D in Figure 2.5 to be skipped it can be replaced by a choice (×-operator) between D and a τ-node (i.e. an unobservable activity), the result of which is shown in Figures 2.5 and 2.6.

An activity can be removed from a process tree by simply removing its leaf node from the tree. However, if the parent node is a ∪-operator then the leaf node cannot be deleted because a loop

Figure 2.5: Another incorrect process tree and a mapping of the alignment of the trace \( l = \langle ABCEG \rangle \) onto the tree. The problematic part of this tree is the →-operator marked in red, but this cannot be observed directly by looking at the alignment.
operator needs to have three children, as mentioned in Section 2.1. In that case, the activity is not removed from the tree but it is instead replaced by a $\tau$-node.

### 2.6.2 Adding Behaviour

Behaviour can be added to a process tree by inserting additional leaf nodes into the tree. However, there are two issues that we have to deal with when adding activities to a process tree. The first issue is that the alignment move mapping gives us two possibilities for the location in the process tree where an activity can be inserted in order to repair the tree, as each log move is mapped onto two leaf nodes. We currently do not have a way to decide between the two locations when adding an activity, other than trying both possibilities and evaluating the quality of both resulting process trees. The second issue is that the activity we are trying to add may not always be executed in every trace at the point in the process tree where it is being added. In our example, activity $E$ is a log move in the alignment shown in Figure 2.5 and it has been mapped to leaf nodes $B$ and $F$. However, activity $E$ cannot simply be added as another child of the parent nodes of $B$ or $F$, because $E$ is only executed in one of the two traces in our log (i.e. trace $l_1$). This means that we need to determine the relationship between the activity we are trying to add and the location where it is added, which is one of the nodes it is mapped to. Once we have determined the type of this relation then we replace the node it is mapped to by an operator node that has both the added node and the mapped node as its children.

To explain how to add an activity in our example, we make the arbitrary decision to add activity $E$ to leaf node $F$ and not to leaf node $B$. This means that we replace $F$ with an operator node that has $E$ and $F$ as its children. We then need to determine the type of the operator node joining $E$ and $F$. In general, if we take the non-exclusive choice ($\lor$-operator) as the relation between two nodes then we are always able to add the activity to the process tree without reducing the replay fitness of the model. However, this hurts the precision quality dimension if the true relation between the two activities is more restrictive. In this case, it allows activities $E$ and $F$ to both be executed in the same process instance, while the traces $l_1$ and $l_2$ in our event log only contain either $E$ or $F$ and never both.

We use the following approach to determine the best type for an operator node to join the activity and the leaf node it is mapped to. First, we classify each trace in our event log by using the trace’s alignment to determine what the allowed operator types are that enable the process tree to correctly replay the executions of the activity and the leaf node in that alignment. We then check which operator types allow for the correct replay of the executions of the activity and the leaf node in at least a certain percentage $p$ of all trace alignments in which at least one of the two is executed. Finally, we choose the most restrictive operator amongst all such operator types, in order to prevent unnecessary losses in precision. Note that the $\neg$-operator is more restrictive than the $\land$-operator and the $\times$-operator is more restrictive than the $\lor$-operator, while the $\land$-operator is also more restrictive than the $\lor$-operator. If there are multiple executions of the activity or the leaf node in a single trace’s alignment then only the $\lor$-operator is considered to be a correct operator for that trace. An exception to the approach above occurs if the two activities that we want to join together are identical, in which case we have identified a self-loop and do not follow the approach to determine the best operator node to join these two activities. Instead, the leaf node in the process tree is simply replaced with a $\lor$-operator that can repeatedly execute the activity.

In our example, the best operator that can join activities $E$ and $F$ is chosen as follows. The trace $l_1$ only contains activity $E$, which means that this trace can only be replayed correctly if the operator joining $E$ and $F$ is a $\times$-operator or a $\lor$-operator. Likewise, the trace $l_2$ only contains the activity $F$, so this trace can also only be replayed if the operator is a $\times$-operator or a $\lor$-operator. The best operator to join $E$ and $F$ is the most restrictive operator that can successfully replay both traces, which is the $\times$-operator in this example.

As a second example, we also show how to add activity $C$ to the process tree in order to repair the process tree in Figure 2.5. We again make an arbitrary choice and add activity $C$ to node $B$ and not to node $F$, which means that we replace node $B$ by an operator node joining $B$ and $C$. The
operator types that allow this combination to correctly replay the occurrences of B and C in trace \( l_1 = (\text{ABCEG}) \) are the \( \to \)-operator, the \( \land \)-operator and the \( \lor \)-operator. In trace \( l_2 = (\text{ACDBFG}) \) the order of B and C is reversed, which means that we need the \( \leftarrow \)-operator (i.e. reverse the order in which the children are joined to the \( \to \)-operator node), the \( \land \)-operator or the \( \lor \)-operator to correctly replay the occurrences of B and C in this trace. This means that only the \( \land \)-operator and the \( \lor \)-operator correctly joins these two activities for both traces and the \( \land \)-operator is used because it is the more restrictive of the two. Note that we only look at the occurrences of the activities we are joining when determining the correct operator type. In trace \( l_2 \) the execution of activity D in between the execution of activities C and B does not affect the choice of our operator type.

To summarise, an activity can be added to a process tree by replacing a node onto which a log move is mapped by an operator node joining the added activity and the mapped node. The type of the operator node is determined by checking for each trace’s alignment what operator type is needed to correctly replay the executions of the added activity and the mapped node in that alignment, and then selecting the most restrictive operator that can correctly replay the executions of the activity and the node in at least \( p \) percent of all trace alignments in which at least one of the two is executed.

If we combine the changes to the process tree in Figure 2.5 that are described in this section and the previous (i.e. to make node D skippable, to add activity E to node F, and to add activity C to node B) then we can repair the process tree and we obtain the process tree shown in Figure 2.7. This tree allows for the same behaviour as the process tree shown in Figure 2.3.

![Figure 2.7: A new process tree that was created by repairing the tree from Figure 2.5. It has the same behaviour as the tree from Figure 2.3.](image)

### 2.6.3 Changing Behaviour

The behaviour of a process tree can be changed through a combination of adding and removing activities, but it can also be changed by changing the type of an operator node in the tree. In this section we show how to repair a process tree when an operator node needs to be changed to a different type. To explain our approach, we use the process tree from Figure 2.6 that has an incorrect \( \to \)-operator node as the parent of B and C. Our event log again consists of the traces \( l_1 = (\text{ABCEG}) \) and \( l_2 = (\text{ACDBFG}) \).

The method that is used to determine the best operator type when changing the type of an operator node is very similar to the method used in Section 2.6.2 to select the correct operator type to join two activities. However, here a child of an operator node can be a leaf node or a sub-tree (i.e. a node and all its descendants). Determining the best operator type for a node depends on the behaviour of the sub-trees that are the children of the operator node. In the following, we use the alignment move mapping as an indication of the behaviour of the sub-trees within a process tree. The behaviour of a node is equal to the model and synchronous moves mapped onto that node. This notion of behaviour is used during the selection of the correct operator type when determining which parts of a trace are explained by which nodes in the process tree.

There is no limit to the number of children that a non-\( \to \)-operator can have, so it may be the case that some children of the operator we are repairing are already joined together with the
correct operator type, while other children need to be split off and need to receive a new parent with a different operator type. For that reason we introduce a compositional approach to change an operator type.

The first step in this approach is the random selection of two children, which are joined together with a new operator type. The correct operator type is determined in a similar way as when adding an activity to a process tree. For each trace, an operator type is determined that allows the two children to correctly replay the behaviour that they explain in that trace’s alignment, and the most restrictive operator type that can correctly replay at least $p$ percent of all trace alignments is selected as the new operator type.

In our example, we arbitrarily select the nodes $B$ and $C$ as the two children of the $\rightarrow$-operator that we are changing first. Given the event log containing the traces $l_1$ and $l_2$, the correct operator to join these two nodes is the $\land$-operator, as shown before. The resulting sub-tree is shown as the second process tree in Figure 2.8.

The next step in the approach is to randomly select another child and to join this child to the sub-tree created in the previous step, by creating a new operator that joins the sub-tree and the selected child, which together form another sub-tree. The type of the new operator is again chosen based on the behaviour that the two children of the new operator explain. This step is repeated until all children of the original operator are added with a new operator to the new sub-tree.

In our example, the remaining child of the incorrect $\rightarrow$-operator is the sub-tree with a $\times$-operator joining $D$ and a $\tau$-node. This sub-tree is joined to the sub-tree with nodes $B$ and $C$ using a new operator. In this case, the behaviour in trace $l_2 = \langle ACDBF \rangle$ that is explained by the $\times$-child (i.e. activity $D$) is interleaving the behaviour explained by the new sub-tree with nodes $B$ and $C$. This means that the $\land$-operator is the most restrictive operator that can correctly replay both traces when both sub-trees are joined together. The resulting sub-tree is shown as the third process tree in Figure 2.8.

If there were additional children of the incorrect $\rightarrow$-operator then they would again be joined to the newly created sub-tree, but because we have added all children, the new sub-tree substitutes the sub-tree rooted at the old incorrect $\rightarrow$-operator. The resulting model is the rightmost process tree shown in Figure 2.8 which has the same behaviour as the process tree shown in Figure 2.3.

The approach described above can be used to change an operator node in order to repair a process tree by improving its replay fitness, while also taking into account the precision dimension. However, there are still several open problems with the approach. First, the sub-tree that is created to replace the incorrect operator is different depending on the order in which the children of the incorrect operator are joined together. It is not possible to determine the best order in which to join the children together without trying all possibilities and evaluating the resulting process trees. Second, there can be multiple locations in a process tree where an operator node can be changed to repair a problem. As was also the case when adding behaviour, there is currently no method to determine the location of a change that would repair a process tree in the best way possible, other than trying multiple possibilities and evaluating the quality of the resulting process.

---

Figure 2.8: The leftmost process tree is the process tree from Figure 2.6 with an incorrect $\rightarrow$-operator marked in red. The two middle process trees are the sub-trees that are created while determining the best way to change the incorrect operator. The rightmost tree is the repaired process tree with the second sub-tree replacing the incorrect operator node.

The approach described above can be used to change an operator node in order to repair a process tree by improving its replay fitness, while also taking into account the precision dimension. However, there are still several open problems with the approach. First, the sub-tree that is created to replace the incorrect operator is different depending on the order in which the children of the incorrect operator are joined together. It is not possible to determine the best order in which to join the children together without trying all possibilities and evaluating the resulting process trees. Second, there can be multiple locations in a process tree where an operator node can be changed to repair a problem. As was also the case when adding behaviour, there is currently no method to determine the location of a change that would repair a process tree in the best way possible, other than trying multiple possibilities and evaluating the quality of the resulting process.
trees. Third, we have not mentioned how to change ↪-operators because we currently do not have a good method to repair an incorrect ↪-operator. These open problems are the reason why we still need a genetic algorithm such as the Evolutionary Tree Miner to be able to use the approach efficiently. In the next chapter we explain how the approaches to repair a process tree that are described in this chapter can be applied to improve the effectiveness of the ETM algorithm.
Chapter 3

Extending the Evolutionary Tree Miner

In this chapter we answer the last two research questions and explain how the alignment-based process tree repair described in Chapter 2 have been used together with a new approach that creates an initial population of process trees to improve the effectiveness of the Evolutionary Tree Miner. We first introduce the ETM algorithm and briefly describe how it works in Section 3.1. In Section 3.2 we then show how the alignment move mapping introduced in Section 2.5 has been used to implement a new type of crossover operation. Following this, in Section 3.3 we explain how the different ways to repair a process tree that are described in Section 2.6 have been implemented as new mutation operations. Finally, in Section 3.4 we give a new approach that is used to create an initial population of process trees that already have a good quality.

3.1 The Evolutionary Tree Miner

The Evolutionary Tree Miner is a genetic process mining algorithm that was first introduced in [8] as a new algorithm that guides process discovery based on the four quality dimensions of replay fitness, precision, generalisation and simplicity. The ETM algorithm has been implemented as a plug-in for the well-known, extensible, process mining framework ProM 6 [19] and the extensions described in this thesis have been made available there.

The overall process that is followed by the ETM algorithm is shown in Figure 3.1. The input of the ETM algorithm is an event log containing a recording of the observed behaviour of a process. The first step of the algorithm is the creation of a randomly generated population of process trees. For each process tree, an overall quality score is computed that is based on a configurable weight of the four quality dimensions. If one of the stop criteria has been met, such as the discovery of a process tree with a certain desired fitness score or a maximum number of generations that has been reached, then the process tree with the highest quality score is returned. Otherwise, the models in the current population will have to be improved. The best models in the population, the elite, are preserved to keep them available to be improved in future generations. The rest of the future population is created by randomly selecting models from the current population and changing them, while giving better models a higher probability to be selected for change. Note that the same model may be selected multiple times and that the elite models can also be selected to be changed. The quality score is recomputed for the new generation of models and the stop criteria are evaluated again. This cycle is repeated until one of the stop criteria is satisfied.

The randomly selected process trees are changed through the application of crossover and mutation operations. First, each of the process trees has a certain, configurable, probability to be selected for crossover. The crossover operation recombines two process trees in order to create two new trees with characteristics from each of the parent models. A randomly chosen mutation operation is then applied on each of the randomly selected process trees, including those changed
through crossover. Mutation operations change an individual process tree, and there are five different mutation operations that are currently used in the ETM algorithm: (1) the replacement of a process tree, (2) adding a node, (3) the removal of a sub-tree, (4) changing a node, and (5) flattening the tree.

In the following sections, the general working of the ETM algorithm remains the same, but the algorithm has been extended through the creation of a new crossover operation, new mutation operations and a new way to generate random process trees.

### 3.2 Alignment-based Crossover

In this section we present an approach that is an alternative to the standard crossover operation used by the ETM algorithm. The crossover operation is used to combine existing process trees in the current population. In the original implementation of the ETM algorithm, crossover is applied on two process trees by randomly selecting a node in each tree and swapping the selected sub-trees between the two models, which produces two new models that may have behaviour that is very different from their parents. This means that the probability is very low that the application of the crossover operation leads to a model of a higher quality, which makes the ETM algorithm slow.

Therefore, we have created a new crossover operation that tries to combine two process trees in a smart way instead of in a random way. The basic idea behind our alignment-based crossover is that the following two observations need to be taken into account when combining two parts from different process trees in the best way possible: (1) the combination of the behaviour that is explained by both parts should come as close as possible to explaining the behaviour observed in the input event log, and (2) the overlap in that behaviour should be as small as possible. The first observation directly relates to the replay fitness quality dimension because that measures the degree to which the behaviour in the event log can be replayed by the model. The second observation is related to the simplicity and generalisation dimensions because explaining the same behaviour multiple times leads to more complex models that have more parts of the model that are less frequently used. This means that if we also have a way to take precision into account when combining two process trees then our approach touches all four quality dimensions.

These observations, together with the insights from Chapter 2, have been used to create a new crossover approach that determines the best way to combine two process trees and that has been implemented as follows. The synchronous moves from the alignment move mapping introduced in Section 2.5 are used as an indication of the behaviour of a node and its descendants in a process tree. So, for each node in one of the two trees that is being combined, we try each possible pairing with nodes from the other tree in order to find a pair such that the following three conditions hold: (1) the union of the synchronous moves from both nodes covers as much of the event log as possible, while (2) the intersection of the synchronous moves is as small as possible, and (3) the
difference in the number of synchronous moves mapped to the two nodes is minimised. The first two conditions are based on the two observations described above and the third condition was added to ensure that there is a balance between the explaining power of the two parts that we are combining. Each of these three conditions provides a score in terms of the number of events in the event log that are not covered by moves in the union, the number of moves that intersect, and the difference in the number of moves mapped to each of the two nodes, respectively. A weighted average of these three scores is used to determine an overall score for each possible pairing of nodes. The two sub-trees rooted at the nodes of the best scoring pair are then combined into a new process tree. They are joined using an operator node and the type of this node is determined using the approach described in Section 2.6.3 for the selection of the most precise operator to combine two sub-trees for which an alignment move mapping is known. Note that this approach always produces a single combined process tree, so we return the same tree twice to ensure that the population size stays the same.

The approach of the new alignment-based crossover takes all four quality dimensions into account. It has been tested and compared with the old crossover operation in several experiments that are presented in Chapter 4.

3.3 Alignment-based Mutation

In this section we present two new mutation operations that are based on the three ways to repair a part of a process tree as described in Section 2.6 (1) removing behaviour, (2) adding behaviour and (3) changing behaviour. In contrast to the new crossover operation, these new mutation operations are not replacing the original mutation operations that are mentioned in Section 3.1, but they are used in addition to those operations.

There are five different mutation operations that are used in the original ETM algorithm: (1) the replacement of a process tree, (2) adding a node, (3) the removal of a sub-tree, (4) changing a node, and (5) flattening the tree. The most extreme mutation operation is the replacement of a process tree, which replaces an entire process tree with a newly generated random tree. This ensures that the bad process trees in a population are occasionally replaced with fresh trees. For node addition, a random node that is not a $\land$-operator is selected in the tree. If the randomly selected node is a leaf node, or with a 50% probability if the randomly selected node is an operator node, a new operator node is inserted between the random node and their old parent. Otherwise, a new random leaf node is added as an extra child to the operator node. For the removal mutation, a random node is selected and removed from the tree, together with all the descendants of that node. The mutation to change a node is also applied on a randomly selected node, but leaf nodes are only changed to other leaf nodes, randomly chosen to represent a different activity, and operator nodes are only changed to a different, random, non-$\land$-operator. The flattening of a process tree is a mutation that merges operator nodes with their children if they are operator nodes of the same type. Flattening the tree in Figure 2.7 merges the two $\land$-operators, which changes the tree into the tree seen in Figure 2.3.

The first new mutation operation that we have implemented is a mutation operation that removes the sub-tree that contains the largest amount of incorrect behaviour when compared to its correct behaviour. For each node in the process tree a ratio is computed. This ratio is the number of executions of a node during the replay fitness calculation that resulted in a log or model move during the execution of that node or during the execution of one of its descendants, over the total number of executions of that node. The node with the worst ratio is then removed from the tree together with all of its descendants.

The second new mutation operation repairs a single node based on the approaches described in Section 2.6. The node that is repaired is randomly selected from the process tree. If that node is a leaf node onto which model moves are mapped, but no log moves, then the ability to skip this node is inserted, as described in Section 2.6.1. If the node is a leaf node onto which log moves are mapped then one of the log moves is randomly selected and that activity is added to the node, as described in Section 2.6.2. If the node that was selected for repair is an operator node then the
CHAPTER 3. EXTENDING THE EVOLUTIONARY TREE MINER

The type of the operator node is changed using the approach described in Section 2.6.3.

The way in which the mutation operators are randomly selected has been modified slightly with the addition of the new mutation operations. In the original ETM algorithm, one of the five mutation operations described above is selected randomly with a certain probability and then this mutation is applied to change a process tree. If this results in the creation of a tree that is already contained in the current population then a random mutation is again selected and applied on the original process tree. This is repeated until a tree is created that is not yet contained in the current population or until a maximum number of mutation attempts have been made. In the new approach, there is a certain probability that either the guided sub-tree removal or the node-repairing mutations are selected and a certain probability that one of the original five mutations is applied on the process tree. We again repeat the application of mutation operations until a new process tree is created or until a certain number of attempts have been made, but in the second case the algorithm stops trying to apply the new mutations and continues to try to create a new process tree using only the application of the old mutations for a number of additional attempts.

The addition of the new mutations has been tested and compared with the old ETM algorithm and with the use of the alignment-based crossover. The results of these experiments are presented in Chapter 4.

3.4 Initial Population Creation

In this section we address the final research question and we give an approach that is used to create an initial population of process trees that already have a reasonable quality. This improves the ETM algorithm when it is used for process discovery because fewer changes are needed from the initial population to reach a model with a high quality. The approach is also used instead of the random tree creation in the tree-replacement mutation to create new process trees.

One possible method to start with good process models is to use the output of a different process discovery algorithm in the creation of the initial population. However, most process mining algorithms only return a single solution [7], while the ETM algorithm works best with a diverse population because this decreases the probability that we start with a model that is very difficult to repair. Another problem with the use of the output of other process discovery algorithms is that most of these algorithms do not produce process trees, which means that we need to translate the output to a process tree in order to use it in those cases. This translation is currently a manual process, which is time-consuming for large or complex process models. Therefore, we propose a different method for the creation of the initial population, which we will explain in the following two sections.

3.4.1 Trace-model Creation

Instead of building an initial population of random models, we suggest creating the initial population by making models that can replay just a single randomly selected trace from the event log. Figure 3.2 shows what such a trace-model looks like for the trace \( l = \langle \text{ADCBDBFG} \rangle \). The only operator node is a sequence operator and the children of that operator are the activities, arranged in the order in which they occur in the trace.

There are several reasons why these trace-models are good candidates for the initial population of process trees. First of all, they score perfectly on precision because there is no possibility of choice in the model. Furthermore, they also score perfectly on simplicity as there are no useless
nodes in the model, and they score very good on generalisation because each node is executed frequently. The replay fitness of these models is usually quite low, but they are able to perfectly replay at least a single unique trace in the log and they are often able to replay more traces because a trace may occur multiple times in an event log, and traces that occur frequently have a higher probability to be selected for the creation of a trace-model. Also, recall that the techniques that repair a process tree, as described in Section 2.6 and implemented as the mutation operations described in Section 3.3, focus mainly on improving the replay fitness quality dimension, so it may be easier to improve a model with a low replay fitness score and a high score for the other quality dimensions than it is to repair a model with a low score for the quality dimensions other than replay fitness.

This alternative approach for creating random models was implemented and tested and the results can be found in Appendix A.1, which shows that it is an improvement over the old method of creating an initial population. However, if the trace contains repeated activities due to the presence of loops in the process we are discovering then the approach creates process trees that are difficult to repair due to the insertion of many duplicate activities. Furthermore, the creation of trace-models does not help with the discovery of processes that contain choice constructs that lead to complex sub-processes because a single trace can only show one of the possible results of a choice. Hence, we have extended the approach to deal with these issues and this advanced trace-model creation is described in the next section.

3.4.2 Advanced Trace-model Creation

As an extension of the creation of trace-models for the initial population, we propose a two-step approach to create advanced trace-models in order to make it easier to discover processes that contain loops and to enable the discovery of choice constructs. Instead of selecting a single trace to create a trace-model, we now select multiple traces for the creation of a single advanced trace-model. We take a random number of traces \( n \) from our event log, with \( n \) chosen according to the probability mass function \( P(n) = \frac{1}{2^{n+1} - m} \) for \( n \geq m \) (i.e., \( m \) is the minimal number of traces that we select). For each randomly selected trace, we build a trace-model as described above and we then detect and repair duplicate activities inside the trace-model. The trace-models are then merged by first selecting two trace-models and merging those and then merging the result with another trace-model and so on, until all trace-models have been merged and we have created an advanced trace-model.

The first step of the approach is the detection of potential loops in the process we are discovering by looking for repeated activities in the initial trace-models. If a trace is selected to be used to create a trace-model and that trace contains repeated activities then we insert a \( \mathcal{L} \)-operator at the first occurrence of the first activity that occurs multiple times. For example, the trace \( l = \{ \text{ADCBDBFG} \} \) contains repetitions of activity \( B \) and \( D \) and the original trace-model is shown in Figure 3.2. The trace-model is adapted by inserting a \( \mathcal{C} \)-operator at the position of the first occurrence of the first repeated activity, which is the \( D \) that occurs after \( A \). The repeated activities are then divided randomly over the ‘do’ and ‘redo’ children of the \( \mathcal{L} \)-operator, with a \( \rightarrow \)-operator combining activities if multiple activities end up at the same child. Two possible changed trace-models for the trace \( l = \{ \text{ADCBDBFG} \} \) after loop detection are shown in Figure 3.3.

The method of inserting \( \mathcal{C} \)-operators described above may result in a lower replay fitness of the trace-models. For example, the process trees in Figure 3.3 can no longer perfectly replay the trace \( l = \{ \text{ADCBDBFG} \} \) because \( C \) cannot be executed right after the first \( D \). However, the adapted trace-models have a better generalisation than the original trace-model in Figure 3.2. Furthermore, an application of the alignment-based mutation on the \( \rightarrow \)-operator at the root of the process trees may result in an evaluation of the relation between the \( \mathcal{C} \)-operator and activity \( C \), which may lead to a reparation of the tree through the insertion of a \( \land \)-operator to join the \( \mathcal{C} \)-operator and activity \( C \). Since the improvement of the replay fitness of a process tree is the main focus of the new mutation operators, we do not consider it to be an issue that the trace-model is no longer able to perfectly replay the trace that was initially used to generate the model.

The second step of the approach to create advanced trace-models involves the merging of
multiple trace-models, which works as follows. We first select two of the trace-models that we have created and we start by mapping all activities from each of the trees onto the other tree in order to see the differences between both trees. Note that such a mapping is easily created because our activity repetition detection has guaranteed that each activity occurs at most once in each trace-model. We then identify the differences between both models, where we do not care about the order in which the activities occur because this can be repaired later through the alignment-based mutation. Following the identification of the differences, a new process tree is created that contains the common behaviour from both trace-models and it offers a choice between the identified differences. The resulting tree is then merged with one of the remaining trace-models and this is repeated until all trace-models have been merged or until a process tree is created in which an activity occurs more than once.

We explain how to merge trace-models using the six trace-models from Figure 3.4 as an example and the approach is also shown in Figure 3.5. In our example, the first step is merging the process trees from Figures 3.5a and 3.5b. The only relevant difference between these two models is the activity that is missing from 3.5b and hence the resulting model that is shown in Figure 3.5c offers a choice between skipping it. The second step is the merging of the resulting model from Figure 3.5c and the trace model in Figure 3.5d. The difference between both models is now bigger and we introduce a choice between the sub-sequence starting at D up to and including F in Figure 3.5c, and the sequence of H and I from Figure 3.5d. The resulting model is shown in Figure 3.5e. Merging the process tree from Figure 3.5e with the trace-model from 3.5f does not result in a new model, as 3.5g is identical to 3.5e. This is because we consider -operators to be included in the model already when there is at least one descendant that can be mapped to an activity in the other tree. In this example, Figure 3.5e already contains the activity H and hence merging it with the model in Figure 3.5f does not add anything new. However, the model from 3.5h does contain new activities, as is shown in Figure 3.5i with the addition of a -operator. Finally, merging Figure 3.5i and 3.5j results in the process tree from Figure 3.5k. This is because activity N is the only activity in Figure 3.5j that cannot be mapped to the other tree and hence it is inserted as a possible choice in between D and F.

In some cases it is possible that the merging of trace-models has to be stopped before all trace-models have been merged, because the merger creates duplicate activities. An example where this happens is shown in Figure 3.6. The differences between 3.6a and 3.6b are the activities D and H, and F and I. However, creating two choices, one between D and H and one between F and I, results in a merged model that is no longer precise. That is because it can then also produce the traces l₁(ABCDEIG) and l₂(ABCFHGI), while there was a dependency between D and F and between H and I in the original trace-models. Therefore, we always capture the differences between two models in at most a single choice per merger, as is shown in the resulting merged model in Figure

---

**Figure 3.3:** Two adapted trace-models for the trace \( l = \langle ADCEBFG \rangle \) after the detection of repeated activities.
Figure 3.4: Six trace-models that have been created by selecting random traces from an event log and detecting repeating activities in each trace.

The approach to create advanced trace-models that is described above has been implemented and is used to create models for the initial population and it is used in the tree replacement mutation operation. Each time that a random tree is generated there is a certain probability that an advanced trace-model is created and a probability that a simple trace-model is returned. The approach has also been tested and the results are presented in Chapter 4.
Figure 3.5: The merging of the six trace-models from Figure 3.4 and the models that result from the mergings, with the differences between the trace-models in different colours.
Figure 3.6: Merging the left and middle process trees results in the creation of a duplicate E activity, as is shown in the resulting merged model on the right.
Chapter 4

Experimental Evaluation

In this chapter we present the results of the experiments that were performed in order to test the effectiveness of the new extensions to the ETM algorithm. In Section 4.1, we cover the tests that were performed on a set of artificial event logs belonging to relatively simple processes. On these event logs the effectiveness of the new implementation for crossover and the extended mutation operators were tested and compared, which is discussed in Section 4.1.1, as well as the new approach to creating a good initial population, which is discussed in Section 4.1.2. Following that, in Section 4.2, we show the results from the tests that were performed on event logs belonging to randomly generated process trees. These tests were used to examine the effectiveness of the ETM extensions on process (re)discovery under variations in process model size, discussed in Section 4.2.1, and variations in the amount of noise in an event log, discussed in Section 4.2.2.

4.1 Testing on Artificial Processes

The effectiveness of the extensions of the ETM algorithm described in sections 3.2, 3.3 and 3.4 has been tested using artificial event logs belonging to five different processes. These processes and the corresponding event logs are described in Appendix B. The performance of the ETM algorithm and its extensions were measured in the experiments by running the algorithm multiple times for a fixed number of generations and recording in each generation the quality of the best scoring model discovered up to that point. In the end, we are interested in obtaining the best possible process model within a certain time limit, and the time it takes to calculate the quality of a process tree is significantly longer than the time it takes to use that information to make good changes. This means that if we can use one of the extensions to obtain a model of equal or higher quality in fewer generations than before then we have made the ETM algorithm faster.

The general ETM parameter settings that were the same in each experiment are shown in the list below, but some parameter settings differed between experiments. The number of generations that the ETM algorithm runs and the number of runs per test are different for the different event logs. This is due to time constraints on the running time of the experiments, as it can take a very long time to calculate the alignments for a large event log or a model that allows for a lot of behaviour.

- **Overall Fitness**: A weighted average of $10 \times$ Replay Fitness, $5 \times$ Precision, $1 \times$ Simplicity and $1 \times$ Generalisation. Experience has shown that this weighted average works well in general for the ETM algorithm to obtain intuitively “good” models.
- **Population Size**: 20. This is the total number of process trees that are being stored by the ETM algorithm in each generation, so this includes the elite and the models being evolved.
- **Elite Count**: 6. These are the process trees that have the highest overall fitness score of all models in the current population. The elite are copied from each generation to the next, until a higher scoring model is discovered to replace them.
• **Alignment Calculation Time-out:** 1 second. This time-out specifies the maximum duration that the calculation to determine the optimal alignment for a single trace may take. If this time-out is triggered then the replay fitness is considered to be very bad, so the entire calculation is cancelled and the model receives an overall fitness of 0.

• **Crossover Probability:** 10%. This is the probability that a process tree will be selected for crossover.

• **Mutation Operator Probability:** 20%. There are five mutation operations in the original ETM algorithm and each operation has an equal (20%) probability to be used on a process tree when it is selected for mutation.

### 4.1.1 Crossover and Mutation Test

In this experiment the ETM extensions in the form of a new crossover operator, described in Section 3.2, and new mutation operations, described in Section 3.3, were evaluated and compared with the original ETM implementation. The experiment used three different setups that were each tested with multiple runs per event log: the original ETM algorithm, the ETM algorithm with the alignment-based crossover replacing the old crossover, and the ETM algorithm extended with the two alignment-based mutation operations while still using the old crossover operator. The following parameter settings are specific for this experiment:

• **Alignment-based Crossover Probability:** 50%. This is the probability that a process tree will be selected for crossover with the new alignment-based crossover.

• **Alignment-based Mutation Probability:** 75%. This is the probability that one of the two new mutation operations will be used instead of the original mutation operations. Therefore, each of the two new mutations has an effective probability of 37.5% to be applied, while each of the original mutations has a 5% probability to be applied to a process tree.

• **Operator Correctness Threshold:** 80%. This is the percentage \( p \) from sections 2.6.2 and 2.6.3 that defines the minimum percentage of all trace alignments that should be correctly replayed by an operator type before it can be chosen as a valid operator to join two activities or sub-trees.

The results below contain two types of graphs. The first type shows the overall fitness of the manually constructed original model together with the overall fitness of the best model per generation for the three different setups, averaged over all runs with a certain setup. The second type of graph shows a breakdown of the overall fitness into all four quality dimensions for a single setup, averaged over all runs with that setup.

### Running Example

An event log that was generated for the process described in Section 2.3 was used in this test and the results are presented below. Figure 4.1 shows the overall fitness of the best model per generation for the three different setups and the overall fitness of the process tree from Figure 2.3. Each setup was tested by running the ETM algorithm 100 times, so the solid lines show the average overall fitness over these runs and the dotted lines show the 95% confidence interval of the average. It is clear that both the use of the alignment-based crossover and the alignment-based mutation have a positive effect on the overall fitness in this test. The ETM algorithm is made faster by both extensions in the sense that the expected number of generations to reach a certain overall fitness level has decreased for all fitness levels. However, the alignment-based mutation setup performs better than the alignment-based crossover setup, and the alignment-based mutation is on average able to discover a model that has an overall fitness close to the fitness of the original model.

Figures 4.2, 4.3, and 4.4 show the breakdown of the overall fitness into the four quality dimensions for the original ETM setup, the alignment-based crossover setup and the alignment-based mutation setup, respectively. In terms of replay fitness, both the alignment-based crossover and the alignment-based mutation setups have a faster growth at the start, but only alignment-based
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.1: The overall fitness of the best model per generation for three different ETM setups on the Running Example event log. The solid lines show the average over 100 runs and the dotted lines show the 95% confidence intervals around the average.

Figure 4.2: A breakdown of the overall fitness score for the results of the original ETM setup on the Running Example event log.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.3: A breakdown of the overall fitness score for the results of the alignment-based crossover setup on the Running Example event log.

Figure 4.4: A breakdown of the overall fitness score for the results of the alignment-based mutation setup on the Running Example event log.
mutation leads to a higher replay fitness in the end. For precision, both extensions show a faster increase and a significantly higher value in later generations than the original ETM. Both extensions also lead to a slightly higher generalisation and simplicity, but what is more interesting is that the metrics for these dimensions appear to be measuring very similar things, as the growth patterns for both values are nearly identical.

Driver’s License Model

The Driver’s License event log described in appendix B.1 was used in this test and the results are presented below. Each setup was again tested by running the ETM algorithm 100 times. Figure 4.5 shows that the alignment-based crossover and the alignment-based mutation again have a rapidly increasing overall fitness in the beginning, but none of the approaches are able to discover models that are as good as the original model. Furthermore, after 150 generations there is no longer a statistically significant difference between the original ETM and the alignment-based crossover setup. The alignment-based mutation setup again has the best performance.

Figures 4.6, 4.7 and 4.8 again show the breakdown of the overall fitness into the four quality dimensions. The biggest difference between the original ETM and the alignment-based crossover setup is that the new crossover leads to a higher replay fitness at the cost of a lower precision, which is not surprising as the alignment-based process tree improvements mainly focus on improving replay fitness. The alignment-based mutations also lead to a higher replay fitness at the cost of a lower precision, but here the replay fitness is significantly higher than for the original ETM.

Insurance Claim Model

The Insurance Claim event log described in appendix B.2 was used in this test and the results are presented below. Due to the size of this event log and the resulting long alignment calculations, the original ETM and the alignment-based mutation setups were tested with 50 runs and the alignment-based crossover was tested with 25 runs. Figure 4.9 again shows that both the alignment-based crossover and the alignment-based mutations lead to models with a higher overall fitness than the original ETM algorithm, with the alignment-based mutations giving the best results. For this event log both ETM extensions eventually lead to the creation of models that fit better with the event log than the original model.

Figures 4.10, 4.11 and 4.12 show the breakdown of the overall fitness into the four quality dimensions. The biggest difference between the original ETM and the alignment-based crossover setup is again that the latter leads to a slightly higher replay fitness score. The alignment-based mutation setup has a significantly higher replay fitness compared to the original ETM, but this comes at the cost of a slight drop in precision. There is no big difference between the three setups in terms of generalisation and simplicity, but it is again clear that both metrics appear to be measuring very similar things.

Simple Loop-choice Model

The Simple Loop-choice event log described in appendix B.3 was used in this test and the results are presented below. The original ETM and the alignment-based crossover setups were tested with 30 runs and the alignment-based mutation setup was tested with 40 runs. Figure 4.13 shows that none of the setups are able to discover process trees that are as good as the model that was used to generate the event log. The alignment-based mutation still performs significantly better than the other two setups and after the first 130 generations there is no statistically significant difference between the original ETM algorithm and the ETM extended with the alignment-based crossover.

Figures 4.14, 4.15 and 4.16 show the breakdown of the overall fitness into the four quality dimensions. Apart from a slightly higher replay fitness at the start, there is no significant difference between the original ETM and the alignment-based crossover setup. The alignment-based mutation setup leads to models that have a much higher replay fitness than the models produced
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.5: The overall fitness of the results of the three different ETM setups on the Driver’s License event log.

Figure 4.6: A breakdown of the overall fitness score for the results of the original ETM setup on the Driver’s License event log.
Figure 4.7: A breakdown of the overall fitness score for the results of the alignment-based crossover setup on the Driver’s License event log.

Figure 4.8: A breakdown of the overall fitness score for the results of the alignment-based mutation setup on the Driver’s License event log.
Figure 4.9: The overall fitness of the results of the three different ETM setups on the Insurance Claim event log.

Figure 4.10: A breakdown of the overall fitness score for the results of the original ETM setup on the Insurance Claim event log.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.11: A breakdown of the overall fitness score for the results of the alignment-based crossover setup on the Insurance Claim event log.

Figure 4.12: A breakdown of the overall fitness score for the results of the alignment-based mutation setup on the Insurance Claim event log.
by the original ETM algorithm, but this comes at the cost of the other quality dimensions and in particular at the cost of precision.

**Double Loop-choice Model**

The Double Loop-choice event log described in appendix B.4 was used in this test and the results are presented below. All three setups were tested by running the ETM algorithm 30 times for each setup. Figure 4.17 shows a pattern that is very similar to the previous results. The alignment-based crossover setup is slightly better than the original ETM algorithm, but the alignment-based mutation is significantly better, although none of the approaches are able to discover models that are as good as the model that was used to generate the event log.

Figures 4.18, 4.19 and 4.20 show the breakdown of the overall fitness into the four quality dimensions. The alignment-based crossover setup has a slightly higher replay fitness that comes at the cost of a small amount of precision when compared to the original ETM setup. The alignment-based mutation also shows a similar result as before, with a much higher replay fitness than the original ETM algorithm, which comes at the cost of a large amount of precision and small amounts of generalisation and simplicity.

**General Observations**

The results from the tests on the five event logs described above show that the extension of the ETM algorithm with alignment-based mutations leads to the creation of process trees that have a higher overall fitness than the models created with the original ETM algorithm or the alignment-based crossover extension. The quality of the discovered models also increases more rapidly in the first generations for both extensions when compared to the original ETM algorithm, but the alignment-based crossover does not always lead to process trees that in the long run are of a statistically significantly higher quality than the original ETM. However, the models created by the alignment-based mutation setup are often improved in terms of replay fitness at the cost of precision. Furthermore, none of the approaches appear able to discover models for all five tests that are as good as the models originally used to generate the event logs. Finally, the results show that the metrics that are used to measure the generalisation and simplicity dimensions are heavily correlated.

The reason why the alignment-based mutations perform better than the alignment-based crossover is most likely because the mutations are able to repair a process tree by removing, adding or changing behaviour, while the alignment-based crossover can only add or change behaviour and this behaviour needs to be present in the population already in some way. Furthermore, the alignment-based crossover operation generally makes large changes to the process tree, while mutation can also change a single node in a small way. This also explains why there is not always a significant difference between the original ETM algorithm and the alignment-based crossover setup in the long run, as the alignment-based crossover is unable to repair the process tree once the general structure of the process has been discovered.

Note that in the experiments described above we have not considered using both the alignment-based mutations and the alignment-based crossover together. However, the combination of both extensions has not led to results that are statistically significantly different from the results that were obtained by using only the alignment-based mutations, as is shown in appendix A.2. Apparently the alignment-based crossover cannot repair a process tree in ways that cannot be done just as efficiently using the alignment-based mutations.

**4.1.2 Initial Population Creation Test**

In this experiment the new approach for the creation of an initial population described in Section 3.4 was evaluated. The experimental setup consisted of the ETM algorithm extended with both the alignment-based mutations and the new initial population creation approach. This experimental setup is compared to the original ETM algorithm and the ETM algorithm extended with only
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.13: The overall fitness of the results of the three different ETM setups on the Simple Loop-choice event log.

Figure 4.14: A breakdown of the overall fitness score for the results of the original ETM setup on the Simple Loop-choice event log.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.15: A breakdown of the overall fitness score for the results of the alignment-based crossover setup on the Simple Loop-choice event log.

Figure 4.16: A breakdown of the overall fitness score for the results of the alignment-based mutation setup on the Simple Loop-choice event log.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.17: The overall fitness of the results of the three different ETM setups on the Double Loop-choice event log.

Figure 4.18: A breakdown of the overall fitness score for the results of the original ETM setup on the Double Loop-choice event log.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.19: A breakdown of the overall fitness score for the results of the alignment-based crossover setup on the Double Loop-choice event log.

Figure 4.20: A breakdown of the overall fitness score for the results of the alignment-based mutation setup on the Double Loop-choice event log.
the alignment-based mutations, which were both tested and compared in Section 4.1.1. The new initial population creation setup was tested over 100 runs on each of the five event logs introduced before. The following parameter settings are specific for this experiment:

- **Alignment-based Mutation Probability**: 75%. This is the probability that one of the two new mutation operations will be used instead of the original mutation operations. Therefore, each of the two new mutations has an effective probability of 37.5% probability to be applied, while each of the original mutations has a 5% probability to be applied to a process tree.

- **Operator Correctness Threshold**: 80%. This is the percentage $p$ from sections 2.6.2 and 2.6.3 that defines the minimum percentage of all trace alignments that should be correctly replayed by an operator type before it can be chosen as a valid operator to join two activities or sub-trees.

- **Minimal Number of Traces**: 4. If an advanced trace-model is created then it will consist of at least 4 trace-models that have been merged.

- **Advanced Trace-model Creation Probability**: 75%. This is the probability that an advanced trace-model is created when a new process tree is generated. In the remaining 25% of the cases a simple trace-model will be created.

**Running Example**

The Running Example event log described in Section 2.3 was used in this test and the results are presented below. Figure 4.21 shows that the initial population creation setup starts by creating models that are on average already of a higher quality than the models created after 500 generations with the original ETM algorithm. These models are quickly improved until they are as good as the model that was originally used to generate the event log. Furthermore, after 300 generations the models created with the experimental setup are on average scoring a small but statistically significant amount higher in terms of overall fitness than the model that was originally used to generate the event log. Comparing Figure 4.2 with Figure 4.22 also shows that the models created with the experimental setup are superior to models from the original ETM algorithm in terms of all four quality dimensions.

**Driver’s License Model**

The Double Loop-choice event log described in appendix B.1 was used in this test and the results are presented below. Figure 4.23 shows that the initial population already contains the model that was used to originally generate the event log. Figure 4.24 shows that generalisation is the only metric that has a score less than 1, which it has by definition. However, note that the generalisation score does show very minor improvements over time, which actually leads to a model that has a slightly higher overall fitness score than the model originally used to generate the event log. The reason for this is that generalisation is improved when the process tree contains more nodes that are frequently executed, so transforming an N-ary tree into its equivalent binary tree improves the generalisation metric and it does not hurt the metrics for the other quality dimensions.

**Insurance Claim Model**

The Insurance Claim event log described in appendix B.2 was used in this test and the results are presented below. Figure 4.25 shows that the experimental setup again starts with models that are just as good in terms of overall fitness as the models created after 200 generations with the original ETM algorithm. Furthermore, within 20 generations on average these models have also been improved in such a way that they are better than the models created after 200 generations when using only alignment-based mutation. Comparing Figure 4.10 with Figure 4.26 shows that the models created with the experimental setup are again better in all four quality dimensions.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.21: The overall fitness of the results of three different ETM setups on the Running Example event log.

Figure 4.22: A breakdown of the overall fitness score for the results of the initial population creation setup on the Running Example event log.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.23: The overall fitness of the results of three different ETM setups on the Driver’s License event log.

Figure 4.24: A breakdown of the overall fitness score for the results of the initial population creation setup on the Driver’s License event log.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.25: The overall fitness of the results of three different ETM setups on the Insurance Claim event log.

Figure 4.26: A breakdown of the overall fitness score for the results of the initial population creation setup on the Insurance Claim event log.
Simple Loop-choice Model

The Simple Loop-choice event log described in appendix B.3 was used in this test and the results are presented below. Figure 4.27 shows that the experimental setup creates models for the initial population that are on average better than the models created after 400 generations with either the original ETM algorithm or only the alignment-based mutation extension. Unfortunately, the experimental approach is still not able to create models that are on average of equal quality as the model that was used to generate the event log, but it comes close. Comparing Figure 4.14 with Figure 4.28 shows that replay fitness has been significantly improved at the cost of only a small amount of precision. There is no significant difference in terms of generalisation and simplicity between the experimental setup and the original ETM algorithm.

Double Loop-choice Model

The Double Loop-choice event log described in appendix B.4 was used in this test and the results are presented below. Figure 4.29 shows very similar results as the previous experiment. The initial population already contains high quality models that are even further improved, but not yet enough to be on average equal to the quality of the model used to generate the original event log. Comparing Figure 4.18 with Figure 4.30 shows that precision has been improved significantly at the cost of a little replay fitness, while generalisation and simplicity have also been improved a little in the experimental setup.

General Observations

The results from the tests on the five event logs described above show that the initial population creation extension in combination with the alignment-based mutation operations lead to the creation of process trees that are superior in quality when compared to the models created with the original ETM algorithm. The quality of the models created to fill up the initial population is often already higher than the quality of the models that are created after several hundred generations with the original ETM algorithm. Furthermore, the experimental setup is now able to regularly discover models that are of equal or higher quality as the models used to generate the event logs.

Another important point is that all four quality dimensions are usually well balanced in the results of the experimental setup. This is because three out of the four quality dimensions have optimal values in the models that are created for the initial population, while the focus of the alignment-based mutation is the improvement of the fourth quality dimension, which is replay fitness. This also means that if we change the weighted average of the four quality dimension metrics that defines the overall process model quality then our approach is still very likely to produce high quality process models according to the new overall quality definition.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.27: The overall fitness of the results of three different ETM setups on the Simple Loop-choice event log.

Figure 4.28: A breakdown of the overall fitness score for the results of the initial population creation setup on the Simple Loop-choice event log.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.29: The overall fitness of the results of three different ETM setups on the Double Loop-choice event log.

Figure 4.30: A breakdown of the overall fitness score for the results of the initial population creation setup on the Double Loop-choice event log.
4.2 Testing on Random Processes

The tests in Section 4.1 have shown that extending the ETM algorithm with a combination of the alignment-based mutations from Section 3.3 and the new approach for the creation of the initial population from Section 3.4 leads to the creation of process trees with a significantly better quality than the process trees created with the original ETM algorithm. However, the ETM extensions have only been tested on a limited number of event logs, so it is unclear what the effects of variations in process model size and the level of noise in the event log are on the quality of the results. Therefore, we have tested the extended ETM algorithm on event logs generated from random process trees and we discuss the results of these tests in sections 4.2.1 and 4.2.2.

The general ETM parameter settings are the same as they were for the tests from Section 4.1. The approach being evaluated is the extension of the ETM with a combination of the alignment-based mutations and the new approach for the creation of the initial population, so this is the same setup as in Section 4.1.2. The process trees used in these tests are generated randomly with a certain minimum number of nodes in the tree. Using these process trees, event logs are then generated with a certain noise level that indicates the probability per event that it may have been replaced by a random other event.

4.2.1 Testing on Models of Varying Size

This section contains the results of the tests on event logs from randomly generated trees of varying sizes. After generating a tree with a random number of nodes, an event log was created for each tree with 250 random traces that can be replayed on the random tree. The event logs contain 10% noise, so there is a 10% probability that an event from a trace was replaced with a random other event. In total, this experiment was performed on 20 random event logs created using an equal number of random trees, and for each event log 25 runs were performed with the original ETM implementation and 25 runs were performed with the extended ETM algorithm. The general ETM parameter settings for this experiment were the same as for the experiments in Section 4.1.

Figure 4.31 shows the aggregated results for five different event logs generated from trees with an average size of 19.0 nodes. The extended ETM setup clearly performs better than the original ETM algorithm, as it is able to discover process trees with an overall fitness equal to the overall fitness of the trees that were used to generate the event logs in much fewer generations. Furthermore, the extended ETM setup is able to discover process models with a higher fitness than the original trees, which means that the process rediscovery performed during these tests can also be seen as an extreme form of process repair where the entire process model is replaced with a new model. Figures 4.32, 4.33, and 4.34 show similar results for the aggregations of five process trees with an average size of 36.2 nodes, 48.8 nodes, and 69.0 nodes, respectively. For this last group of process trees we were not always able to calculate the overall fitness of the process trees used to generate the event logs, because of time-outs being triggered during the calculation of the replay fitness for these large models. However, the overall fitness levels for the process trees mined on these event logs show the same patterns as for the other event logs.

Figures 4.35, 4.36, and 4.37 show the overall fitness split up into the different quality dimensions. These figures show that the quality of the models created by the extended ETM setup are often superior to the original process trees in all four quality dimensions. However, these figures also show that for increasing sizes of the process models used to generate the event logs, the quality scores of the mined process trees increasingly resemble the quality scores that were seen for the different dimensions during the tests of the alignment-based mutation in Section 4.1.1 instead of the quality scores seen during the tests in Section 4.1.2. This means that the initial population is of a lower quality and that precision is being sacrificed in order to improve the replay fitness of these models. A possible reason why the effect of the initial population creation is reduced for larger process trees is that the initial population creation has a small minimal number of traces that it selects. The effect of this small minimal number of traces is that the merged trace-models that are created then have a large probability to only cover a small amount of the total behaviour allowed by these large models, which leads to an increase in the number of repairs that need to
be performed. However, it is possible to increase the minimal number of traces that are selected during the creation of the models for the initial population, which may increase the quality of the initial population again.

These results show that the extended ETM algorithm is still able to find high quality process trees for increasingly large processes. The discovered process models are often even better in terms of their overall fitness than the process trees that were used to generate the event logs. The size of a process does have an effect on the quality of the results, especially in terms of the precision of the mined models. However, it may be possible to decrease this effect by adjusting the parameter that determines how many traces are merged during the creation of the initial population. Finally, it becomes impractical to mine process trees for models that contain more than 70 nodes, because the calculation of the alignments takes an increasingly long amount of time.

4.2.2 Testing on Event Logs with Varying Noise Levels

This section contains the results of the tests on event logs with varying levels of noise that were created using randomly generated process trees. Each event log contains 250 traces and the noise level indicates the probability with which an event in a trace may have been replaced with a random other event. This experiment was performed on 5 event logs generated from random trees with an average size of 20.3 nodes, and for each event log 10 runs were performed per noise level with the original ETM algorithm and 10 runs per noise level were performed with the extended ETM implementation. The results in the graphs shown are aggregated per noise level.

Figures 4.38a and 4.38b show the effect of an increasing amount of noise in the event logs. It is clear that an increasing amount of noise in the event logs results in a lower overall fitness of the mined models. However, the extended ETM setup is affected less by noise than the original ETM algorithm. The graphs in appendix A.3 show that there are very little statistically significant differences between the quality of the models created after 200 generations with the extended ETM setup for the logs with 10%, 20%, 30% and 40% noise. These graphs also show that the extended ETM setup performs better than the original ETM algorithm on all noise levels. Furthermore, Figures 4.39, 4.40 and 4.41 show that the biggest effect of an increased noise level is the increasing amount of precision that needs to be sacrificed in order to repair the replay fitness of the process trees.

These results show that the ETM algorithm is quite robust to noise and that the proposed ETM extensions make it even more robust to noise than before. Also, the amount of noise in an event log does not influence the difference in quality between the process trees discovered by the extended ETM setup when compared with the models discovered by the original ETM algorithm.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.31: The overall fitness of the aggregated results on the five event logs with an average size of 19 nodes for the generating models.

Figure 4.32: The overall fitness of the aggregated results on the five event logs with an average size of 36.2 nodes for the generating models.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.33: The overall fitness of the aggregated results on the five event logs with an average size of 48.8 nodes for the generating models.

Figure 4.34: The overall fitness of the aggregated results on the five event logs with an average size of 69 nodes for the generating models.
Figure 4.35: The left graph shows a breakdown of the overall fitness score for the results of the experimental setup on the five event logs with an average size of 19 nodes for the generating models. The middle graph shows the average quality of the models used to generate these event logs. The right graph shows the quality of the results of the original ETM algorithm on the same event logs.

Figure 4.36: The left graph shows a breakdown of the overall fitness score for the results of the experimental setup on the five event logs with an average size of 36.2 nodes for the generating models. The middle graph shows the average quality of the models used to generate these event logs. The right graph shows the quality of the results of the original ETM algorithm on the same event logs.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.37: The left graph shows a breakdown of the overall fitness score for the results of the experimental setup on the five event logs with an average size of 48.8 nodes for the generating models. The middle graph shows the average quality of the models used to generate these event logs. The right graph shows the quality of the results of the original ETM algorithm on the same event logs.

(a) The effect of different noise levels on the original ETM algorithm.

(b) The effect of different noise levels on the extended ETM algorithm.

Figure 4.38: An overview of the effect of different noise levels in event logs on the overall fitness of the models discovered by the original and the extended ETM algorithm.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.39: A breakdown of the overall fitness score for the results of the extended ETM setup on the event logs with a noise level of 0%.

Figure 4.40: A breakdown of the overall fitness score for the results of the extended ETM setup on the event logs with a noise level of 30%.

Figure 4.41: A breakdown of the overall fitness score for the results of the extended ETM setup on the event logs with a noise level of 50%.
Chapter 5

Future Work

In this chapter we discuss the limitations of the work we have presented in this thesis and possible
directions for future research that may provide additional improvements of the Evolutionary Tree
Miner. In Section 5.1 we explain how the work from Section 2.5 could be applied on process model
notations other than process trees and how the work can be extended to cover quality dimensions
other than replay fitness. In Section 5.2 we discuss the areas where there is room to improve the
approach to repair a process tree, which was covered in Section 2.6. There are also improvements
that can be made to the implementation of the Evolutionary Tree Miner itself, which we discuss
in Section 5.3 together with some additional ideas for possible crossover and mutation operators,
and alternatives for the creation of process trees for the initial population. Finally, in Section 5.4
we discuss the additional experimental evaluation that we think may provide additional insights.

5.1 Identifying Problem Areas within Process Trees

Our first research question focusses on the identification of parts in a process model that have a
low quality, which we answered by giving an approach in Section 2.5 that maps an alignment onto
a process tree. We chose to focus on the process tree notation because that is the process model
representation that is used by the ETM algorithm internally and we focussed on the replay fitness
quality dimension because that is generally considered to be the most important quality dimension
[9, 13]. However, our approach could also be applied on process models constructed in a different
process modelling language and for quality dimensions other than replay fitness. Furthermore, the
current alignment move mapping does not provide information on the quality of areas of a process
tree that are not executed, which also leaves room for improvement.

In order to use the approach we have described to identify problem areas within a process
model with an arbitrary modelling notation, we need to translate that process model into a
process tree. Automatically translating a process tree to a different modelling language is possible
because each operator can be translated to a specific modelling construct (shown in Figure 2.1)
and these constructs can be composed hierarchically. On the other hand, it is not always possible
to translate an arbitrary (sound) process model to a process tree because process trees are well-
structured models and an arbitrary process model may be inherently unstructured [17]. Even for
the class of models that can be translated to a process tree, there is currently no tool that can
perform this translation automatically. However, recent work by Polyvyanyy et. al. [18] could
be used to create an implementation that transforms such process models into block-structured
process models that can then be automatically translated to process trees, in a similar way as
the translation from a process tree to a different modelling language. Of course, it is debatable
whether the problem areas identified in a process tree after such a transformation are also problem
areas in the original model, and whether repairing those problem areas and returning an improved
process tree can be considered an improvement of the original process model.

It is possible to create a mapping capable of identifying problem areas for quality dimensions
other than replay fitness that is analogous to the alignment move mapping presented in Section 2.5 depending on the implementation of those metrics. For the simplicity metric that we use, which measures the percentage of useless nodes in the process tree, it is trivial to create a mapping of useless nodes onto nodes in the process tree that tells us the number of useless nodes in each sub-tree. For precision, we could create a mapping that tells us the precision of each sub-tree in the process tree, by looking at the number of outgoing edges and used edges in the state space of the execution of the nodes of each sub-tree and calculating a precision score. Finally, we could create a mapping for generalisation that gives us the generalisation of each sub-tree, by looking at the number of executions of each sub-tree and the nodes inside.

A problem which we have not addressed before is that the alignment move mapping does not provide information on the sub-trees of a process tree that are not executed. This causes problems if there is an incorrect $\times$-operator, because only one of its children will be executed. This means that because all the moves are mapped to nodes in the single sub-tree that is executed, all the missing activities, which may be present in the other children of the $\times$-operator, can only be inserted in that single sub-tree when this alignment move mapping is used to repair the process tree. The operator itself cannot be changed with the approach described in Section 2.6 either, because we have no information on the behaviour of the sub-trees that are not executed, so we rely on the ETM algorithm to change the operator through random mutation. Perhaps we should also store information on the nodes in a process tree that are not executed, while they could have been, in order to identify the problem areas that we currently miss.

5.2 Repairing Problem Areas within Process Trees

In Section 2.6 we answered the second research question of this thesis, by explaining how the replay fitness of a process tree can be improved using the information obtained from the alignments between the process tree and an event log. There are still several open issues for the approach presented there, which we avoided because we applied the approach in the context of the ETM algorithm. However, solving these issues may make the ETM algorithm even more effective.

One of the open issues that we have mentioned is that there are often multiple possible locations where missing activities can be inserted into a process tree and multiple possible locations where an operator may be changed in order to improve the quality of a process model, while we have no clear method to determine which of the possible choices leads to the best process model. The solution that we currently use is that we rely on random choices and the context of the ETM algorithm in order to have a good probability to, eventually, make the right choices. If we can come up with a way to estimate the difference in process model quality after a change to the process tree, without having to entirely recalculate the replay fitness, then we may be able to determine the possible changes that have a good probability to improve the process model’s quality and speed up the ETM algorithm by making the right changes more often.

Another open issue is the repair of $\lor$-operator nodes and the detection of repetition in a process. Currently, we are only able to detect and insert self-loops, but if a process contains more complex repetition then we rely on the ETM algorithm and the original random mutations to insert a $\lor$-operator, whose children can then be repaired using the alignment-based mutations. Perhaps we can detect loops in a process using a similar approach as was used in Section 3.4 by looking at repeated activities in all traces in the event log and aggregating the results.

The approach presented in Section 2.6 is only used to improve the replay fitness and possibly the precision of a process tree, but if we create additional mappings that also identify problematic areas in a process tree for the other quality dimensions then we can also come up with new ways to improve those other dimensions in a process tree. For simplicity, we could simply remove the nodes that are identified as useless nodes from the process tree. Precision is more difficult to improve, but sub-trees that have a large number of escaping edges should perhaps be pruned or removed from the process tree. Generalisation can be improved by removing the nodes that are very infrequently used, but changing the process tree into a binary tree, wherever possible for non-$\lor$-operators, will also improve the generalisation metric that is currently being used.
5.3 Evolutionary Tree Miner Improvements

In Chapter 3 we have shown how the ETM algorithm can be improved, by applying alignment-based process repair techniques and through the creation of an initial population with high quality process trees. However, we have not discussed the memory use of our extensions or possible improvements of the metrics that are used to calculate the quality of a process tree. The ETM algorithm may also be improved even further by implementing additional crossover and mutation operations or by modifying the way in which the initial population is created.

The ETM algorithm is a process mining algorithm that is not just time-consuming, but it is also memory intensive and that is one aspect of our improvements to the ETM algorithm that we have not discussed before. The most computational-intensive task in the ETM algorithm is the calculation of alignments, but the alignments also take up a lot of memory, so in the original ETM algorithm the alignments themselves are thrown away after the quality scores of a process tree have been calculated. However, in our extensions of the ETM algorithm we rely on the information from the alignments to identify and repair weak areas in a process tree, which means that we now have to store the results of the alignment calculation. We have an alignment for every unique trace in our event log, and every process tree in the population has its own set of alignments, so for big event logs this can take up a lot of memory. We currently have not put effort into efficient storage of the alignments because we did not run into issues with the event logs we tested on, but this is something that should be investigated in order to be able to apply the ETM algorithm for event logs containing a large number of unique traces.

We rely on the metrics discussed in Section 2.2 to calculate a score for each of the different quality dimensions, but we have seen in the results presented in Chapter 4 that the metrics for simplicity and generalisation appear to be measuring the same things. So, we have come up with an alternative metric for generalisation, which has been implemented but not yet tested, that measures the standard deviation of the replay fitness per trace as an indication of generalisation. This metric says that we consider a process model that can correctly explain 80% of the behaviour of all traces in an event log to be more general than a process model that can correctly explain all of the behaviour of 80% of the traces in an event log and none of the behaviour of 20% of the traces. The reason why we consider the first situation to be more general is that a new, unobserved trace from the process we are modelling will likely be replayed correctly for a large part of the trace, while in the second situation this trace will likely be replayed either fully correctly or not at all. This new metric does not fully capture the concept of generalisation, but neither did the previous metric.

5.3.1 Crossover

We have created a new crossover operation that is described in Section 3.2, but we have remarked in Chapter 4 and shown in appendix A.2 that it does not provide any additional benefit to the ETM algorithm when the new alignment-based mutation operations are used. In general, the usefulness of crossover as an operation in Genetic Programming algorithms, the class of algorithms to which the genetic process mining of process trees belongs, has been the subject of much debate [15, 20]. This means that it may be possible that any crossover operation can also be implemented as an equivalent mutation operation that equally improves the effectiveness of the ETM algorithm.

That being said, we do have a new idea for a crossover operation that has already been implemented, but not yet tested. It works in a similar fashion as the crossover operation in the original ETM algorithm, which selected a random sub-tree in one process tree and swapped it with a randomly selected sub-tree from the other tree. This new crossover selects a sub-tree in one process tree and searches for a sub-tree from the other tree to swap with that explains as much as possible from the behaviour (mapped synchronous moves) and the missing behaviour (mapped log moves) of the first sub-tree, in the hope that the second tree contains a better explanation of the behaviour covered by the first sub-tree. Ideally, it would select a sub-tree to be replaced in the first tree that is of a low quality, but we have not yet come up with a way to quantify the quality of individual sub-trees, so the initial selection is still random.
CHAPTER 5. FUTURE WORK

5.3.2 Mutation

We have created a set of new mutation operations that are described in Section 3.3 that have improved the effectiveness of the ETM algorithm, which are used in addition to the original mutation operations. However, the alignment-based mutation operations still have difficulties discovering complex sub-processes being repeated inside a loop. Furthermore, once a large part of the process model’s structure has been discovered then it becomes difficult to make changes to a process tree that require the application of multiple small changes, because those small changes cause large losses of the model’s quality, so those intermediate models are quickly removed from the population.

As an alternative to a possible solution that allows us to use the alignment move mapping to repair \(\otimes\)-operator nodes, we present the following idea to rediscover the sub-process inside a loop. By using the new approach for the creation of an initial population, we are able to obtain models that contain \(\otimes\)-operator nodes with repeating activities, but the structure of those repeating activities is often incorrect. Therefore, we propose a new mutation operation that can be applied on a \(\otimes\)-operator node, which creates a new partial event log containing only the behaviour mapped to the \(\otimes\)-operator node, and which subsequently uses the Inductive Miner described in [14] on that partial event log to discover a process tree that explains the repeating sub-process. The reasons why we would use the Inductive Miner is that it directly produces a process tree and that it runs in polynomial-time, so it is probably fast enough on a partial event log to be used as a mutation operation. Note that we can also easily use this mutation on other types of operator nodes to (re)discover a sub-tree that explains the behaviour (synchronous moves) and missing behaviour (log moves) mapped to that node.

We can also create a set of mutations that rearrange the structure of a process tree, which may make it easier for other mutations to change the process tree without these changes breaking the overall structure of the process tree. In general, there are often many different process trees that are trace equivalent in terms of the behaviour that they can produce. Therefore, we can create mutation operations that distribute operator nodes differently over their parents. For example, a sequence with a binary choice inside can also be modelled as a choice between two sequences. Note that the transformed models may not be equivalent to the original models for all four quality dimensions.

5.3.3 Initial Population Creation

We have explained in Section 3.4 how we can create a population of process trees that already have a good quality, as was shown in the experiments in Chapter 4. The results showed that this approach was very effective, but there may be additional ways to improve it. Another alternative that we have also mentioned before is the creation of an initial population using the output from (multiple) different process mining algorithms.

Although the creation of advanced trace-models was very effective for the generation of a good initial population, we did observe that the quality of the initial population was lower when discovering larger processes than when discovering smaller processes. This is most likely related to the number of traces that are selected for the creation of the advanced trace-models. The minimal number of traces that was selected during the experiments was quite low and the probability mass function that determines the actual number of traces that are selected has an exponentially decreasing probability to generate numbers that are larger than the minimal number of traces. The result of the creation of a model that is built by merging only a small number of traces from the event log is that these models only cover a small amount of the possible behaviour for large processes. A solution may be to use a different probability mass function that has more variation in the number of traces that are selected to be merged into a single model for the initial population.

When we discussed the creation of the initial population we already indicated that it is possible to seed the initial population using the output from other process mining algorithms. The main reason why this has not yet been tested is that most process mining algorithms do not produce process trees and hence we need to apply a time-consuming manual translation to create the input...
for our ETM algorithm. However, it may be possible to implement an automatic translation of process models to process trees, as we have mentioned in Section 5.1, which would make it easier to generate the initial populations to test this approach.

5.4 Additional Experimental Evaluation

In Chapter 4 we have provided the results of several experiments on a combination of artificial event logs and randomly generated event logs, which show that the extensions presented in this thesis improve the effectiveness of the ETM algorithm, but additional experimental evaluation could provide additional insights.

Due to time constraints it was not possible to test the extensions of the ETM algorithm on real-life event logs or in a practical case study. Such additional evaluations would provide additional validation of the improvements made to the ETM algorithm, but they could also provide an opportunity to compare the practical usability of the ETM algorithm to that of other process mining algorithms. An important point to note here is that we have only evaluated the quality of the results in this thesis by looking at (a weighted average of) the scores of the four quality metrics described in section 2.2. However, these metrics are flawed and the resulting process models may not correspond to what a domain expert would consider a high quality model. Therefore, to properly evaluate the practical usability of the ETM algorithm, it would also be necessary to perform case studies on real-life event logs where the resulting process models are evaluated by domain experts.

Furthermore, we only evaluated the speed of the ETM algorithm and its extensions by comparing the quality of the best model in each generation, but we did not discuss the real time that the test runs took. Additional experimental evaluation is needed to investigate this aspect in depth, but to give an idea: the average time (and 95% confidence interval) for a single run on one of the five random event logs without noise took 48.1 ± 8.3 minutes with the original ETM and 30.9 ± 5.6 minutes with the extended ETM. This means that the extended ETM is around 50% faster than the original ETM. However, on the five random logs with 10% noise a single run took on average 81.7 ± 8.9 minutes with the original ETM and 105.8 ± 14.1 with the extended ETM, so in these cases the original ETM was faster. The reason for these differences is that the time it takes to calculate an alignment increases up until a replay fitness score of around 0.8 to 0.9, after which it decreases again. The models created with the extended ETM have a higher quality and reach that quality faster than the models created by the original ETM, which means that the extended algorithm creates more models for which it is expensive to calculate an alignment on the logs with 10% noise than the original ETM algorithm does. Therefore, the total time of an entire run is increased. On the logs without noise, the models that are created by the extended ETM algorithm have a higher replay fitness, so calculating an alignment for these models becomes less expensive. This illustrates that it is difficult to compare the real time that each run takes, which is why we only compared the speed of the ETM algorithm by looking at the quality in each generation.

In Section 1.2 we discussed two approaches that can be used to repair a given process model in order to improve its quality. It would be interesting to include the edit distance as a quality dimension and then compare the extended ETM algorithm with those two approaches in an experiment. We have mainly focussed on process discovery during the evaluation of our extensions, although we did show that the extended ETM algorithm is able to improve the quality of random process models in a drastic form of process repair by rediscovering an entirely new process tree.

In the description of our experiments in Chapter 4 we listed several parameter settings, without explaining why we chose those specific values for the parameters. Most of the values such as the population size and the elite count were chosen based on previous experience of what worked well for the ETM algorithm in most cases, but they do not have a significant impact on the results. However, it may be useful to do a proper sensitivity analysis to determine what the effects are of varying those parameters.

Finally, the ETM algorithm can also be used to discover a Pareto front of mutually non-
dominating process models, which allows the user to investigate the different trade-offs between different quality dimensions. It would be interesting to investigate how the extended ETM algorithm compares to the original ETM algorithm when discovering a Pareto front.
Chapter 6

Conclusions

In this thesis we have presented several approaches and extensions that improve the Evolutionary Tree Miner, a genetic process mining algorithm, and enable it to make smart changes to models, in order to obtain models of a higher quality in less time than the original implementation, while taking into account the four process model quality dimensions of replay fitness, simplicity, precision and generalisation. These approaches and extensions are based on concepts and ideas from process model repair, which have been applied in the context of the genetic ETM algorithm. We have shown, through experiments on both artificial and randomly generated event logs, that our approach is superior to the original ETM algorithm in its ability to quickly produce high quality models. Furthermore, we have presented additional areas of future research that may improve the effectiveness of the ETM algorithm even further.

In order to improve the effectiveness of the ETM algorithm and the quality of its results, we have answered the following research questions:

Process Model Repair

- How can we identify the parts of a process model that have a low quality, so that we can improve those parts instead of trying to improve parts that are already good?
- How can we improve the parts of a process model that have a low quality, so that we can create models of a higher quality?

Evolutionary Tree Miner

- How can we apply the concepts and ideas from process model repair, so that we can improve the effectiveness of the Evolutionary Tree Miner?
- How can we create an initial population of process models that already have a reasonable quality, so that we need few changes to reach a model with a high quality?

To answer our first research question, we have created an approach that can be used to identify areas of a process tree that have a low quality. This approach relies on the fact that the replay fitness is the most important quality dimension and therefore it focuses on identifying areas of a process tree that have a low replay fitness. Our approach uses information from the calculation of alignments between traces in the event log and model elements in order to create a so-called alignment move mapping that identifies potential locations for the insertion or removal of activities in the process tree. This enables us to improve those locations, instead of trying to improve the areas of a process model that already have a good quality.

We have answered the second research question of this thesis by presenting several different ways, based on concepts and ideas from process model repair, in which the quality of a process tree can be improved by using the information from the alignment move mapping. However, the current methods of alignment-based process model repair are not complete. Therefore, we make
use of the ETM algorithm and its randomness to explore the search space of possible repairs in order to find the best way to repair a process model.

The third research question has been answered when we discussed how the effectiveness of the ETM algorithm can be improved by applying the alignment-based process model repair approaches. These approaches have been applied in the ETM algorithm as crossover and mutation operations, which aim to improve the quality of a process tree.

We have answered the fourth research question by creating an approach that makes it possible to create an initial population of high quality process trees. This initial population creation approach randomly selects traces from an event log, which are then merged into a single process tree.

Extensive experimental evaluations have been performed in order to test the effectiveness of the extensions to the ETM algorithm when compared with the original implementation. These experimental evaluations have shown that the new alignment-based mutation operations and the new approach to create an initial population can create process trees that have a higher quality in significantly fewer generations than the original ETM algorithm. In addition to that, we have also tested the effects of varying process sizes and noise levels on the quality of the process trees that are discovered, and we have shown that the extensions to the ETM algorithm make it more robust to noise and able to discover high quality process models for both large and small processes.

Finally, we have discussed additional areas of interest for future work that aims at improving the ETM algorithm even further. This has shown that there are a lot of opportunities for additional extensions and improvements to the Evolutionary Tree Miner.
Bibliography


62 Alignment-based Process Model Repair and its Application to the Evolutionary Tree Miner
Appendix A

Additional Evaluation Results

In this appendix we present the results from several additional experiments that have been performed. In Section A.1 we show the evaluation of the creation of simple trace-models as an approach to creation models for the initial population. In Section A.2 we show the results of the experiment where alignment-based crossover and alignment-based mutation were combined. Finally, in Section A.3 we show additional graphs for the experimental evaluation of the effect of noise in the event log on the performance of the original ETM algorithm and the extended ETM algorithm.

A.1 Trace-model creation

Figures A.1, A.2, A.3, A.4 and A.5 show that the creation of advanced trace-models described in Section 3.4.2 performs better than the creation of simple trace-models described in Section 3.4.2.

![Graph showing overall fitness of results over generations](image)

Figure A.1: The overall fitness of the results of the creation of simple trace-models for the initial population on the Running Example event log.
APPENDIX A. ADDITIONAL EVALUATION RESULTS

Figure A.2: The overall fitness of the results of the creation of simple trace-models for the initial population on the Driver’s License event log.

Figure A.3: The overall fitness of the results of the creation of simple trace-models for the initial population on the Insurance Claim event log.
Figure A.4: The overall fitness of the results of the creation of simple trace-models for the initial population on the Simple Loop-choice event log.

Figure A.5: The overall fitness of the results of the creation of simple trace-models for the initial population on the Double Loop-choice event log.
A.2 Combining Mutation and Crossover

Figures A.6, A.7, A.8, A.9, and A.10 show that the combination of the alignment-based crossover operation described in Section 3.2 and the alignment-based mutations described in Section 3.3 performs no better than the alignment-based mutations combined with the old crossover operation.

![Graph showing overall fitness across generations](image)

Figure A.6: The overall fitness of the results of the combination of the alignment-based crossover and the alignment-based mutations on the Running Example event log.
APPENDIX A. ADDITIONAL EVALUATION RESULTS

Figure A.7: The overall fitness of the results of the combination of the alignment-based crossover and the alignment-based mutations on the Driver’s License event log.

Figure A.8: The overall fitness of the results of the combination of the alignment-based crossover and the alignment-based mutations on the Insurance Claim event log.
APPENDIX A. ADDITIONAL EVALUATION RESULTS

Figure A.9: The overall fitness of the results of the combination of the alignment-based crossover and the alignment-based mutations on the Simple Loop-choice event log.

Figure A.10: The overall fitness of the results of the combination of the alignment-based crossover and the alignment-based mutations on the Double Loop-choice event log.
A.3 Varying Noise Levels

Figure A.11 is a modified version of Figure 4.38a with the addition of confidence intervals, which shows that there are very little statistical significant differences between the quality of the models created for different noise levels in event logs. After 200 generations there is only a statistical significant difference between the models created for the event logs with 10% and 40% noise.

Figures A.12, A.13, A.14, A.15, A.16 and A.17 show that there is a statistical significant difference between the performance of the original ETM algorithm and the extended ETM algorithm, independent of the amount of noise in the event logs.

Figure A.11: An overview of the effect of different noise levels in event logs on the overall fitness of the models discovered by the extended ETM algorithm.
Figure A.12: A comparison of the overall fitness of the results of the original ETM algorithm and the extended ETM algorithm, for event logs with 0% noise.

Figure A.13: A comparison of the overall fitness of the results of the original ETM algorithm and the extended ETM algorithm, for event logs with 10% noise.
Figure A.14: A comparison of the overall fitness of the results of the original ETM algorithm and the extended ETM algorithm, for event logs with 20% noise.

Figure A.15: A comparison of the overall fitness of the results of the original ETM algorithm and the extended ETM algorithm, for event logs with 30% noise.
Figure A.16: A comparison of the overall fitness of the results of the original ETM algorithm and the extended ETM algorithm, for event logs with 40% noise.

Figure A.17: A comparison of the overall fitness of the results of the original ETM algorithm and the extended ETM algorithm, for event logs with 50% noise.
Appendix B

Artificial Models and Event Logs

In this appendix we introduce four of the artificial models and event logs that were used in the experiments described in Section 4.1.

B.1 Driver’s License Model

Figure B.1 shows the Driver’s License Model. This model was used to generate an event log containing 40 traces, of which 4 are unique, without noise.

B.2 Insurance Claim Model

Figure B.2 shows the Insurance Claim Model. This model was used to generate an event log containing 5000 traces, of which 936 are unique, with a significant amount of noise in the form of missing activities, repeated activities and activities replaced with incorrect activities.

B.3 Simple Loop-choice Model

Figure B.3 shows the Simple Loop-choice Model that was generated using the Process Log Generator from [11]. This model was used to generate an event log containing containing 500 traces, of which 106 are unique, without noise.

B.4 Double Loop-choice Model

Figure B.4 shows the Double Loop-choice Model that was generated using the Process Log Generator from [11]. This model was used to generate an event log containing containing 500 traces, of which 59 are unique, without noise.
APPENDIX B. ARTIFICIAL MODELS AND EVENT LOGS

Figure B.1: The Driver’s License Model.

Figure B.2: The Insurance Claim Model.
APPENDIX B. ARTIFICIAL MODELS AND EVENT LOGS

Figure B.3: The Simple Loop-choice Model.

Figure B.4: The Double Loop-choice Model.