Static and Dynamic Visualization of Quality and Performance Dimensions on Process Trees

R. Wolffensperger

Supervisors:
prof.dr.ir. W.M.P. van der Aalst
ir. S.J.J. Leemans
dr. O. Türetken

Final version

Eindhoven, October 2014
Abstract

Nowadays, more and more companies record information about their processes in event logs to better understand them, and, subsequently, improve their efficiency. Process mining aims to discover, monitor, and improve real processes by extracting knowledge from event logs in a structural way. This thesis discusses how companies could get detailed insights into their process by measuring quality (i.e., conformance) and performance not only on the entire process model or single activities, but also on parts of the model, i.e., on different levels of granularity. For this purpose process trees are used, in which the process is represented by a tree. The resulting hierarchy can be exploited to perform quality and performance analyses on different levels of granularity.

Currently, no techniques are present to measure quality and performance and visualize these measurements on process trees. Therefore, we define quality and performance metrics for process trees, and visualize the results in both static and dynamic ways. The static visualizations project the results on elements of the process tree, the dynamic visualizations show an animation of the steps executed in the process on the process tree.

In order to extract information from an event log we need to construct an alignment between a process model discovered from an event log and the event log itself. Then, we can measure the quality of the model and the performance of the process. Subsequently, we can visualize these measurements by projecting them on elements of the process model.

All work presented in this thesis has been implemented in the Process Mining (ProM) framework.

**Keywords:** process mining, event log, process tree, measuring quality and performance, visualize quality and performance results
Preface

This master’s thesis describes the main results of my graduation project for the Business Information Systems (BIS) master at Eindhoven University of Technology (TU/e). It was conducted within the Architecture of Information Systems (AIS) research group of the Mathematics and Computer Science department.

First of all, I would like to thank Wil van der Aalst for his guidance during this project. I am very thankful that he provided me with this interesting opportunity. I would also like to thank Sander Leemans for the valuable advices during our regular meetings.

I want to thank my office mates Bart Hompes and Jeroen van Mourik. Our discussions during the project helped me a great deal. Furthermore, my thanks go out to Joos Buijs for his \LaTeX template [1], which gave me a head start in writing this thesis.

Last but not least, I want to thank my parents and sister for their support over the years. They are always there for me.

Robin Wolffensperger
Eindhoven, October 2014
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiii</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Context ........................................... 1
1.2 Problem Statement .................................. 2
1.3 Approach .......................................... 2
1.4 Outline ........................................... 4

## 2 Preliminaries

2.1 Event Log ............................................ 5
2.2 Process Tree ........................................ 6
2.3 Alignment .......................................... 9
2.4 Quality Dimensions ................................. 10
2.5 Multisets .......................................... 11

## 3 Measuring Quality and Performance on Process Trees

3.1 Aligning Process Tree and Event Log ................. 13

### 3.1.1 Extend Process Tree and Convert to Petri Net 13
### 3.1.2 Construct Alignment ........................ 14

3.2 Measuring Quality of Process Trees ............... 14

### 3.2.1 Fitness ................................... 14
### 3.2.2 Precision ................................ 15
### 3.2.3 Generalization ............................. 17
### 3.2.4 Simplicity ................................ 17

3.3 Measuring Performance in Process Trees ........... 18

### 3.3.1 Time Metrics ................................ 18

3.4 Additional metrics ................................. 22

### 3.4.1 Frequency .................................. 22
### 3.4.2 Positioning Log Moves ...................... 22
### 3.4.3 Useless Nodes .............................. 24

3.5 Discussion ......................................... 24

## 4 Static and Dynamic Visualizations on Process Trees

4.1 Static Visualizations Framework .................... 27

### 4.1.1 Normalization of Measurements ................ 27
### 4.1.2 Projecting Measurements on a Process Tree 28
### 4.1.3 Resulting Framework ......................... 30

4.2 Dynamic Visualizations Framework .................. 31

### 4.2.1 Determine Token Path ........................ 32
## CONTENTS

4.2.2 Construct Timed Path .................................................. 33
4.2.3 Animation ................................................................. 34
4.2.4 Resulting Framework ................................................... 36
4.2.5 Discussion ................................................................. 36

5 Evaluation ................................................................. 37
5.1 Synthetic Logs ............................................................ 37
  5.1.1 Quality_analysis_log .................................................. 37
  5.1.2 Telecom_service_log .................................................. 43
  5.1.3 Discussion ............................................................. 51
5.2 Real-life Log .............................................................. 51
  5.2.1 Loan Application Subprocess ....................................... 51
  5.2.2 Loan Offer Subprocess .............................................. 54
  5.2.3 Work Items Subprocess .............................................. 56
  5.2.4 Discussion ............................................................. 59

6 Conclusions and Future Work ........................................... 61
  6.1 Future Work .............................................................. 62

Bibliography ............................................................... 63

Appendix ................................................................. 65

A ProM Plugin Manual ...................................................... 65
  A.1 Replay a Log on a Process Tree for Quality and Performance Analysis Plugin ... 65
    A.1.1 General information ................................................ 65
    A.1.2 Manual .............................................................. 66
# List of Figures

1.1 An example of a process tree. .................................................. 2  
1.2 The intended goal of the project. ............................................. 3  
1.3 A schematic overview of the approach. ..................................... 3  
2.1 An example of a process tree. .................................................. 7  
2.2 Another example of a process tree. .......................................... 7  
2.3 A process tree represented as a DAG. ....................................... 8  
2.4 The unfolded process tree. ..................................................... 8  
2.5 The resulting Petri net after converting the process tree in Figure 2.1. The boxes indicate the process tree operator nodes. ......................... 9  
2.6 The four quality dimensions. .................................................. 10  
3.1 Process tree in which leaves are labeled with activity names. .......... 14  
3.2 Transformation of a leaf into a sequence of start and complete. ...... 14  
3.3 Example of an alignment automaton. ....................................... 16  
3.4 Process tree discovered from the event log in Table 3.1. ............... 16  
3.5 Process tree used to illustrate the calculation of the time metrics. .... 18  
3.6 Conversion of the or operator into a Petri net. ........................... 22  
3.7 Process tree in which the position of the log move is highlighted. ..... 24  
3.8 Taxonomy of the metrics defined in this chapter. ........................ 25  
4.1 Example legends, all using the same color scheme. ....................... 28  
4.2 Example of a projection of “Average execution time” on the node color, “Useless node” on the border color, and “Frequency” on the edge color. Corresponding legends are displayed below the tree. ......................... 28  
4.3 Examples of projections on node, border, and edge size, and on edge and leaf label. ............................................................ 29  
4.4 The static visualizations framework. ....................................... 30  
4.5 Process tree discovered from the repairexample event log. ............ 31  
4.6 Token path through a process tree. ....................................... 31  
4.7 Token animation on a process tree. ....................................... 35  
4.8 The dynamic visualizations framework. .................................... 36  
5.1 Legend used for the quality visualizations (see section 4.1.2). ........ 37  
5.2 A good model for the exampleLog. It can replay all of the events observed in the log, it balances overfitting and underfitting, and is simple. .......... 38  
5.3 Visualizations of the quality analysis on the good model. It is clear that this process tree and all of its subtrees score good on all four quality criteria. ......................................................... 39  
5.4 Model for the most frequent trace in the exampleLog, i.e., other traces will not fit this model. ................................................. 40  
5.5 Visualizations of the quality analysis on the most frequent trace model. Figure 5.5a shows this process tree’s fitness is low, meaning it cannot replay all traces observed in Quality_Analysis_log. ................................. 40
5.6 Flower model for the exampleLog. .................................................. 41
5.7 Visualizations of the quality analysis on the flower model. Figure 5.7b clearly shows
precision is low for this process tree. .................................................. 41
5.8 Part of the trace-model for the exampleLog. ....................................... 41
5.9 Visualizations of the fitness and precision results on the trace model. This process
tree scores well on both fitness and precision. .................................... 42
5.10 Visualizations of generalization on the trace model. As can be seen, generalization
of this process tree is low, since .......................................................... 42
5.11 Visualizations of simplicity on the trace model. Simplicity of this process tree is
really low, since there are a lot of leaves having the same label, i.e., duplicate nodes. 43
5.12 Process tree for the telecom service provider process .................................. 44
5.13 Visualizations of the quality analysis on the telecom service process tree. It scores
well on all quality criteria. ................................................................. 45
5.14 Average execution time. “Activate Number Transfer” is the activity that takes the
longest. Furthermore, the loop seems to be a bottleneck for this process. .......... 46
5.15 Average waiting time. All waiting times are relatively low compared to the execution
times (see Figure 5.14). “Activate Number Transfer” has the highest waiting
time. ........................................................................................................ 47
5.16 Average sojourn time. Again, the loop seems to be a bottleneck for this process. 48
5.17 Average synchronization time for the two parallel operators. ..................... 49
5.18 Token animation on the telecom service process. Most of the tokens are held up in
the loop part of the process (indicated in blue). The tokens in red are waiting for
this part to be completed. ..................................................................... 50
5.19 Process tree for the loan application subprocess. ...................................... 52
5.20 Visualizations of the quality analysis on the loan application subprocess. This
process tree scores well on all quality criteria. ........................................ 52
5.21 Average sojourn time projected on node color and edge label. The parallel subpro-
cess as well as the activity “A CANCELLED” seem to be the bottlenecks of this
process. .................................................................................................. 52
5.22 Average synchronization time projected on node color and edge label. Since syn-
chronization time is 0, the three parallel activities are all completed at the same
time. ........................................................................................................ 53
5.23 Token animation on the loan application subprocess. Tokens are either on their
way or held up in the bottlenecks of the process. ..................................... 53
5.24 Frequency projected on node color and edge label. We can conclude that loan
applications are often declined. .............................................................. 53
5.25 Process tree for the loan offer subprocess. ............................................. 54
5.26 Visualizations of the quality analysis on the loan offer subprocess. This process tree
scores well on all four quality criteria, although it could be simpler. .............. 55
5.27 Average sojourn time projected on node color and edge label. “O CANCELLED”
takes the longest, the loop and the operator node labeled xor next to it seem to
be the bottlenecks for this process. ......................................................... 55
5.28 Token animation on the loan offer subprocess. Most tokens are held up in the loop
construct, which means this seems to be a bottleneck of this process. .......... 55
5.29 Frequency projected on node color and edge label. It is clear that the “body” of
the loop is executed the most in this subprocess. .................................... 56
5.30 Process trees for the work items subprocess. .......................................... 57
5.31 Visualizations of the quality analysis on the work items subprocess. Fitness is very
low (0.15), which means the other metrics become less reliable. .................. 57
5.32 Visualizations of the performance analysis on the work items subprocess. ....... 58
A.1 ProM’s workspace. .............................................................................. 66
A.2 The event log and the process tree are imported in ProM. ......................... 67
A.3 The “Replay a Log on a Process Tree for Quality and Performance Analysis” plugin is selected. .......................................................... 67
A.4 The “Replay a Log on a Process Tree for Quality and Performance Analysis” plugin is running. .................................................. 68
A.5 Choose a visualizer from the dropdown menu. .......................... 68
A.6 Visualization when “Visualize Process Tree Animation” is selected. .... 69
A.7 Visualization when “Visualize Process Tree Statistics” is selected. .... 69
A.8 An example of a visualization in the Static Visualization Framework. Fitness is projected on node label, average sojourn time on node size and color, useless node is projected on border width and color, and frequency is projected on edge label, size, and color. ................................................................. 70
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Graphical representation of the process tree operators.</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>A simple event log.</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>Trace with timestamps and lifecycle transitions.</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>LaCoSeL for each node in exampleTree.</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>Starting and closing leafs for the operators in exampleTree.</td>
<td>20</td>
</tr>
<tr>
<td>3.5</td>
<td>nodeStamps for the nodes in exampleTree and timestamps in exampleTrace.</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Trace in the <code>repairexample</code> process.</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Timed path for the <code>repairexample</code> process.</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>A simple event log.</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Execution, waiting, and sojourn times for the activities in the telecom service process.</td>
<td>44</td>
</tr>
<tr>
<td>5.3</td>
<td>Bottlenecks in the loan application subprocess.</td>
<td>51</td>
</tr>
<tr>
<td>5.4</td>
<td>Bottlenecks in the loan offer subprocess.</td>
<td>54</td>
</tr>
<tr>
<td>5.5</td>
<td>Bottlenecks in the work items subprocess.</td>
<td>57</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This master’s thesis describes the main results of my graduation project for the Business Information Systems (BIS) master at Eindhoven University of Technology (TU/e). It was conducted within the Architecture of Information Systems (AIS) research group of the Mathematics and Computer Science department.

1.1 Context

Nowadays, more and more data, containing information about processes, is recorded by information systems. For example, the steps taken in booking a trip on internet are stored in databases of the organisations involved, or the steps involved in the repair of a MRI scanner in a hospital are recorded by the supplier’s information system. These steps are called events and a sequence of steps, i.e., an execution of the process, is called a trace. An event log is a collection of process executions.

Companies record information about their processes to gain insight into them in order to improve their efficiency. Companies want to check for instance if all tasks in the process are executed as expected, and to discover bottlenecks. Naturally, this cannot be achieved by just looking at an event log. We need tools to extract the information stored in event logs and analyze this data in a structural way.

This is where process mining comes into play. Process mining is an emerging discipline providing comprehensive sets of tools to provide fact-based insights and to support process improvements by extracting knowledge from event logs recorded by an information system. The AIS group investigates methods, techniques and tools for the design and analysis of data stored in event logs. This investigation is not only focussed on information systems and their architecture, but also on modeling and analyzing the business processes and organizations they support.

The Process Mining Framework (ProM) toolset is an extensible framework that supports a wide variety of process mining techniques in the form of plugins. These plugins can be used for a wide range of activities, such as discovering process models from event logs, analyzing the quality of a discovered model with respect to the recorded process, or analyzing the performance of the process. Quality analysis is used to find commonalities and discrepancies between modeled and observed behavior. This is also known as conformance checking. Performance analysis is used to discover bottlenecks in the process and see where the process could be improved in terms of efficiency. Results of these analyses can be visualized on the model by projecting them on model elements or by animating tokens on the model representing a case being handled.
CHAPTER 1. INTRODUCTION

1.2 Problem Statement

Companies and research groups, such as the AIS group, want to get insight into their processes in as much detail as possible. However, existing methods to compute quality and performance can only say something about the whole model or at activity level. In order to get more insight into specific parts of the process, one should be able to say something about parts of the model as well. Then, companies or research groups can improve the quality of the model, or the performance of the process at the cause of the problem.

Process trees are very suitable for this purpose. A process tree is a process model in which the process is represented by a tree (see Figure 1.1), and can be discovered from an event log using the Inductive Miner or the Evolutionary Tree Miner plugins in ProM. Furthermore, process trees respect the soundness property (the model contains no deadlocks and other anomalies), because of their hierarchical structure; every subtree models a specific part of each trace in the event log. As a result, we can say something about the quality (i.e., conformance) and performance of parts of the model as well, since we can consider each subtree as a separate process and perform quality and performance analysis on each subtree separately. However, quality and performance metrics which are applicable to (parts of) process trees, and visualizations that project these metrics, and animate tokens on process trees do not exist yet.

![Figure 1.1: An example of a process tree.](image)

1.3 Approach

Therefore, we need to develop quality and performance metrics on process trees (first goal), and, subsequently, come up with visualizations that project these metrics on elements of the process tree (second goal). These visualizations aim at giving direct information about the process regarding conformance of (parts of) the model with respect to the log and performance of the activities and (sub-)processes. Figure 1.2 depicts an example of what we want to achieve in this project.

We visualize quality and performance statistics in both static and dynamic ways. Static visualizations are used to get insight into the quality and performance of (parts of) the model. We assign a color to, adjust the size of, and/or add labels to nodes and edges in the tree so that for instance conformance issues and bottlenecks can be seen directly. Dynamic visualizations are used to replay the event log on the process tree; tokens flow through the tree so that we can see for instance whether all steps in the process are executed as expected.

In order to reach the first goal, an alignment between event log and process tree is constructed to be able to compute the quality of the model and to project additional information stored in the
log on the process tree. At the same time, research on existing quality and performance metrics is conducted. Based on this research metrics are developed and implemented such that they can be applied on process trees using the constructed alignment. The result is provided as a plugin in the ProcessTreeReplayer package in ProM.

For the second goal, two frameworks for the visualizations are designed and implemented; one for the static visualizations and one for the dynamic visualizations. The static visualizations framework shows quality and performance statistics on the model elements of the process tree. These are made configurable for the user; the user can choose which metric is to be visualized on which model element according to a specific color scheme. The dynamic visualizations framework shows the token animation in which the user can see tokens flowing through the model representing cases being handled at a specific time. Both frameworks are implemented as visualizer plugins in the ProcessTreeReplayer package in ProM. A schematic overview of this approach is depicted in Figure 1.3.

Finally, the visualizations are evaluated to show they are useful and can be applied in real-life.
1.4 Outline

This document is structured as follows: Chapter 2 introduces concepts such as event log, process tree, and alignment. The metrics used for computing quality and performance statistics are described in Chapter 3. In Chapter 4 the design and implementation of the visualization framework is covered, followed by the evaluation of the implemented visualizations in Chapter 5. Finally, conclusions and future work are presented in Chapter 6.
Chapter 2

Preliminaries

In this chapter basic concepts related to process mining and process trees are introduced. Furthermore, some basic concepts used in the remainder of this thesis are introduced as well.

2.1 Event Log

An event log records the events that occur in a certain process. A process consists of cases, a case consists of events, such that each event relates to precisely one case, events within a case are ordered, events can have attributes, and events can be related to some activity in the process [2]. More formally:

Definition 1 (Event, attribute).
Let $E$ be the event universe, i.e., the set of all possible event identifiers. Let $AN$ be a set of attribute names. For any event $e \in E$ and name $n \in AN$: $\#_n(e)$ is the value of attribute $n$ for event $e$. If event $e$ does not have an attribute named $n$, then $\#_n(e) = \perp$ (null value).

Examples of attributes are the activity associated to the event, the resource executing or initiating the event, or the timestamp of the event. Sometimes also the trans attribute, which specifies the lifecycle transition they represent in a transactional model of their generating activity [3], is recorded. The most common used lifecycle transitions are the “start” and “complete” transitions of a single activity.

We can use classifiers to classify events. A classifier is a function that maps the attributes of an event onto a label used in the process model discovered from the event log or in constructing the alignment between process model and event log. This classifier can be seen as the “name” of the event:

Definition 2 (Classifier).
For any event $e \in E$, $\perp$ is the name of the event.

For example, events can be classified by their associated activity ($e = \#_{\text{activity}}(e)$), or by their associated activity in combination with their lifecycle transition ($e = (\#_{\text{activity}}(e), \#_{\text{trans}}(e))$).

A case, trace and event log are defined as follows:

Definition 3 (Case, trace, event log).
Let $C$ be the case universe, i.e., the set of all possible case identifiers. Cases, like events, have attributes. For any case $c \in C$ and name $n \in AN$: $\#_n(c)$ is the value of attribute $n$ for case $c$ ($\#_n(c) = \perp$ if case $c$ has no attribute name $n$). Each case has a mandatory attribute trace: $\#_{\text{trace}}(c)$ (shorthand $\hat{c}$) $\in E^*$.

A trace is a finite sequence of events $\sigma \in E^*$ such that each event appears only once, i.e., for $1 \leq i < j \leq |\sigma|: \sigma(i) \neq \sigma(j)$.

An event log is set of cases $L \subseteq C$ such that each event appears at most once in the entire
log, i.e., for any \( c_1, c_2 \in L \) such that \( c_1 \neq c_2 : \delta_{\text{set}}(c_1) \cap \delta_{\text{set}}(c_2) = \emptyset \), where \( \delta_{\text{set}}(\sigma) \) converts the sequence \( \sigma \) into a set, e.g., \( \delta_{\text{set}}(\langle d, a, a, d \rangle) = \{a, d\} \).

If an event log contains timestamps, then the ordering in a trace should respect these timestamps, i.e., for any \( c \in L \), \( i \) and \( j \) such that \( 1 \leq i < j \leq |\hat{c}| : \#_{\text{time}}(\hat{c}(i)) \leq \#_{\text{time}}(\hat{c}(j)) \).

Events and traces are represented using unique identifiers (\( e \in E \) and \( c \in C \), respectively). This allows us to point to a specific event or case. This is important, because there may be many events having identical attributes. Similarly, there may be different cases that followed the same path in the process.

### 2.2 Process Tree

The Inductive Miner and Evolutionary Tree Miner plugins in ProM can be used to mine a process tree from an event log. A process tree is a process model and a compact abstract representation of a block-structured workflow net: a rooted tree in which leaf nodes are labeled with activities, and all other nodes are labeled with operators (from now on called operator nodes) [4]. All nodes have a unique identifier to distinguish leaves having the same label. Furthermore, process trees are sound by definition [5], therefore, correctness of the process model is guaranteed.

There are a total of eight standard operators related to control flow defined for process trees: 

- **seq** denotes the children are to be executed sequentially, 
- **AND** denotes the children are be executed in parallel, 
- **XOR** denotes an exclusive choice between the children based on data, 
- **loopXOR** denotes a loop where the decision to continue or stop the loop is based on data, 
- **OR** denotes a multi-choice between the children (i.e., one or more children are to be executed), 
- **event** denotes an event construct (this can either be a **timeout** or a **message**), 
- **DEF** denotes a choice between the children based on events, 
- **loopDEF** denotes a loop where the decision to continue or stop the loop is based on events.

Table 2.1 shows the graphical representation of aforementioned operators.

<table>
<thead>
<tr>
<th>operator</th>
<th>graphical representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td><img src="image" alt="seq" /></td>
</tr>
<tr>
<td>AND</td>
<td><img src="image" alt="AND" /></td>
</tr>
<tr>
<td>XOR</td>
<td><img src="image" alt="XOR" /></td>
</tr>
<tr>
<td>loopXOR</td>
<td><img src="image" alt="loopXOR" /></td>
</tr>
<tr>
<td>OR</td>
<td><img src="image" alt="OR" /></td>
</tr>
<tr>
<td>timeout</td>
<td><img src="image" alt="timeout" /></td>
</tr>
<tr>
<td>message</td>
<td><img src="image" alt="message" /></td>
</tr>
<tr>
<td>DEF</td>
<td><img src="image" alt="DEF" /></td>
</tr>
<tr>
<td>loopDEF</td>
<td><img src="image" alt="loopDEF" /></td>
</tr>
</tbody>
</table>

In general, any operator node, can have any amount of children except for the **loopXOR**, **loopDEF**, and **EVENT** operator nodes. The **loopXOR** and **loopDEF** operator nodes have three children; the leftmost child can be seen as the body of the loop, the middle child is the redo part, after which the leftmost child has to be executed again, and the right child is used to stop executing the loop. The **EVENT** operator node encodes an event and always has exactly one child and can only be directly under a **DEF** operator node, or a **loopDEF** operator node as its middle or rightmost child. [6].

We introduce the following functions on process trees which aid in the calculation of the metrics in Chapter 3:

- **label(n)** - returns the label of node \( n \).
- **children(n)** - returns all direct children of operator node \( n \).
- **rightmostchild(n)** - returns the direct rightmost child of operator node \( n \).

Figure 2.1 shows an example of a process tree. This tree describes a process in which \( a \) is executed first, followed by the execution of \( b \). Then, a loop and \( f \) are executed in parallel. The loop describes that \( \tau \) (the silent activity) or \( c \) is executed at least once. If \( d \) is executed, then
again $\tau$ (leaf labeled “tau”) or $c$ is executed. If $e$ is executed, the execution of the loop is terminated. Finally, when both the loop and $f$ are executed, $g$ is executed after which the process ends. $\sigma_M = \langle a, b, c, d, f, \tau, e, g \rangle$ is a possible execution of the process tree.

In Figure 2.2 another process tree is depicted. This tree describes a process in which $a$ is executed first, followed by the execution of $b$. Then, a multi-choice between a loop and $g$ is made, so either the loop or $g$, or both are executed. The loop describes that $c$ or $d$ is executed at least once. The choice between $c$ and $d$ is based on events, when the MESSAGE event is triggered $c$ is executed, when the TIMEOUT event is triggered $d$ is executed. The loop continues if a TIMEOUT event occurs after which $e$ is executed. The loop is exited when a MESSAGE is received and $f$ is executed. Finally, $h$ is executed after which the process ends. $\sigma_M = \langle a, b, d, e, c, g, f, h \rangle$ is a possible execution of the process tree.

Figure 2.1: An example of a process tree.

Figure 2.2: Another example of a process tree.
The ProcessTreeReplayer package supports process trees discovered with the Inductive Miner \cite{7} and Evolutionary Tree Miner \cite{8} plugins based on the EventNameClassifier ($e = \#_{activity}(e)$). Although, currently, both plugins cannot mine DEF, LOOP\text{DEF}, and EVENT constructs from event logs, all aforementioned process tree operators are supported in the ProcessTreeReplayer package. Furthermore, process trees with duplicate activities are supported as well. Even when they are represented as a DAG, like in Figure 2.3 where A has three incoming edges. In this case we convert the DAG into a tree, where every node has exactly one incoming edge. Therefore, for every leaf $l$ that has $x$ incoming edges ($x > 1$), we create ($x - 1$) leaves having the same label as $l$ for ($x - 1$) of these edges. Note that these newly created leaves all have a unique identifier. For the DAG in Figure 2.3 this results in the tree in Figure 2.4.

![Figure 2.3: A process tree represented as a DAG.](image1)

![Figure 2.4: The unfolded process tree.](image2)

Finally, to compute quality and performance of a process tree, we need to construct an alignment between the two. To this end, we convert the process tree into a Petri net and align the event log with the Petri net. Although, an alignment between process tree and event log can be constructed directly, we need an alignment between Petri net and event log in the computation of one of the quality metrics in Chapter 3.

A process tree can be converted into a Petri net, since each process tree operator has a formal translation to a sound, block-structured workflow net. As the tree is converted, a mapping is maintained between process tree leaves and Petri net transitions. For example, the process tree in Figure 2.1 is translated into the Petri net in Figure 2.5. The black transitions represent the silent split and join transitions of the \textsc{AND} operator.
2.3 Alignment

An alignment between an event log and a process model (in our case a Petri net converted from a process tree) relates events in the log to execution steps of the model [9, 10]. Even if the model contains duplicate and/or silent activities (τ's), the alignment is able to tell to which transition in the Petri net an event belongs to, since each transition has a unique identifier. As the executions of cases are typically performed independently of each other, aligning is performed on the basis of traces.

For each trace in an event log that fits a process model each “move” in the trace, i.e., an event observed in the log, can be mimicked by a “move” in the model, i.e., an action in the model. However, this is not the case if the trace does not fit the model perfectly. We use the symbol ≫ to denote “no move” in either the log or the model. Furthermore, let $A_L$ be the set of events observed in the log and let $A_M$ be the set of all transitions in the model, including silent transitions (represented by τ). Then, a move in log is represented by any $x \in A_L^p$ where $A_L^p = A_L \cup \{\gg\}$, and a move in model is represented by any $y \in A_M^p$ where $A_M^p = A_M \cup \{\gg\}$. Hence, a move is defined as follows:

**Definition 4 (Move).**

A move is a pair $(x, y) \in A_L^p \times A_M^p$, such that:

- $(x, y)$ is a log move if $x \in A_L$ and $y = \gg$,
- $(x, y)$ is a model move if $x = \gg$ and $y \in A_M \setminus \{\tau\}$,
- $(x, y)$ is an invisible move if $x = \gg$ and $y = \tau$,
- $(x, y)$ is a synchronous move if $x \in A_L$, $y \in A_M$, and $x = y$ (i.e., $x$ is mimicked by $y$),
- $(x, y)$ is an illegal move in all other cases.

We use $A_{LM}$ to denote the set of all pairs of legal moves, i.e., all possible pairs of log, model, invisible, and synchronous moves. Let $L$ be a log over $A_L$, let $\sigma_L \in L$ be a trace, let $M$ be a model over $A_M$, and let $\sigma_M \in M$ be a complete execution of the model. An alignment between $\sigma_L$ and $\sigma_M$ is a sequence $\gamma \in A_{LM}^*$ where the projection of the first row (ignoring $\gg$) yields $\sigma_L$ and the projection of the second row yields $\sigma_M$.

Take for example a trace $\sigma_L = (a, b, c, d, e, g, f, h)$ and the model from Figure 2.1, possible alignments between the two are:

$$\gamma_1 = \begin{array}{cccccccc} a & b & c & d & e & \gg & g & f & h \\ a & b & \gg & \gg & g & f & h \end{array}$$

$$\gamma_2 = \begin{array}{cccccccc} a & b & c & \gg & d & e & \gg & g & f & h \\ a & b & c & e & d & e & \gg & g & f & h \end{array}$$

$$\gamma_3 = \begin{array}{cccccccc} a & b & c & d & \gg & \gg & e & g & f & h \\ a & b & c & \gg & e & c & \gg & g & f & h \end{array}$$

Figure 2.5: The resulting Petri net after converting the process tree in Figure 2.1. The boxes indicate the process tree operator nodes.
The set of all possible alignments between $\sigma_L$ and $\sigma_M$ is defined as follows:

**Definition 5 (Set of Alignments).**

$$\Gamma_{\sigma_L,M} = \{ \gamma \in A_{LM} | \exists \sigma_M \in M : \gamma \text{ is an alignment between } \sigma_L \text{ and } \sigma_M \}$$

We want to find the set of alignments with the lowest cost (i.e., in which the least log and/or model moves occur). Therefore, we define a distance function $\delta$ to measure the cost of an alignment.

**Definition 6 (Cost of an Alignment).**

$$\delta : A_{LM} \to \mathbb{N} \text{ where for all } (x,y) \in A_{LM}:$$

$$\delta((x,y)) = \begin{cases} 
0 & \text{if } (x,y) \text{ is a synchronous or invisible move} \\
1 & \text{otherwise}
\end{cases}$$

Note that $x = y \Rightarrow$ is an illegal move and, hence, is not an element of $A_{LM}$.

With this function we can define the set of optimal alignments:

**Definition 7 (Set of Optimal Alignments).**

$$\Gamma^O_{\sigma_L,M} = \{ \gamma \in \Gamma_{\sigma_L,M} | \forall \gamma' \in \Gamma_{\sigma_L,M} \delta(\gamma) \leq \delta(\gamma') \}$$

This set consists of those alignments in $\Gamma_{\sigma_L,M}$ that have the lowest cost. For example, $\{ \gamma_1, \gamma_2 \}$ is the set of optimal alignments between $M$ and $\sigma_L$.

### 2.4 Quality Dimensions

With the constructed alignment we can compute the quality of the process tree compared to the event log. The events in the log are linked to transitions in the Petri net (if possible), and, therefore, also to leaves in the process tree via the mapping between transitions and leaves.

There are four dimensions related to quality: fitness, precision, generalization and simplicity (see Figure 2.6) [2]. When a process model has a good balance between those four dimensions its quality is high.

![Quality Dimensions Diagram](image)

**Figure 2.6**: The four quality dimensions.

Fitness indicates the degree to which the discovered model allows for the behavior seen in the event log. When a model can replay all behavior seen in the log, its fitness is high. Precision indicates the degree to which the discovered model allows for behavior that is completely different than seen in the event log. If the model only allows for behavior seen in the log, its precision is high. Generalization indicates the degree to which the discovered model generalizes the example behavior seen in the event log. If the model allows for other behavior than seen in the event log, generalization is high. Simplicity indicates how simple the discovered model is with respect to complexity. The simplest model that can describe the behavior seen in the log, is the best model. This principle is also known as Occam’s Razor, which states that “one should not increase, beyond what is necessary, the number of entities required to explain anything” [2].
2.5 Multisets

In this thesis, multisets are used to represent the state of an alignment automaton, which we need in the calculation of one of the quality dimensions (see Chapter 3).

$\mathcal{B}(A)$ is the set of all multisets over some set $A$. For some multiset $b \in \mathcal{B}(A)$, $b(a)$ denotes the number of times element $a \in A$ appears in $b$. For example, take $A = p, q, r$, then:

- $b_1 = [],$ denotes the empty multiset,
- $b_2 = [p, p, q],$ denotes the multiset over $A$ where $b_2(p) = 2$ and $b_2(q) = 1$,
- $b_3 = [p, q, r],$ denotes the multiset over $A$ where $b_3(p) = b_3(q) = b_3(r) = 1$,
- $b_4 = [p, p, q, p, q, r],$ denotes the multiset over $A$ where $b_4(p) = 3$, $b_4(q) = 2$, and $b_4(r) = 1$,
- $b_5 = [p^3, q^2, r] = b_4$, i.e., the ordering of elements is irrelevant and a more compact notation may be used for repeating elements.

The standard set operators can be extended to multisets, e.g., $p \in b_2$, $b_2 \uplus b_3 = b_4$, $b_5 \setminus b_2 = b_3$, and $|b_5| = 6$.

This chapter described the preliminary knowledge required in the remainder of this thesis. The next chapter describes how quality and performance is measured in process trees.
This chapter gives an overview of the metrics implemented in the quality and performance dimensions. Beside these metrics, some additional metrics are discussed that were developed during the implementation of the other metrics, e.g., the useless nodes metric is used in the computation of the simplicity metric in the quality dimension. These metrics are specifically developed for process trees so that we can compute statistics (e.g., fitness, execution times, frequency, etc.) not only for the process tree as a whole, but also for parts (subtrees) of the process tree (first goal of this project). These statistics are then visualized on the process tree (see Chapter 4). The metrics discussed are all defined for nodes, edges just get assigned the value obtained for their target node.

We start with a section about constructing the alignment between process tree and event log, which is needed to link events in the event log to leaves in the process tree, and, subsequently, extract information from the log which is used in the calculation of the metrics.

3.1 Aligning Process Tree and Event Log

In order to link events in the event log to activities in the process tree, we need to construct an alignment between the two (see Section 2.3). Before we construct the alignment, some preprocessing of the process tree is necessary.

3.1.1 Extend Process Tree and Convert to Petri Net

The process trees supported by the ProcessTreeReplayer package contain leaves which are labeled with activity names. However, when we construct the alignment between the process tree and the event log where activities have multiple instances, e.g., an event with lifecycle transition “start” and one with “complete” for the same activity (see Section 2.1), then only one event can be mimicked in the model and the others will become log moves. Take, for example, the process tree in Figure 3.1 and the trace $\sigma_L = \langle A_S, A_C, B_S, B_C \rangle$, where the subscripts $S$ and $C$ stand for the “start” and “complete” lifecycle transitions, respectively. Then, this will be the resulting alignment:

\[
\gamma = \begin{bmatrix}
A_S & A_C & B_S & B_C \\
A & B & \gg & \gg
\end{bmatrix}
\]

Therefore, before constructing the alignment, we look in the event log which events have a lifecycle transition attribute with value “start”. For these events, all leaves in the process tree with the same activity name are transformed as in Figure 3.2. This is based on the method proposed in Chapter 9 of [10]. The result of this transformation is that more activity instances, i.e., events, can be mapped on process tree leaves and, therefore, the model becomes more accurate. We only
do this for the “start” and “complete” lifecycle transitions, because these are the most common and we need them in the calculation of the different time metrics (see Section 3.3.1).

While transforming the process tree, a mapping is maintained between the original leaves and the start and complete variants of this leaf, because the original tree will be visualized in the end, so we need to translate the transformed tree back to the original one.

Because we need a Petri net together with the alignment for the precision metric (see Section 3.2.2), we convert the process tree into a Petri net and construct the alignment between Petri net and event log after the transformations. As the process tree is converted into a Petri net a mapping between process tree nodes and Petri net transitions and vice versa is maintained. With this mapping we can link the synchronous silent, and model moves in the alignment to the leaves in the process tree.

3.1.2 Construct Alignment

Finally, the alignment is constructed between Petri net and event log. This is based on a classifier; we use the EventNameClassifier when the event log does not contain events with the lifecycle transition attribute. If the lifecycle transition attribute is present, we use a classifier which classifies events based on event name together with lifecycle transition.

The resulting alignment is an arbitrary optimal alignment out of the set of optimal alignments. With this alignment we can compute the quality of the model and extract additional information from the event log, for instance to compute execution times of the activities.

3.2 Measuring Quality of Process Trees

The metrics developed for computing a value for the four quality dimensions are discussed in this section.

3.2.1 Fitness

Fitness indicates the degree to which the discovered model allows for the behavior seen in the event log. It is expressed on a scale from 0 to 1, where 0 means the model allows for no behavior seen in the log, and 1 means the model allows for all behavior seen in the log.
Fitness is directly related to the number of synchronous, model, and log moves in the alignment between model and log. If the alignment consists of only synchronous moves, it means the model can mimic every event observed in the log with the execution of a corresponding action in the model. Hence, the model allows for all behavior seen in the log and has perfect fitness. However, if the alignment contains some model and/or log moves, this means the model could not mimic the behavior in the log, hence, fitness decreases. The more log and/or model moves the lower the fitness of the model.

From the above we recursively define fitness for nodes in the process tree. First the fitness of leaves is calculated, then the fitness of operators is calculated by using the fitness of their children. Fitness for a leaf $l$ is defined as follows:

**Definition 8 (Fitness - Leaf).**

$$\text{Fitness}(l) = \begin{cases} \#\text{syncMoves}(l) \\ \text{Unknown} \end{cases} \frac{\#\text{syncMoves}(l) + \#\text{modelMoves}(l) + \#\text{logMoves}(l)}{\#\text{syncMoves}(l) + \#\text{modelMoves}(l) + \#\text{logMoves}(l)}$$

where $\#\text{syncMoves}(l)$, $\#\text{modelMoves}(l)$, and $\#\text{logMoves}(l)$ are the total number of synchronous, model, and log moves on leaf $l$, respectively. See Section 3.4.2 for an explanation on how log moves are projected on process tree nodes.

The result of this function is a number between 0 and 1, where 0 means no synchronous moves occurred on $l$, and 1 means all moves on $l$ were synchronous. This reflects the idea of fitness discussed earlier. Leaves labeled with $\tau$ have no fitness, since they are not really part of the process and cannot be linked to any event observed in the log anyway.

Fitness of operator nodes is determined by computing a weighted average over the fitness of their children, excluding leaves labeled $\tau$ for the same reason as before. This results in the following recursive definition for the fitness of an operator node $n$:

**Definition 9 (Fitness - Operator node).**

$$\text{Fitness}(n) = \frac{\{c \in \text{children}(n) \mid \text{label}(c) \neq \tau\} \text{Fitness}(c) \cdot (\#\text{syncMoves}(c) + \#\text{modelMoves}(c) + \#\text{logMoves}(c))}{\{c \in \text{children}(n) \mid \text{label}(c) \neq \tau\} \cdot (\#\text{syncMoves}(c) + \#\text{modelMoves}(c) + \#\text{logMoves}(c)) + \#\text{logMoves}(n)}$$

In this definition, fitness is further decreased if there were log moves projected on the operator node itself (+ $\#\text{logMoves}(n)$ in the denominator).

### 3.2.2 Precision

Precision indicates the degree to which the discovered model allows for behavior that is completely different than seen in the event log. It is expressed on a scale from 0 to 1, where 0 means the model allows for much completely different behavior than seen in the log, and 1 means the model allows for no other behavior than the behavior seen in the log.

The metric used for calculating precision is taken from [11]. In this paper an alignment automaton is constructed based on a Petri Net, event log, and corresponding alignment. We use the unordered, representative, forward-based variant of the alignment automaton in this thesis. This means we see a state as a multiset (see Section 2.5) of transitions; every state represents a sequence of transitions. We use a single optimal alignment for constructing the alignment automaton, and we use the prefixes of the complete activity sequences to build the automaton. There are two reasons for this: (1) the obtained alignment automaton is more compact, because states can be merged when states contain the same transitions (duplicate and silent transitions can be distinguished by their identifier), and (2) computation time, since obtaining all optimal alignments and constructing both the forward- and backward-based alignment automaton are very time consuming. A drawback of only computing the forward-based alignment automaton is that the result might be biased when there exist long-distance dependencies between activities.

In Figure 3.3 the alignment automaton for the process tree in Figure 3.4 discovered from the event log in Table 3.1 is depicted. We see that each state consists of a prefix (first line in the
state’s label), and a weight (second line in the state’s label). The weight is based on the number of traces of which the state is a prefix, e.g., the state “A,C 50.0” is a prefix of the trace $\sigma_L = ACD$ which occurs 50 times in the event log, hence, the weight of this state is 50. An arc defines the concatenation between the prefixes of the two states the arc connects, e.g., states “A 50.0” and “A,C 50.0” are connected by an arc labeled “C”. Furthermore, the empty prefix is the initial state, its weight is equal to the total number of traces in the event log. Finally, the black states indicate imprecisions in the model, i.e. this state can be reached in the model, but does not occur in the event log. For example, if we look at the black state with incoming arc labeled “E”, this corresponds to the complete execution of the model $\sigma_M = ACE$. However, this behavior is not seen in the event log, hence, it is an imprecision in the model.

![Figure 3.3: Example of an alignment automaton.](image)

**Table 3.1: A simple event log.**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BCE</td>
<td>50</td>
</tr>
<tr>
<td>ACD</td>
<td>50</td>
</tr>
</tbody>
</table>

![Figure 3.4: Process tree discovered from the event log in Table 3.1.](image)
The metric proposed in [11] can also be used for computing precision of subtrees when we only consider states in the automaton which have outgoing arcs with labels having the same name as leaves in the subtree. The weight of a state $s$ is determined based on the number of traces that have $s$ as prefix in the alignment. This results in the following definition for precision:

**Definition 10 (Precision).**

$$
\text{Precision}(n) = \begin{cases} 
\sum_{s \in Q_n} \omega(s) \cdot |e_x(s)| & \text{if } n = \text{operator node} \\
\sum_{s \in Q_n} \omega(s) \cdot |a_v(s)| & \text{Unknown}
\end{cases}
$$

where $Q_n$ is the set of states which have outgoing arcs with labels having the same name as leaves in $n$’s subtree, $\omega(s)$ is the weight of state $s$, $e_x(s)$ is the set of executed actions, i.e., activities really executed, and $a_v(s)$ is the set of available actions, i.e., transitions enabled in the Petri net, but never fired at this point in the trace.

Note that leaves have no precision since precision indicates whether the structure of a model or part of a model is precise or not. A leaf on its own has no structure, so we cannot determine its precision.

### 3.2.3 Generalization

Generalization indicates on a scale from 0 to 1 the degree to which the discovered model generalizes the example behavior seen in the event log. When the model allows for no other behavior than the example behavior seen in the log, generalization is close to 0. When the model allows for much other behavior than seen in the log, generalization is close to 1.

The generalization metric was already defined for process trees in [8] and is slightly modified to apply it to subtrees and leaves:

**Definition 11 (Generalization).**

$$
\text{Generalization}(n) = 1 - \frac{\sum_{m \in \text{nodes}(n)} (\sqrt{\text{Frequency}(m)})^{-1}}{\text{nodes}(n)},
$$

where $\text{nodes}(n)$ is the set of nodes, including $n$, in $n$’s subtree, and $\text{Frequency}(m)$ is the number of executions of node $m$ (see Section 3.4.1).

The computation of the frequency of a node is discussed in Section 3.4.1.

### 3.2.4 Simplicity

Simplicity indicates on a scale from 0 to 1 how simple the discovered model is with respect to complexity.

The simplicity metric was already defined for process trees in [12, 13]. It is slightly modified so that it can be applied to subtrees and leaves:

**Definition 12 (Simplicity).**

$$
\text{Simplicity}(n) = 1 - \min(1, \frac{\#\text{uselessNodes}(n) + \#\text{duplicateNodes}(n)}{\text{nodes}(n)}),
$$

where $\text{nodes}(n)$ is the set of nodes, including $n$, in $n$’s subtree, and $\#\text{uselessNodes}(n)$ and $\#\text{duplicateNodes}(n)$ are the number of useless and duplicate nodes in $n$’s subtree, including $n$.

Since a node can be both useless and duplicate, the result of the fraction could be larger than 1, resulting in a negative simplicity. Hence, we take the minimum of 1 and the fraction to account for this. Duplicate nodes are leaves having the same activity name as another leaf in the tree. A node is useless if it can be reduced without changing the behavior of the process tree [12]. The criteria for a node to be useless are discussed in Section 3.4.3.
3.3 Measuring Performance in Process Trees

There are lots of dimensions related to performance: time, resource usage, costs, etc. Because we cannot treat them all in this project, only the time dimension is considered in this section. This dimension consists of execution, waiting, sojourn, and synchronization time calculation for the activities in the process model. This section describes how these are calculated for nodes in a process tree. We illustrate the calculation of these time metrics with the trace in Table 3.2, from now on referred to as exampleTrace, and the process tree depicted in Figure 3.5, from now on referred to as exampleTree.

Table 3.2: Trace with timestamps and lifecycle transitions.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Timestamp</th>
<th>Lifecycle Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2014-10-01T14:30:00</td>
<td>complete</td>
</tr>
<tr>
<td>B</td>
<td>2014-10-01T14:35:00</td>
<td>complete</td>
</tr>
<tr>
<td>H</td>
<td>2014-10-01T14:41:00</td>
<td>complete</td>
</tr>
<tr>
<td>F</td>
<td>2014-10-01T14:44:00</td>
<td>start</td>
</tr>
<tr>
<td>F</td>
<td>2014-10-01T14:46:00</td>
<td>complete</td>
</tr>
<tr>
<td>D</td>
<td>2014-10-01T14:47:00</td>
<td>start</td>
</tr>
<tr>
<td>I</td>
<td>2014-10-01T14:52:00</td>
<td>complete</td>
</tr>
<tr>
<td>D</td>
<td>2014-10-01T14:53:00</td>
<td>complete</td>
</tr>
<tr>
<td>G</td>
<td>2014-10-01T15:56:00</td>
<td>complete</td>
</tr>
</tbody>
</table>

Figure 3.5: Process tree used to illustrate the calculation of the time metrics.

3.3.1 Time Metrics

As mentioned in Section 2.1 events can have attributes, if an event contains the timestamp attribute together with the lifecycle transition attribute we can use this to compute execution and/or waiting time of the executed activity. If an event has the timestamp attribute, but no lifecycle transition attribute, we assume this timestamp represents the lifecycle transition “complete”.

18 Static and Dynamic Visualization of Quality and Performance Dimensions on Process Trees
CHAPTER 3. MEASURING QUALITY AND PERFORMANCE ON PROCESS TREES

**LastCompletedSequentialLeaf**

We determine for each leaf the leaf that was executed before it using the alignment. This is not always the direct predecessor of the leaf in the alignment, because, when the model contains parallelism and we have a leaf \( p \) that is the direct predecessor of a leaf \( l \), but \( p \) is in parallel with \( l \), the completion of \( p \) has no influence on the start of \( l \). To this end we use the process tree.

Say we want to compute the waiting time for a leaf \( m \), then we look for the last synchronous move in the alignment for which can be determined that the corresponding leaf, say \( n \), was really executed before \( m \). To this end we introduce the **LowestCommonParent (LCP)** as the common parent of two nodes that is deepest in the process tree of all common parents of the two nodes. This parent indicates the order between the two nodes; \& and \texttt{or} indicate a parallel order between their subtrees, all other operators a sequential order. Now, if the LCP of \( m \) and \( n \) is not the \& or \texttt{or} or operator, we call \( n \) the LastCompletedSequentialLeaf (LaCoSeL) of \( m \) and of each operator node \( o \) for which \( m \) is the starting leaf (see next paragraph).

Take the following alignment for the process tree in Figure 3.5 and trace in Table 3.2, where the subscript indicates the start of an event (\( S \)) or the completion of an event (\( C \)):

\[
\gamma = \begin{array}{cccccccc}
A & B & C & H & F & C & D & G \\
A & B & C & H & F & C & D & G \\
\end{array}
\]

Now, if we want to determine the LaCoSeL of leaf \( D \), we go back in the alignment step by step:

1. \( E \): model move, so we move further back
2. \( F \): \texttt{LCP}(\( D, F \)) = \texttt{AND}, so we move further back (\( F \) is mapped to \( F \))
3. \( F \): \texttt{LCP}(\( D, F \)) = \texttt{AND}, so we move further back
4. \( H \): log move, so me move further back
5. \( B \): \texttt{LCP}(\( D, B \)) = \texttt{SEQ}, so \( B \) becomes LaCoSeL of \( D \)

Table 3.3 lists the LaCoSeL of each node in exampleTree.

<table>
<thead>
<tr>
<th>Node</th>
<th>LaCoSeL</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>unknown (no preceding leaf)</td>
</tr>
<tr>
<td>( B )</td>
<td>( A )</td>
</tr>
<tr>
<td>( D )</td>
<td>( B )</td>
</tr>
<tr>
<td>( E )</td>
<td>unknown (model move)</td>
</tr>
<tr>
<td>( F )</td>
<td>( B )</td>
</tr>
<tr>
<td>( G )</td>
<td>( D )</td>
</tr>
<tr>
<td>( I )</td>
<td>( F )</td>
</tr>
<tr>
<td>\texttt{SEQ} (root)</td>
<td>unknown (no preceding leaf)</td>
</tr>
<tr>
<td>\texttt{AND}</td>
<td>( B )</td>
</tr>
<tr>
<td>\texttt{SEQ}</td>
<td>( B )</td>
</tr>
</tbody>
</table>

**Mapping Timestamps on Nodes**

With the extension applied to process trees as discussed in Section 3.1.1, the lifecycle transitions “start” and “complete” of an event are aligned to the “start” and “complete” leaves in the process tree. Subsequently, these are mapped to the leaf in the original tree.

Similarly, we map “start” and “complete” timestamps to operator nodes. We determine for each operator node \( n \) the starting and closing leaf of \( n \) in each trace in the event log. Therefore, we look at all parents up to the root of the tree for each leaf corresponding to an event in a
CHAPTER 3. MEASURING QUALITY AND PERFORMANCE ON PROCESS TREES

Depending on whether an operator is added or removed compared with the parents of the previous event, a construct is respectively started or completed. When an operator is added, the leaf corresponding to the current event is the start of that operator, when an operator is removed, then the leaf corresponding to the previous event is the end of that operator. The “start” timestamp of the starting leaf and the “complete” timestamp of the closing leaf are mapped to $n$.

An exception to this method is when one of the added parents is an AND or OR operator. In this case we apply above method to each branch separately, because in a parallel execution we can switch between branches. In our example this is the case when $F$ is executed in exampleTrace followed by $D$, the parents of $F$ are SEQ, AND, and SEQ (root), the parents of $D$ are AND and SEQ (root). We see that SEQ is removed, but $F$ is not the closing leaf of SEQ. Therefore, we look at each branch separately and then $I$ becomes the closing leaf of SEQ.

The starting and closing leafs for the operators in the process tree in our example are listed in Table 3.4.

<table>
<thead>
<tr>
<th>Operator</th>
<th>starting leaf</th>
<th>closing leaf</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQ (root)</td>
<td>A</td>
<td>G</td>
</tr>
<tr>
<td>AND</td>
<td>F</td>
<td>D</td>
</tr>
<tr>
<td>SEQ</td>
<td>F</td>
<td>I</td>
</tr>
</tbody>
</table>

Finally, we introduce nodeStamps, a triple which contains three timestamps: the “complete” timestamp of the LaCoSeL of a node $n$ in trace $t$, the “start” timestamp mapped to $n$ in $t$, and the “complete” timestamp mapped to $n$ in $t$. Because nodes can have more than one timestamp (even in the same trace), we introduce the function getTimestamps($n,t,i$), which returns, given a node $n$, a trace $t$, and an index $i$ the nodeStamps of $n$. The index $i$ points to the $i^{th}$ execution of $n$ in $t$. The nodeStamps for our example are listed in Table 3.5. ⊥ denotes the timestamp does not exist in the trace. For example, getTimestamps($A$, exampleTrace, 1).complete returns the “complete” timestamp of leaf $A$ in exampleTrace at index 1 (first, and only, execution of $A$), which is 2014-10-01T14:30:00.

Execution Time

Execution time of an activity is defined as the time between the start and finish of its execution. We can compute the execution time of a node $n$ in a trace $t$ at an index $i$ as follows:

$$\text{ExecutionTime}(n,t,i) = \text{getTimestamps}(n,t,i).\text{complete} - \text{getTimestamps}(n,t,i).\text{start}$$

Note, when one of the two timestamps is missing, execution time cannot be computed. In our example, we can compute the execution times of leaves $D$ and $F$ in exampleTrace at index $i = 1$.
(only one execution of events \(D\) and \(F\) in the trace), which are 3 and 2 minutes, respectively. Furthermore, the execution times of operator nodes AND, and SEQ can also be calculated, these are 9 and 8 minutes, respectively.

**Waiting Time**

Waiting time is defined as the time between the completion of an activity and the start of the next one. For computing the waiting time, given a node \(n\), trace \(t\), and index \(i\), we need two timestamps: the “start” timestamp of \(n\) in \(t\) at \(i\), and the “complete” timestamp of the LaCoSeL of \(n\) in \(t\) at \(i\).

\[
\text{WaitingTime}(n,t,i) = \text{getTimestamps}(n,t,i).\text{start} - \text{getTimestamps}(n,t,i).\text{LaCoSeL}
\]

Note that it holds that when one of the two timestamps is missing, no waiting time can be computed.

In our example, we can compute the waiting time of leaves \(D\) and \(F\) in exampleTrace at index \(i = 1\), which are 12 and 9 minutes, respectively. Furthermore, the waiting time of AND and SEQ in exampleTrace at index \(i = 1\) is 9 minutes.

**Sojourn Time**

Sojourn time is the waiting time plus the execution time of an activity, i.e., the time between the completion of an activity and the completion of the next activity. The sojourn time of a node \(n\) (either a leaf or an operator node) in a trace \(t\) at an index \(i\) can be computed as follows:

\[
\text{SojournTime}(n,t,i) = \text{WaitingTime}(n,t,i) + \text{ExecutionTime}(n,t,i)
\]

Note that we only need the LaCoSeL and “complete” timestamp of \(n\) in \(t\) at \(i\) to compute its sojourn time.

In our example, the sojourn time of the leaves \(B\), \(D\), \(F\), \(I\), and \(G\), and the operators AND and SEQ in exampleTrace at index \(i = 1\) can be calculated. These are 5, 18, 11, 6, and 3 minutes, and 18 and 17 minutes, respectively.

**Synchronization Time**

Synchronization time can only be computed for operator nodes labeled with AND or OR. It is defined as the time between the branch that is completed earliest and the branch that is completed latest in the trace. This is determined based on the “complete” timestamps of all subtrees of the parallel operator \(p\) in a trace \(t\), at an index \(i\):

\[
\text{SynchronizationTime}(p,t,i) = \max_{c \in \text{children}(p)} (\text{getTimestamps}(c,t,i).\text{complete}) - \min_{c \in \text{children}(p)} (\text{getTimestamps}(c,t,i).\text{complete})
\]

Note that this formula only applies to operator nodes labeled with AND, because their children are executed as many times as the node itself. The OR operator is not supported yet, because children of an OR operator node do not need to be executed as many times as the node itself. Therefore, the \(i^{th}\) execution of an OR operator node is not necessarily the \(i^{th}\) execution of its children.

The synchronization time of the AND operator in exampleTrace at index \(i = 1\) is equal to the “complete” timestamp of \(D\) in exampleTrace at index \(i = 1\) minus the “complete” timestamp of SEQ in exampleTrace at index \(i = 1\), which is 1 minute.
3.4 Additional metrics

In this section metrics developed during the implementation of the other metrics are discussed.

3.4.1 Frequency

The frequency of a leaf node is the number of times the node is executed in the model over all traces according to the alignment. We look at a complete execution of the model and count the number of occurrences of the leaf, i.e., the synchronous moves + model moves on the leaf. Let \( l \) be a leaf then its frequency is defined as follows:

**Definition 13 (Frequency - Leaf).**

\[
\text{Frequency}(l) = \#\text{syncMoves}(l) + \#\text{modelMoves}(l) + \#\text{silentMoves}(l)
\]

where \( \#\text{syncMoves}(l) \), \( \#\text{modelMoves}(l) \), and \( \#\text{silentMoves}(l) \) are the total number of synchronous, model, and silent moves on leaf \( l \).

If the leaf was transformed (see Section 3.1.1), we divide the resulting frequency by 2, since the moves on the start and complete leaf in the alignment are all projected on the original leaf, but an execution of start and complete is just a single execution of the activity.

Frequency of operators is defined recursively for an operator node \( n \), except for the \( \text{or} \) operator:

**Definition 14 (Frequency - Operator node).**

\[
\text{Frequency}(n) = \begin{cases} 
\text{Frequency}(c), \text{with } c \in \text{children}(n) & \text{if } \text{label}(n) \in \{\text{SEQ, AND, EVENT}\} \\
\text{Frequency(rightmostchild(n)))} & \text{if } \text{label}(n) \in \{\text{LOOPXOR, LOOPDEF}\} \\
\sum_{c \in \text{children}(n)} \text{Frequency}(c) & \text{if } \text{label}(n) \in \{\text{XOR, DEF}\}
\end{cases}
\]

For the \( \text{or} \) operator a different approach is followed, since we cannot determine its frequency based on the frequency of its children. Therefore, we use the Petri net converted from the process tree. When an \( \text{or} \) operator is converted, additional transitions are introduced to represent the multi-choice between the events (black transitions in Figure 3.6). These transitions are mapped to the corresponding \( \text{or} \) operator in the process tree. If we count the number of occurrences of one of these black transitions in the alignment we know how often the \( \text{or} \) operator is executed in all traces, and hence, the frequency.

![Figure 3.6: Conversion of the or operator into a Petri net.](image)

3.4.2 Positioning Log Moves

Log moves are moves on events in the log that cannot be matched by an action in the model. As a consequence, they cannot be linked to a node in the process tree. Log moves indicate the model cannot replay the behavior in the log and, therefore, have an influence on the model’s fitness. To account for log moves in the fitness calculation of the model we project log moves on nodes in the tree.
We look at the synchronous moves that occur right before and right after the log move in the alignment. These moves have a matching element in the model so that we can narrow down the area in which the log move occurred. We cannot use model moves to say something about the position of a log move in the model, because when we see a log move followed by a model move in the alignment, we can construct another alignment with the same cost where the log move and the model move are switched.

We introduce the `PreviousMatchingElement (PME)` as the leaf in the process tree that corresponds to the last synchronous move before a log move, and `NextMatchingElement (NME)` as the leaf in the process tree that corresponds to the first synchronous move after a log move. To indicate the area (i.e., subtree) in which the log move occurred in the process tree, we project a log move on the:

- PME, if PME has the same name as the event on which the log move occurred.
- NME, if NME has the same name as the event on which the log move occurred.
- LCP of PME and NME, if PME and NME exist.
- Root, if PME or NME does not exist.

The first two cases occur when the activity in the event log has multiple instances and the model does not account for them. Furthermore, if the log move is projected on the LCP or the root, we know for sure the log move has occurred within this subtree (i.e., could not have occurred in a higher subtree), because of the hierarchical structure of process trees. It could have occurred in a lower subtree, but we cannot be certain. An exception to this statement are the AND and OR operators. When the PME, NME, and LCP are all part of a subtree which has an AND or OR as root, then the log move could have occurred in a higher subtree, because the activity could be parallel to all activities in the subtree. A possible fix is to project every log move that occurs within a parallel operator on this operator. However, this may result in a projection that is too coarse (e.g., in a tree which has an AND as root). This drawback outweighs the benefit, so for now, we do not apply this fix and take this exception for granted.

We illustrate the projection of log moves with an example: take the process tree in Figure 3.7 and the trace \( \sigma_L = \langle A, B, C, E, F, H, I \rangle \), then we get the following alignment:

\[
\gamma = \begin{array}{cccccccc}
A & B & C & E & F & H & I \\
A & B & C & \gg & F & H & I
\end{array}
\]

Following above method the log move on \( E \) is projected on the LCP of the PME (C) and NME (F), which is the SEQ operator highlighted in Figure 3.7. Because the log move occurred between two activities in the SEQ construct, we know the log move has occurred somewhere within this subtree. Therefore, we project the log move on the SEQ to indicate there is a deviation within this subtree with respect to the event log.

When the LCP is a choice operator (XOR or DEF), we do not project the log move on this node. The PME and the NME are in this case in different branches of the choice, because otherwise a lower common parent can be found. Hence, the choice is executed more than once in the trace. So some sort of LOOP is missing in the model. Therefore, we project the log move on the parent of the choice operator, because in this part of the tree something is missing that allows for multiple execution of the choice operator. For example: take the model in Figure 3.7 and the trace \( \sigma_L = \langle A, B, C, E, D, F, H, I \rangle \) then we get the following possible alignment (there are two other possible alignments where the log and model moves are in a different order):

\[
\gamma = \begin{array}{cccccccc}
A & B & C & E & \gg & \gg & D & F & H & I \\
A & B & C & \gg & F & G & D & F & H & I
\end{array}
\]

Now, C is the PME and D the NME of the log move on E. Then, the log move is projected on the SEQ instead of the XOR, because something is missing in this part of the tree that allows the XOR to be executed more than once.
3.4.3 Useless Nodes

A node is useless if it can be reduced without changing the behavior of the process tree [12]. Useless nodes are:

1. operators with only one child, except for the message and timeout,
2. \( \tau \)'s which are a direct child of a seq or and,
3. any additional \( \tau \) as direct child of a xor when xor already has a \( \tau \) as direct child,
4. loop operators consisting of a loop as leftmost child and a \( \tau \) as middle and rightmost child,
5. \( \tau \)'s that are the rightmost child of a loop, if the loop has a seq as parent and is not the rightmost child of that seq
6. seq operators that have two children of which the left child is a loop with a \( \tau \) as rightmost child.

The first four criteria were already defined in [12], the last two are developed during this project. In 5 the \( \tau \) can be replaced by the right sibling of the loop. In 6 the seq can be removed, because the right child can replace the \( \tau \) in the loop, after which the seq has only one child left.

3.5 Discussion

This chapter described the first goal of our project. In order to conduct quality and performance analyses on process trees, we constructed an alignment between process tree and event log, and defined quality, performance, and some additional metrics for process trees. With these metrics we can compute values for each node and edge in the process tree, and, therefore, say something about parts of the model as well. For the time metrics an average, minimum, and maximum value
CHAPTER 3. MEASURING QUALITY AND PERFORMANCE ON PROCESS TREES

Figure 3.8: Taxonomy of the metrics defined in this chapter.

is computed, since nodes usually are executed more than once. An overview of these metrics is
given in Figure 3.8. Of course, this is just a subset of metrics which can be used. Other metrics
for these or other dimensions can be defined, such as resource usage, costs, etc. These can be
added to the ProcessTreeReplayer package so that these are also computed for nodes and edges in
a process tree.

The next chapter describes how values obtained from a metric are projected on each node
and edge in the process tree. Furthermore, the alignment constructed in this chapter is used to
determine for each trace a path through the process tree, which is used in the animation of tokens
on a process tree.
Chapter 4

Static and Dynamic Visualizations on Process Trees

In the previous chapter we defined a number of metrics on the process tree. The values computed for these metrics can be visualized on the process tree. The following sections discuss the design and implementation of the frameworks for the static and dynamic visualizations. The process tree used throughout this chapter is discovered with the Inductive Miner (Variant: Inductive Miner - infrequent, Noise threshold: 0.50) from the repairexample event log taken from processmining.org.

4.1 Static Visualizations Framework

In the static visualizations framework values of a metric in the quality or performance dimension are projected on the visual primitives of the process tree. These primitives relate to the nodes and edges of the tree; nodes have a color, a size, a label (only leaves), a border color, and a border width. Edges have a color, a size, and a label. The framework should be as generic as possible; the user should be able to project all metrics on any of the aforementioned visual primitives. Furthermore, the user should be able to select an appropriate color scheme.

4.1.1 Normalization of Measurements

Before we can use the metric results in the visualization, we normalize the values obtained for each node such that the minimum corresponds to 0 and the maximum to 1. This is done with the following formula:

\[
\text{Normalize}(v) = \frac{v - \min_{u \in \text{Values}}(u)}{\max_{u \in \text{Values}}(u) - \min_{u \in \text{Values}}(u)}
\]

where Values is the set of all values obtained for each node, the values for nodes for which no value could be calculated are excluded.

Note that if both maximum and minimum have the same value (i.e., there is only one value, or all values are equal), then this value is normalized to 1. This formula is applied to the performance and counting metrics only, since the values for the metrics in the quality dimension (see Section 3.2) are already between 0 and 1. This also holds for the “Useless node” and “Duplicate node” metrics, since they have only two values: 1 means the node is useless/duplicate, and 0 means the node is not.
4.1.2 Projecting Measurements on a Process Tree

The normalized values are projected on the different graphical elements of a process tree.

**Assigning Colors**

Different color schemes are implemented from which the user can pick a suitable one for the metric they want to visualize. These color schemes are picked from ColorBrewer, which is a tool for selecting specific color schemes in RGB, CMYK, Hex, Lab, and HSV [14]. Each scheme consists of ten colors representing a tenth of the whole range of values calculated for all nodes. One extra color is added to each scheme to represent that the value is unknown, e.g., when precision is projected on the node color all leaves get this color, because precision is unknown for leaves. This way every node or edge gets assigned a color to it.

A color scheme can be set for node, border, and edge color separately, so three different color schemes can be used at the same time. Furthermore, a legend indicates the range of values represented by each color in the scheme (see Figure 4.1).

![Color Scheme Legend](image)

(a) Legend for one of the quality metrics.

(b) Legend for one of the model metrics (“Useless node” and “Duplicate node”).

(c) Legend for one of the performance and counting metrics.

Figure 4.1: Example legends, all using the same color scheme.

![Tree Visualization](image)

Figure 4.2: Example of a projection of “Average execution time” on the node color, “Useless node” on the border color, and “Frequency” on the edge color. Corresponding legends are displayed below the tree.
The legend for the quality dimensions has a fixed range from 0 to 1 (see Figure 4.1a). In Figure 4.1b the legend for the “Useless node” and “Duplicate node” metrics is shown. Only the first and last colors in the scheme are used, the first indicating the node is useless/duplicate (“Yes”), the last indicating it is not (“No”). For all other metrics a legend like the one in Figure 4.1c is used. The not normalized intervals are displayed on each color.

Figure 4.2 depicts the projection of “Average execution time”, “Useless node”, and “Frequency” on the node, border, and edge color respectively.

**Adjust Sizes**

Each node, border, and edge has a default size, depending on the value of the metric the size is adjusted. The higher the value the larger the node, border, or edge gets. The new size for an element $e$ (node, border, or edge) is calculated as follows:

$$ \text{NewSize}(e) = \text{DefaultSize}(e) + \text{Value}(e) \times \text{MAXSIZE} $$

where $\text{Value}(e)$ is the value of the result of the metric for $e$ (normalized if applicable), and $\text{MAXSIZE}$ is a constant indicating the maximum increase in size.

In Figure 4.3a an example of the projection of three different metrics on the node, border, and edge size is shown.

![Example of a projection of “Frequency” on the node size, “Useless node” on the border size, and “#Log moves” on the edge size.](image)

![Example of a projection of “Frequency” on edge label and “Fitness” on leaf label.](image)

**Set Labels**

Edges and leaf nodes can have a label. This label shows for edges the value for the metric projected on it and for leaf nodes the name of the metric plus its value. Figure 4.3b depicts the projection of “Frequency” and “Fitness” on the edge labels and leaf labels respectively.

Figure 4.3: Examples of projections on node, border, and edge size, and on edge and leaf label.
4.1.3 Resulting Framework

The final static visualizations framework is depicted in Figure 4.4. Some additional options were added: the user can zoom in and out on the process tree ("Scale slider") and can filter out leaf nodes based on frequency ("Frequency slider"). When the frequency of a leaf node falls below the threshold this node is removed from the visualization. When an operator node has no leaves left, this operator node is also removed. With this option the user can see the most frequent executed parts of the tree. Furthermore, when the user hovers over a node in the tree, a table shows for some of the metrics the values calculated for that particular node ("Metrics table").

In the "Projection settings pane" the user can choose which metric is to be projected on which visual primitive. The "Color scheme settings pane" lets the user choose which color scheme is to be used for the projection on node, border, or edge color. The resulting visualization is drawn on the Drawing pane.

Figure 4.4: The static visualizations framework.
4.2 Dynamic Visualizations Framework

The dynamic visualizations framework shows an animation of the replay of an event log on the process tree (Token replay). Each token represents a single case in the process and moves with a certain speed depending on the execution time of a certain task in the process or the waiting time between two tasks. The establishment of this animation is done in several steps: first, for each trace a path through the tree is constructed (see Section 4.2.1), then time is added to these paths (see Section 4.2.2), and finally, tokens are drawn on the edges and nodes in the tree (see Section 4.2.3). We illustrate these steps with a running example using the trace in Table 4.1 and the model in Figure 4.5.

Table 4.1: Trace in the repairexample process.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Timestamp</th>
<th>Lifecycle Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>2014-10-01T14:30:00</td>
<td>complete</td>
</tr>
<tr>
<td>Analyze Defect</td>
<td>2014-10-01T14:31:00</td>
<td>start</td>
</tr>
<tr>
<td>Analyze Defect</td>
<td>2014-10-01T14:35:00</td>
<td>complete</td>
</tr>
<tr>
<td>Repair (Complex)</td>
<td>2014-10-01T14:37:00</td>
<td>start</td>
</tr>
<tr>
<td>Repair (Complex)</td>
<td>2014-10-01T14:52:00</td>
<td>complete</td>
</tr>
<tr>
<td>Inform User</td>
<td>2014-10-01T14:55:00</td>
<td>complete</td>
</tr>
<tr>
<td>Test Repair</td>
<td>2014-10-01T15:01:00</td>
<td>start</td>
</tr>
<tr>
<td>Test Repair</td>
<td>2014-10-01T15:12:00</td>
<td>complete</td>
</tr>
<tr>
<td>Archive Repair</td>
<td>2014-10-01T15:19:00</td>
<td>complete</td>
</tr>
</tbody>
</table>

Figure 4.5: Process tree discovered from the repairexample event log.
4.2.1 Determine Token Path

To determine how a token should flow through the process tree we construct a token path based on the alignment. We look at complete executions of the model (i.e., the synchronous and model moves in the alignment) for constructing the path. The token path consists of edges the token visits during the execution of the case and starts and ends at the root of the tree. Edges are added to the path as we walk through the alignment to see which activity is to be executed next. From the root a token flows over the edges and through operator nodes to the first activity (leaf) that is executed according to the alignment. From there it moves back up to the LCP of the last executed leaf and the to be executed leaf. As said earlier the LCP of two leaves indicate the order between them, therefore the token moves to this node to indicate which operator applies to both leaves. Then it moves down to the leaf to be executed. This process continues until the last activity is executed, from there the token flows back up to the root of the tree. When the path hits a parallel operator (AND or OR) the path splits, since two or more subprocesses are executed in parallel. Therefore, all branches of the parallel operator are treated as separate trees where each branch gets its own token and separate path. In case only one branch of OR is executed we treat it as a XOR.

Figure 4.6 shows the token path constructed for the trace in Table 4.1. The numbers along the edges indicate the order of the edges in the token path. We can only say something about the order of edges in each branch of the AND operator separately, therefore we use 1. and 2. to indicate the order in the two branches.

![Figure 4.6: Token path through a process tree.](image-url)

32 Static and Dynamic Visualization of Quality and Performance Dimensions on Process Trees
4.2.2 Construct Timed Path

Now that we have a path through the tree for each trace in the event log, for each trace we set timestamps on the source and target of the edges in the path to indicate the duration of the process. First, we use the timestamps of the leaf nodes, if they exist, to set timestamps on the edges which have a leaf node as source and/or target. When an edge has a leaf as target, the target timestamp is set to the start timestamp of the leaf. When an edge has a leaf as source, the source timestamp is set to the complete timestamp of the leaf. When source and target of an edge is a leaf, the source timestamp is set to the start timestamp of the leaf, and the target timestamp is set to the complete timestamp of the leaf. Finally, when for all edges these timestamps are set, the earliest and latest timestamps are determined (the first executed and last executed leaf, respectively). Then, the source timestamp of the first edge in the path is set to the earliest timestamp minus a constant, and the target timestamp of the last edge in the path is set to the latest timestamp plus a constant. This is done to start the process with a token flowing down from the root of the tree to the first executed leaf, and to end the process with a token flowing up from the last executed leaf to the root of the tree.

If we apply this to the token path constructed for the repair example the timestamps indicated in green in Table 4.2 are obtained. The numbers represent the edges in the token path presented in Figure 4.6.

To determine all source and target timestamps of the remaining edges in the path (all blue timestamps in Table 4.2) we use linear interpolation. For each trace we walk through the path and set the unknown timestamps as follows: if an edge has no source timestamp, we set it to the target timestamp of its preceding edge in the path. If an edge has no target timestamp, we get its source timestamp and the next edge with a timestamp in the path, nextEdge. The target timestamp of an edge \( e \) in the path then becomes:

\[
\text{TargetTimestamp}(e) = \begin{cases} 
\frac{\text{nextEdge.sourcetimestamp} - \text{e.sourcetimestamp}}{|\text{edges}| + 1} + \text{e.sourcetimestamp} & \text{if nextEdge has a source timestamp} \\
\frac{\text{nextEdge.targettimestamp} - \text{e.sourcetimestamp}}{|\text{edges}| + 2} + \text{e.sourcetimestamp} & \text{if nextEdge has a target timestamp}
\end{cases}
\]

where \( \text{edges} \) is the set of edges in between \( e \) and nextEdge.

The fraction determines the duration on each edge in the path from \( e.\text{source} \) to \( \text{nextEdge.target} \) or \( \text{nextEdge.source} \). In case \( \text{nextEdge} \) has a target timestamp we need to account for both \( e \) and \( \text{nextEdge} \) in the calculation of this duration (+2 in the denominator), since the token has to flow over all edges in between \( e \) and nextEdge, as well as these two edges within the time between \( e.\text{sourcetimestamp} \) and \( \text{nextEdge.targettimestamp} \). When \( \text{nextEdge} \) has a source timestamp we only need to account for \( e \) (+1 in the denominator), since the token has to flow over all edges in between \( e \) and \( \text{nextEdge} \), and \( e \) itself.

When an edge, \( e \), has a parallel operator, \( p \), as source and is directed upwards (parallel operator is finished executing), we ensure tokens from the different branches of \( p \) arrive at the same time at \( p \). To this end we take the maximum of all timestamps of edges directed upwards, which have \( p \) as target, and assign this timestamp as target timestamp to these edges, and as source timestamp to \( e \). In Table 4.2 this is done when the source timestamp of edge 31 (indicated in red) is determined. We take the maximum of the target timestamps of edges 1.30 and 2.12 (2014 – 10 – 01T15 : 15 : 00 and 2014 – 10 – 01T14 : 59 : 48 (indicated in blue), respectively), and set it as target timestamp of edge 2.12 (indicated in red), and as source timestamp of edge 31.
### Table 4.2: Timed path for the repair example process.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Source Timestamp</th>
<th>Target Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2014-10-01T14:29:30</td>
<td>2014-10-01T14:29:40</td>
</tr>
<tr>
<td>3</td>
<td>2014-10-01T14:29:50</td>
<td>2014-10-01T14:30:00</td>
</tr>
<tr>
<td>4</td>
<td>2014-10-01T14:30:00</td>
<td>2014-10-01T14:30:30</td>
</tr>
<tr>
<td>5</td>
<td>2014-10-01T14:30:30</td>
<td>2014-10-01T14:31:00</td>
</tr>
<tr>
<td>6</td>
<td>2014-10-01T14:31:00</td>
<td>2014-10-01T14:35:00</td>
</tr>
<tr>
<td>7</td>
<td>2014-10-01T14:35:00</td>
<td>2014-10-01T14:35:12</td>
</tr>
<tr>
<td>8</td>
<td>2014-10-01T14:35:12</td>
<td>2014-10-01T14:35:24</td>
</tr>
<tr>
<td>9</td>
<td>2014-10-01T14:35:24</td>
<td>2014-10-01T14:35:36</td>
</tr>
<tr>
<td>1.10</td>
<td>2014-10-01T14:35:36</td>
<td>2014-10-01T14:35:48</td>
</tr>
<tr>
<td>1.11</td>
<td>2014-10-01T14:35:48</td>
<td>2014-10-01T14:36:00</td>
</tr>
<tr>
<td>1.12</td>
<td>2014-10-01T14:36:00</td>
<td>2014-10-01T14:36:12</td>
</tr>
<tr>
<td>1.15</td>
<td>2014-10-01T14:36:36</td>
<td>2014-10-01T14:36:48</td>
</tr>
<tr>
<td>1.16</td>
<td>2014-10-01T14:36:48</td>
<td>2014-10-01T14:37:00</td>
</tr>
<tr>
<td>1.17</td>
<td>2014-10-01T14:37:00</td>
<td>2014-10-01T14:55:00</td>
</tr>
<tr>
<td>1.18</td>
<td>2014-10-01T14:55:00</td>
<td>2014-10-01T14:59:48</td>
</tr>
<tr>
<td>2.10</td>
<td>2014-10-01T14:35:36</td>
<td>2014-10-01T14:35:48</td>
</tr>
<tr>
<td>2.11</td>
<td>2014-10-01T14:45:18</td>
<td>2014-10-01T15:15:00</td>
</tr>
<tr>
<td>31</td>
<td>2014-10-01T15:15:00</td>
<td>2014-10-01T15:16:00</td>
</tr>
<tr>
<td>32</td>
<td>2014-10-01T15:15:00</td>
<td>2014-10-01T15:17:00</td>
</tr>
<tr>
<td>33</td>
<td>2014-10-01T15:15:00</td>
<td>2014-10-01T15:18:00</td>
</tr>
<tr>
<td>34</td>
<td>2014-10-01T15:15:00</td>
<td>2014-10-01T15:19:00</td>
</tr>
<tr>
<td>35</td>
<td>2014-10-01T15:15:00</td>
<td>2014-10-01T15:19:15</td>
</tr>
<tr>
<td>36</td>
<td>2014-10-01T15:15:00</td>
<td>2014-10-01T15:19:30</td>
</tr>
</tbody>
</table>

### 4.2.3 Animation

In order to animate the flow of tokens through the process tree, we implemented a global clock which starts at the earliest timestamp and ends at the latest timestamp over all timed paths. At each time step we select those timed paths for which it holds the current time is in its time interval, i.e., if current time is between the source timestamp of the first edge and the target timestamp of the last edge in the timed path. If this is the case, we select those edges in the timed path whose time interval (source timestamp to target timestamp) contains the current time. For such an edge $e$ we determine the fraction of time that has passed on $e$ in comparison to its time interval:
PassedTime(e) = \frac{\text{currentTime} - e\text{.sourcetimestamp}}{e\text{.targettimestamp} - e\text{.sourcetimestamp}}

This fraction is then used in the vector notion of the edge to compute the token’s position on the edge:

TokenPosition(e) = e\text{.source} + (e\text{.target} - e\text{.source}) \cdot PassedTime(e)

Finally, the token is drawn on this position on the edge and the current time is updated with a certain time step. Then, the process starts again. Each time the visualization is updated the amount of time units indicated by the time step is passed in the animation.

Figure A.6 shows the animation of tokens on the process tree of the repair example. We see three cases are executed, two of which are in a parallel execution (each case gets as many tokens as the number of branches of AND). Furthermore, we see “Analyze Defect”, “Repair (Complex)”, “Test Repair”, and “Inform User” are executed at this point in time in the animation (tokens are inside these leaf nodes).

Figure 4.7: Token animation on a process tree.
4.2.4 Resulting Framework

The final dynamic visualizations framework is depicted in Figure 4.8. The user can see the current date and time in the animation and the time step, which is the increase in time after each update of the visualization. By default this is set to one second. The user can pause the animation with the “Pause/Play” button, increase the time step and decrease the time step with the “Increase TimeStep” and “Decrease TimeStep” buttons, respectively. The time step is then increased/decreased with one second, with a minimum of one second. Furthermore, the user can zoom in and out on the animation with the “Scale Slider”.

![Figure 4.8: The dynamic visualizations framework.](image)

4.2.5 Discussion

This chapter described the resulting frameworks for the visualizations of metrics defined for process trees. We defined a static and a dynamic visualizations framework. In the static visualizations framework we can directly see for instance which parts of the model have low quality or contain bottlenecks. Furthermore, in the dynamic visualizations framework tokens are animated on the process tree, so we can follow the steps in the process and see whether all activities are executed as expected.

The next chapter evaluates whether these visualizations give a more detailed insight into the quality of the model and the performance of the process. This evaluation is conducted on the basis of synthetic and real-life logs.
Chapter 5

Evaluation

In this chapter, the static and dynamic visualizations defined in the previous chapter are evaluated using synthetic and real-life logs. We want to show the visualizations are easy to understand and indeed help organizations to better understand their processes.

In order to achieve this, process trees are discovered from event logs and the quality and performance measurements are projected on them. Then, we evaluate whether these visualizations indeed help to gain insight into (parts of) the process in terms of quality and performance.

5.1 Synthetic Logs

The synthetic logs and corresponding process trees have certain properties with respect to quality and performance. These properties have an effect on (parts of) the process trees, which are visible in the visualizations.

5.1.1 Quality_analysis_log

To evaluate the visualizations of the different quality metrics on the process tree we use the log in Table 5.1, from now on referred to as Quality_analysis_log, taken from [2]. The Quality_analysis_log consists of 1,391 cases, 7,539 events, 8 activities, and 21 unique paths through the process. The longest case consists of 17 events, the shortest one consists of 5 events. This event log is used to construct four different process trees, each having a different impact on the four quality dimensions; a good model, a most frequent trace model, a flower model, and a trace model. These are discussed in more detail in the remainder of this section. The results of the quality analysis are projected on the node color in each of the models. The legend in Figure 5.1 shows the range of values corresponding to each node color.

Figure 5.1: Legend used for the quality visualizations (see Section 4.1.2).

Good model

The process tree in Figure 5.2 is a good model with respect to several quality criteria. It can replay all of the traces observed in the log (fitness), it balances overfitting (generalization) and underfitting (precision), and is simple [2].

The projection of each quality metric on the nodes in the good model is shown in Figure 5.3. From these visualizations it is clear that the process tree and all its subtrees have high values for each of the quality dimensions, which is exactly what we expected to see.
CHAPTER 5. EVALUATION

Table 5.1: A simple event log.

<table>
<thead>
<tr>
<th>#</th>
<th>trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>455</td>
<td>ACDEH</td>
</tr>
<tr>
<td>191</td>
<td>ABDEG</td>
</tr>
<tr>
<td>177</td>
<td>ADCEH</td>
</tr>
<tr>
<td>144</td>
<td>ABDEH</td>
</tr>
<tr>
<td>111</td>
<td>ACDEG</td>
</tr>
<tr>
<td>82</td>
<td>ACDEG</td>
</tr>
<tr>
<td>56</td>
<td>ABDEH</td>
</tr>
<tr>
<td>47</td>
<td>ACDEFGDBEH</td>
</tr>
<tr>
<td>38</td>
<td>ABDEG</td>
</tr>
<tr>
<td>33</td>
<td>ACDEFBDEH</td>
</tr>
<tr>
<td>14</td>
<td>ACDEFGDBEG</td>
</tr>
<tr>
<td>11</td>
<td>ACDEFBDEG</td>
</tr>
<tr>
<td>9</td>
<td>ACCEFDEH</td>
</tr>
<tr>
<td>8</td>
<td>ACCEFDBEH</td>
</tr>
<tr>
<td>5</td>
<td>ACCEFDBDEG</td>
</tr>
<tr>
<td>3</td>
<td>ACDEFBDEFGDBEG</td>
</tr>
<tr>
<td>2</td>
<td>ACCEFDBDEG</td>
</tr>
<tr>
<td>2</td>
<td>ACCEFDBEDFBDEG</td>
</tr>
<tr>
<td>1</td>
<td>ACCEFDBEFGDBEHD</td>
</tr>
<tr>
<td>1</td>
<td>ACBEFDBEDFBDEG</td>
</tr>
<tr>
<td>1</td>
<td>ACCEFDBEDFDBEG</td>
</tr>
</tbody>
</table>

Figure 5.2: A good model for the exampleLog. It can replay all of the events observed in the log, it balances overfitting and underfitting, and is simple.
CHAPTER 5. EVALUATION

(a) Fitness results.
(b) Precision results.
(c) Generalization results.
(d) Simplicity results.

Figure 5.3: Visualizations of the quality analysis on the good model. It is clear that this process tree and all of its subtrees score good on all four quality criteria.
CHAPTER 5. EVALUATION

Most Frequent Trace Model

Figure 5.4 shows a process tree which models the most frequent trace in the event log ($\sigma_L = \langle A, C, D, E, H \rangle$). Therefore, fitness and generalization results for this model are expected to be low, since this model cannot replay all of the observed behavior and does not allow for any other behavior than the modeled trace.

Figure 5.4: Model for the most frequent trace in the exampleLog, i.e., other traces will not fit this model.

Figure 5.5 shows the resulting visualizations after projecting each of the quality metrics on the node color of the most frequent trace model. Using the legend in Figure 5.1, it can be derived the fitness of this process tree is between 0.6 and 0.7. Furthermore, we can see which activities have a lower fitness value indicating these parts of the model cannot replay all of the traces in the event log. In contrast to what we expected, generalization is high in this process tree. This might be due to the fact that fitness is poor and, therefore, cannot trust the results of other metrics.

Figure 5.5: Visualizations of the quality analysis on the most frequent trace model. Figure 5.5a shows this process tree’s fitness is low, meaning it cannot replay all traces observed in QualityAnalysisLog.

Flower Model

The model in Figure 5.6 is a variant of the “flower model”; only the start and end activities are modeled right. As a result, precision is low, since the model allows for much other behavior than observed in the log.

The visualizations of each of the quality metrics on the flower model are presented in Figure 5.7. From Figure 5.7b it is clear that the precision of this model is low, as we expected. Furthermore, we see this is caused by the operators labeled XOR and LOOPXOR, because this part of the model allows for much other behavior than seen in the event log.
CHAPTER 5. EVALUATION

Figure 5.6: Flower model for the exampleLog.

(a) Fitness results.

(b) Precision results.

(c) Generalization results.

(d) Simplicity results.

Figure 5.7: Visualizations of the quality analysis on the flower model. Figure 5.7b clearly shows precision is low for this process tree.

Trace Model

Figure 5.8 depicts part of a process tree which simply enumerates the 21 different traces seen in the event log, a so-called trace model. For this model, simplicity and generalization are low, because there are many duplicate leaves and the model allows for no other behavior than seen in the log.

Figure 5.8: Part of the trace-model for the exampleLog.

Figure 5.9 shows the visualizations of the fitness and precision metrics on the trace model. As expected, these are high for all nodes in this process tree, because it can replay all observed behavior and only allows for this behavior. In Figure 5.10 generalization is visualized on the trace model.
CHAPTER 5. EVALUATION

Generalization is highly influenced by frequency, therefore, we see a decrease in generalization in each modeled trace (each subtree with operator node labeled seq as root). The most frequent traces have a high value for generalization, less frequent traces have a low value. Generalization of the whole model is low, as expected. Finally, Figure 5.11 shows the simplicity visualized on the trace model. Here, we see that the parts of the model that contain duplicate activities have a lower simplicity. Since the model contains a lot of duplicate activities, the simplicity of the model is very low, which is what we expected.

Figure 5.9: Visualizations of the fitness and precision results on the trace model. This process tree scores well on both fitness and precision.

Figure 5.10: Visualizations of generalization on the trace model. As can be seen, generalization of this process tree is low, since
Figure 5.11: Visualizations of simplicity on the trace model. Simplicity of this process tree is really low, since there are a lot of leaves having the same label, i.e., duplicate nodes.

5.1.2 Telecom_service_log

To evaluate the visualizations of the different performance metrics we modeled a fictitious process in CPN Tools and generated an event log out of this process [15], which contains 1,000 cases, 9,950 events, 13 activities, and 108 unique paths. The longest case consists of 18 events, the shortest one consists of 7 events. The process describes what happens inside a telecom service provider, after a customer has placed an order. It consists of three main paths: the first path concerns ordering the mobile phone on behalf of the customer, the second path concerns activation of the customer’s subscription on the mobile network, and the third path concerns the optional transferring of an existing mobile telephone number to the new subscription. The process tree that models this process was manually constructed and is depicted in Figure 5.12.

The activities in this process have a high variation in execution and waiting times, and therefore, also in sojourn times. In Table 5.2 these times are listed for each activity in the process, times between brackets indicate an interval (e.g., [5,10] means the time varies between 5 and 10 minutes). All activities have a corresponding event in the log with a “complete” timestamp, but not all activities have a corresponding event in the log with a “start” timestamp. Therefore, we cannot compute execution and waiting times for these activities.

First, we look at the quality of this process tree to show there are no conformance issues between model and event log. Especially fitness should be high to provide meaningful performance results, since fitness indicates the degree to which the process model is able to replay the events observed in the log. If an event could not be mapped to a leaf in the process tree, the additional information (such as timestamp and lifecycle transition) recorded for this event is lost, resulting in unreliable performance statistics.
Figure 5.12: Process tree for the telecom service provider process

Table 5.2: Execution, waiting, and sojourn times for the activities in the telecom service process.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Execution time (minutes)</th>
<th>Waiting time (minutes)</th>
<th>Sojourn time (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive Order</td>
<td>Unknown</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
<tr>
<td>Order Phone</td>
<td>[10,20]</td>
<td>[0.5]</td>
<td>[10,25]</td>
</tr>
<tr>
<td>Receive Out-of-stock</td>
<td>Unknown</td>
<td>Unknown</td>
<td>[25,50]</td>
</tr>
<tr>
<td>Timeout</td>
<td>Unknown</td>
<td>Unknown</td>
<td>25</td>
</tr>
<tr>
<td>Discuss With Customer</td>
<td>[5,10]</td>
<td>0</td>
<td>[5,10]</td>
</tr>
<tr>
<td>Receive Phone</td>
<td>Unknown</td>
<td>Unknown</td>
<td>50</td>
</tr>
<tr>
<td>Activate Subscription</td>
<td>[10,20]</td>
<td>[1.5]</td>
<td>[11,25]</td>
</tr>
<tr>
<td>Send Package</td>
<td>[5,15]</td>
<td>0</td>
<td>[5,15]</td>
</tr>
<tr>
<td>Request Number Transfer</td>
<td>[5,10]</td>
<td>[1.5]</td>
<td>[6,15]</td>
</tr>
<tr>
<td>Receive Confirmation</td>
<td>Unknown</td>
<td>Unknown</td>
<td>[1.5]</td>
</tr>
<tr>
<td>Activate Number Transfer</td>
<td>[20,40]</td>
<td>[0.18]</td>
<td>[20,58]</td>
</tr>
<tr>
<td>Skip Number Transfer</td>
<td>Unknown</td>
<td>Unknown</td>
<td>[1.5]</td>
</tr>
<tr>
<td>Archive</td>
<td>Unknown</td>
<td>Unknown</td>
<td>[10,25]</td>
</tr>
</tbody>
</table>
(a) Fitness results. 
(b) Precision results. 
(c) Generalization results. 
(d) Simplicity results.

Figure 5.13: Visualizations of the quality analysis on the telecom service process tree. It scores well on all quality criteria.

Figure 5.13 shows the process tree scores well on all quality criteria. Each quality criteria is projected on node color and edge label. The process tree has perfect fitness, so all events in the event log could be mapped to leaves in the process tree. Therefore, we can extract all additional information recorded in the event log and perform a reliable performance analysis.
CHAPTER 5. EVALUATION

Now that we have shown there are no conformance issues, we can accurately measure performance of the telecom service process. The results are projected on node color and edge label in each of the visualizations. In Figure 5.14 the average execution times are depicted. The activity “Activate Number Transfer” is the task with the highest execution time, which corresponds with the information in Table 5.2. Furthermore, we are now able to see execution times of the different subprocesses. For example, the subprocess (subtree of operator node labeled LOOPDEF) which starts with ordering the phone and ends with receiving the phone seems to be a bottleneck for this process, since it might be the case the phone is not in stock or a timeout occurs, after which the phone needs to be ordered again.

Figure 5.14: Average execution time. “Activate Number Transfer” is the activity that takes the longest. Furthermore, the loop seems to be a bottleneck for this process.
CHAPTER 5. EVALUATION

Figure 5.15 depicts the average waiting times of the different activities and subprocesses. We see, waiting times are very short on average, maximal 4 minutes. Furthermore, we see that on average “Activate Subscription” has a shorter waiting time, than the subprocess of ordering and receiving the phone (operator node labeled \texttt{ LOOPDEF}), which means on average “Activate Subscription” is started before “Order Phone” is started for the first time. We cannot draw this conclusion from looking at activities only, because “Order Phone” has a waiting time of 1 minute on average, because this waiting time depends on when “Receive Order” is completed (when “Order Phone” is executed for the first time, “Receive Order” is its LaCoSeL), and when “Discuss With Customer” is completed (when “Order Phone” is executed more than once, “Discuss With Customer” is its LaCoSeL). However, waiting time of the subprocess modeled by the \texttt{ LOOPDEF} operator node always depends on when “Discuss With Customer” is completed, just as is the case for “Activate Subscription”. Therefore, we can only compare these two nodes, which is only possible, because we can perform performance analysis on subprocess as well.

Figure 5.15: Average waiting time. All waiting times are relatively low compared to the execution times (see Figure 5.14). “Activate Number Transfer” has the highest waiting time.
In Figure 5.16 the sojourn times (i.e., waiting time + execution time) are depicted. We roughly see the same colors as in Figure 5.14, because the execution times have a larger impact on the sojourn time than the waiting times in Figure 5.15. Furthermore, we can compute sojourn times of almost all activities, since each activity has “complete” timestamps and a LaCoSeL, except for “Receive Order”. Therefore, no average sojourn time could be calculated for this activity and the root.

The process spends most time in activity “Receive Phone”, 50 minutes on average. This is the time between the last completion of “Order Phone” and the completion of “Receive Phone” in the process. This further supports the earlier observation that the subprocess which starts with ordering the phone and receiving the phone is the bottleneck of this process. So, to improve efficiency of the total process, one should look into this part of the process.

Figure 5.16: Average sojourn time. Again, the loop seems to be a bottleneck for this process.
Figure 5.17 shows the synchronization times of the two operator nodes labeled AND. We see that, on average, the lowest parallel operator has a slightly lower synchronization time than the highest parallel operator. Furthermore, synchronization times are quite long, 86 and 87 minutes respectively. For the lowest parallel operator this is due to the fact that one of the two parallel subprocesses consists of only one activity (“Activate Subscription”), which is most likely to finish much earlier than the other subprocess. For the highest parallel operator this is the case when the right parallel subprocess is skipped (“Skip Number Transfer”). Even when the number transfer subprocess (sequence of “Request Number Transfer”, “Receive Confirmation”, and “Activate Number Transfer”) in the right parallel process is executed, it is most likely that it finishes before the left parallel subprocess, but then the synchronization time is probably lower.

Figure 5.17: Average synchronization time for the two parallel operators.
Finally, Figure 5.18 shows a screenshot of the token animation on this process tree. A lot of tokens are held up in “Order Phone”, “Activate Subscription”, and “Activate Number Transfer”, as these are the bottlenecks of this process. Furthermore, the red-colored tokens are all waiting for the tokens in blue to be finished, representing the synchronization time of the highest parallel operator.

Figure 5.18: Token animation on the telecom service process. Most of the tokens are held up in the loop part of the process (indicated in blue). The tokens in red are waiting for this part to be completed.
CHAPTER 5. EVALUATION

5.1.3 Discussion

In this section we have shown that with a more detailed insight into the process, we can for instance detect parts of the model which have low quality, or are a bottleneck to the entire process. These kind of observations are all possible because of the hierarchical structure of process trees, which allows us to perform quality and performance analysis on subtrees. Subsequently, the visualizations of the result of these analyses immediately draws attention to the parts of the model which have different behavior than the rest of the model. Therefore, our preliminary conclusions are:

- Fitness needs to be high in order to provide reliable performance analysis.
- The visualizations give a detailed insight into the process and are easy to understand.

In the next section we perform quality and performance analysis on a process tree discovered from a real-life event log in order to show these visualizations are also applicable to real-life situations.

5.2 Real-life Log

Next, we use an event log taken from a Dutch financial institution. The process consists of three large subprocesses: one for loan applications, one for loan offers, and one for work items. Therefore, we split the event log in three sublogs, each containing activities of only one of the subprocesses, and discover a process from each of these event logs with the Inductive Miner [7].

This log was used in the BPI Challenge of 2012 (BPIC12) where it was analyzed in terms of performance; discovering bottlenecks, resource usage, etc. [16–20]. Currently, the time metrics are the only performance related metrics defined for process trees. Therefore, we investigate whether our approach is able to discover the bottlenecks of the three subprocesses, using the time metrics defined in Chapter 3 and the visualizations developed in Chapter 4. The bottlenecks found in the BPIC12 reports are all on activity level, since no analysis is performed on parts of the model. With this in mind, we measure the performance of each of the subprocesses and project the results on the corresponding process tree. We analyze these visualizations to find bottlenecks in each of the subprocesses, and compare our findings to the ones found in the BPIC12. We first verify that our visualizations can achieve the same results and second, can give even more insight into the process.

5.2.1 Loan Application Subprocess

This sublog consists of 13,087 cases, 60,849 events, 10 activities, and 17 unique paths. Case lengths vary from 3 to 8 events. In Table 5.3 the durations of the bottleneck activities extracted from the BPIC2012 reports in the loan application subprocess are listed. Of this subprocess only the completion times of the activities are recorded, therefore, we can only determine sojourn times of the activities and subprocesses. The process tree discovered from this sublog with the Inductive Miner (Variant: Inductive Miner - infrequent, Noise threshold: 0.50) is depicted in Figure 5.19.

Table 5.3: Bottlenecks in the loan application subprocess.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Sojourn time (days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_CANCELLED</td>
<td>17.99</td>
</tr>
<tr>
<td>A_APPROVED</td>
<td>16.00</td>
</tr>
<tr>
<td>AREGISTERED</td>
<td>16.00</td>
</tr>
<tr>
<td>A_ACTIVATED</td>
<td>16.00</td>
</tr>
<tr>
<td>A_DECLINED</td>
<td>1.92</td>
</tr>
</tbody>
</table>

In Figure 5.20 the results of the quality analysis on this process tree are shown. This process tree scores well on all quality criteria. Only precision is relatively low for the second and third operator nodes labeled XOR. This might be due to the fact that this model allows all possible
CHAPTER 5. EVALUATION

Figure 5.19: Process tree for the loan application subprocess.

Figure 5.20: Visualizations of the quality analysis on the loan application subprocess. This process tree scores well on all quality criteria.

Now that we know there are no conformance issues, we can measure performance of this process. Figure 5.21 shows the results of the performance analysis on this process tree. The average sojourn time (in minutes) is projected on node color and edge label. It is clear that “A_CANCELED” is the activity that takes the longest, 25,899 minutes (17.99 days) on average. Furthermore, “A_REGISTERED”, “A_APPROVED”, and “A_ACTIVATED” each take 23,036 minutes (16.00 days) on average. Finally, “A_DECLINED” takes 2,767 minutes (1.92 days) on average. These correspond to the results obtained from [16]. Besides from these observations, the
rightmost operator node labeled XOR is the subprocess that takes longest compared with all other nodes on the same level.

In Figure 5.22 the average synchronization time of the parallel operator node is projected. Because this time is equal to 0 minutes, we can conclude that the three activities are each completed on exactly the same time (in minutes). After inspecting the event log, it is indeed the case that the three activities executed in parallel have identical timestamps (only differ in milliseconds).

Figure 5.22: Average synchronization time projected on node color and edge label. Since synchronization time is 0, the three parallel activities are all completed at the same time.

Figure 5.23: Token animation on the loan application subprocess. Tokens are either on their way or held up in the bottlenecks of the process.

Figure 5.24: Frequency projected on node color and edge label. We can conclude that loan applications are often declined.

As shown in Figure 5.23, most of tokens are either on their way to the rightmost part of the process tree, or are in the rightmost subtree. This further supports the conclusion that this part of the process is a bottleneck.
Finally, Figure 5.24 depicts the frequency of each node in the process recorded in the log. We can conclude from this visualization that loan applications are often declined, and the activities “A_ACCEPTED” and “A_FINALIZED” are often skipped.

### 5.2.2 Loan Offer Subprocess

This sublog consists of 5,015 cases, 31,244 events, 7 activities, and 168 unique paths. Case lengths vary from 3 to 8 events. In Table 5.4 the durations of the bottleneck activities extracted from [16] in the loan offer subprocess are listed. Also for this subprocess only the completion times of the activities are recorded, so we can only determine sojourn times of the activities and subprocesses. The process tree discovered from this sublog with the Inductive Miner (Variant: Inductive Miner - infrequent, Noise threshold: 0.50) was slightly adjusted. Since the discovered process tree started with a loop construct, there was no place without an incoming arc in the Petri net converted from this tree (i.e., a source). Such a place is needed for constructing the alignment between process tree and event log. Therefore, we moved the leaf “O_SELECTED”, which was the leftmost child of the operator node labeled seq in the “body” of the loop, out of the loop and set it as leftmost child of the root. The resulting process tree is depicted in Figure 5.25. From the results of the

<table>
<thead>
<tr>
<th>Activity</th>
<th>Average Duration (days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_CANCELLED</td>
<td>11.85</td>
</tr>
<tr>
<td>O_SENT_BACK</td>
<td>9.58</td>
</tr>
<tr>
<td>O_ACCEPTED</td>
<td>4.05</td>
</tr>
<tr>
<td>O_DECLINED</td>
<td>4.02</td>
</tr>
</tbody>
</table>

quality analysis, depicted in Figure 5.26, we can conclude the process tree for this subprocess has a high quality. It could be simpler, since we see in Figure 5.26d that the rightmost child of the loop (activity labeled “tau”, representing \( \tau \)) is useless, since it can be replaced by the subtree next to it, which has the operator node labeled XOR as root. Furthermore, precision of the two operator nodes labeled XOR is relatively low, since this model allows for more combinations of activities part of these operator nodes than present in the event log.
CHAPTER 5. EVALUATION

(a) Fitness results.

(b) Precision results.

(c) Generalization results.

(d) Simplicity results.

Figure 5.26: Visualizations of the quality analysis on the loan offer subprocess. This process tree scores well on all four quality criteria, although it could be simpler.

Figure 5.27 shows the results of the performance analysis on this process tree. The average sojourn time (in minutes) is projected on node color and edge label. It is clear that “O_CANCELED” is the activity that takes the longest time, 17,529 minutes (about 12 days) on average. Furthermore, “O_SENT_BACK” takes 13,788 minutes (about 9.5 days) on average. Finally, “O_ACCEPTED” and “O_DECLINED” take 5,831 and 5,783 minutes respectively, which is about 4 days. These correspond to the results obtained from the BPIC12. Besides from these observations, we can conclude that the loop and the operator node labeled xor next to it are bottlenecks of this process. They take about 12,300 (8.5 days) and 13,746 minutes (9.5 days) respectively. So, improving the efficiency of these two subprocesses could have a positive effect on the overall performance.

Figure 5.27: Average sojourn time projected on node color and edge label. “O_CANCELED” takes the longest, the loop and the operator node labeled xor next to it seem to be the bottlenecks for this process.

When we look at the token animation (see Figure 5.28), most tokens are held up in the loop part of the process, which substantiates our previous conclusion, that this subprocess is a bottleneck of the process, even further.
Finally, Figure 5.29 shows the frequency projected on the process tree for the loan offer process. It is clear that the “body” of the loop is executed the most in this subprocess, which is intuitively logical, since only in a loop activities can be executed more than once in a trace. Furthermore, “O,CANCELLED” is executed about as many times as it is skipped.

5.2.3 Work Items Subprocess

This sublog consists of 9,658 cases, 170,107 events, 19 activities, and 2,921 unique paths. Case lengths vary from 3 to 8 events. The work items subprocess contains only one bottleneck as depicted in Table 5.5. In this subprocess the times on which an activity is scheduled for execution (lifecycle:transition = “schedule”), and when the activity is started and completed are recorded in the event log. This means we can calculate waiting, execution, and sojourn times of the activities. Furthermore, we can expect log moves for each event which has a lifecycle:attribute with value “schedule”, since we cannot deal with this lifecycle transition at the moment. As a result, fitness of this process tree might be low. As the Inductive Miner cannot mine or constructs, and we know from the results of the BPIC12, either “W,Completeren aanvraag”, “W, Afhandelen leads”,

Figure 5.28: Token animation on the loan offer subprocess. Most tokens are held up in the loop construct, which means this seems to be a bottleneck of this process.

Figure 5.29: Frequency projected on node color and edge label. It is clear that the “body” of the loop is executed the most in this subprocess.
or both are executed in the leftmost part of the process tree, we manually changed a discovered AND construct into an OR construct (see Figure 5.30).

Table 5.5: Bottlenecks in the work items subprocess.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Average Duration (days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>W_Nabellen offertes</td>
<td>3.07</td>
</tr>
</tbody>
</table>

Figure 5.30: Process trees for the work items subprocess.

(a) Fitness results.  
(b) Precision results.  
(c) Generalization results.  
(d) Simplicity results.

Figure 5.31: Visualizations of the quality analysis on the work items subprocess. Fitness is very low (0.15), which means the other metrics become less reliable.
Figure 5.31 depicts the results of the quality analysis. What is most striking is the low value of fitness for the entire process tree (0.15), which means a lot of traces cannot be replayed by this model. As said before, this impacts the results of the other metrics, since we cannot use all information recorded in the event log. Therefore, performance measurements might be inaccurate.

However, we still have measured the performance of this subprocess, since fitness of the subtrees is quite high, which means that a reasonable amount of events can be mapped on the leaves in these subtrees. In Figure 5.32 the results of the performance analysis are depicted. As we can see in Figure 5.32a, the execution times of the individual tasks are relatively low, maximum 33 minutes on average. Furthermore, we can see that the node operator labeled seq has the longest execution time (544 minutes, about 9 hours). Average waiting time could only be calculated for “W_Nabellen incomplete dossiers” and the lowest operator node labeled xor (see Figure 5.32b), because we need a LaCoSeL to be able to compute waiting times. The other branches of the highest operator node labeled xor consist of only a single activity, or parallel activities, so these have no LaCoScLs in the process tree. As a result also the average sojourn time can only be calculated for these two nodes. Unfortunately, we cannot show the average synchronization time for the or operator node, since this has not been implemented yet.

Figure 5.32: Visualizations of the performance analysis on the work items subprocess.

As we expected from the quality analysis on this process tree, our performance analysis results are not reliable. In contrast to the results in the BPIC12, our visualizations show a bottleneck in the subtree with operator node labeled seq.
5.2.4 Discussion

As we have seen in this section, our approach is applicable to real-life processes and can achieve the same results as the approaches used in the BPIC12 as regards discovering bottlenecks. A condition is that the process tree should be of high quality in order to perform a reliable performance analysis. However, this is a requirement for most performance analysis techniques. Furthermore, we have shown our visualizations are easy to understand and give a more detailed insight into the process, because we can also perform quality and performance analysis on parts of the model. For instance, we discovered the parallel subprocess is a bottleneck for the loan application subprocess, and the loop is a bottleneck for the loan offer subprocess.
Chapter 6

Conclusions and Future Work

For organizations it is vital to obtain detailed and reliable insights into the way processes are actually executed. In this thesis we aimed to visualize quality and performance of processes in novel ways. Therefore, we used process trees, which are process models in which the process is represented by a tree, and have a hierarchical structure. As a result, quality and performance analyses can be performed on parts of the model as well. Moreover, quality and performance metrics applicable to (parts of) process trees, and visualizations to project the results of these metrics on a process tree were needed to be developed. Therefore, the project had two main goals: (1) develop quality and performance metrics applicable to process trees, and (2) come up with visualizations that project these metrics on elements of the process tree. These visualizations highlight quality of (parts of) the model with respect to the log and performance of the activities and (sub-)processes.

In order to reach the first goal, we analyzed existing quality and performance metrics and used some concepts in the development of metrics applicable to (parts of) process trees. Additionally, we constructed an alignment between process tree and event log to be able to link events in the log to nodes in the process tree, and to extract information stored in the attributes of events in the log. With the alignment and the developed metrics we can compute quality and performance statistics on (parts of) process trees. For the second goal, we designed a static and a dynamic visualizations framework. The former projects quality and performance statistics on process tree elements. The latter shows an animation of steps executed in the process. In this animation tokens flow through the process tree, each representing a case being handled in the process. All work can be found in the ProcessTreeReplayer package. Appendix A provides a manual on how to use the developed plugins for ProM can be found.

Finally, the visualizations were evaluated to show they give a detailed insight into the process. Evaluation was conducted on the basis of two synthetic logs and one real-life log. The synthetic logs and corresponding process trees were used to show the possibilities of the visualizations. The real-life log was used to show our approach is applicable to real-life processes and achieves the same results as other, existing approaches, and even gives a more detailed insight into the process than the existing approaches.

We have demonstrated that the visualizations give a detailed insight into the process: low quality in parts of the model and bottlenecks of the process on subprocess level were detected, which allows companies and research groups to improve the quality of the model, and performance of the entire process by addressing just a small part of it. A condition is that the discovered process tree should be of high fitness, because otherwise the results of the performance analysis might be unreliable. After all, poorly fitting models cannot replay all cases recorded in the event log, so not all of the information can be linked to process tree nodes. As a result, this information is lost, which for instance causes unreliable results in the calculation of execution times. However, this is a limitation of most performance analyses techniques.

---

1See: https://svn.win.tue.nl/repos/prom/Releases/Packages/ProcessTreeReplayer/
CHAPTER 6. CONCLUSIONS AND FUTURE WORK

To conclude, our approach offers novel ways to analyze processes and process models. Until now, analysis techniques allowed us to reason about the entire process (high level), or individual activities (low level) only. For example, Business Intelligence (BI) tools [21] typically provide a dashboard of the process in which information is visualized about individual activities and the entire process, and Online Analytical Processing (OLAP) [22] provides multidimensional data structures to visualize information such as profit, costs, and resource usage per category. Our approach can be used to perform analyses on subprocess level as well, which gives companies a more detailed insight into their process.

6.1 Future Work

Of course, the ProcessTreeReplayer package could be extended with other metrics which are applicable to (parts of) process trees, e.g., one could calculate resource usage and costs on activity, subprocess, and process level. Furthermore, existing metrics could be replaced by other metrics that are more accurate or are more efficient in terms of time complexity. The current generalization metric, for instance, is too dependent on frequency, as a consequence, the results of this metric seem too high. Secondly, currently only the forward-based alignment automaton in the computation of the precision metric is computed, and this might introduce a bias in the result, when there exist long-distance dependencies between activities. Therefore, if we can reduce the time complexity of computing both forward- and backward-based alignment automata, we can take the average of both results and remove this bias in the precision results. Thirdly, the calculation of synchronization time for operator nodes labeled or has not been implemented yet, due to time constraints. Besides aforementioned extensions, visualizations could also be extended. For instance, by normalizing the results of the quality metrics, we could even see small differences between the results, when all values are high for a certain quality metric.

Our evaluation showed the tool’s possibilities and that it is applicable to real-life situations: it achieves the same results as other ones, and, moreover, gives a more detailed insight into the process. However, before the ProcessTreeReplayer package can be used in practice, it should be evaluated on user-friendliness. One could conduct a survey among the future users of the tool to measure their experiences with it. Then, the results give an indication of the difficulty to perform certain actions in the tool. Based on these findings, one could decide to make the tool more user-friendly.

Finally, our tool could be made more interactive as we see in OLAP, where data is visualized on the fly dependent on the user’s input. Furthermore, Wave, a visualization platform made by Marijn Grootjans at the TU/e [23], visualizes a soccer match in a so called “Wave diagram”, which shows the progress of the match at-a-glance. A process could also be viewed as a soccer match in which certain events occur at a certain time, and visualize these events in a similar way, such that companies get insight into duration, costs, and resource usage in one visualization.
Bibliography


Appendix A

ProM Plugin Manual

This appendix chapter describes how the different plugins in the ProcessTreeReplayer package should be used.

A.1 Replay a Log on a Process Tree for Quality and Performance Analysis Plugin

Input:
An event log and a process tree mined from this event log.

Output:
An annotated process tree.

A.1.1 General information

An alignment between event log and process tree is constructed and, subsequently, quality and performance is measured. Furthermore, a path through the process tree is constructed on the basis of the alignment to determine how tokens should flow through the tree. Timestamps along the path indicate the speed of a token when it moves from one node to another or inside a node. The result is a process tree, which contains the results of the quality and performance analyses for each node and edge in the process tree, and token paths for each trace in the event log. This annotated process tree is visualized using one of the available visualizer plugins for process trees. In order to visualize the annotated process tree in one of the developed frameworks, one must select one of the following visualizers:

Visualize Process Tree Statistics
This visualizer plugin shows the static visualizations framework. The annotated process tree is displayed in the “Drawing Pane” and the user can select which metric is to be visualized on which process tree element.

Visualize Process Tree Animation
This visualizer plugin shows the dynamic visualizations framework. The annotated process tree is displayed in the “Drawing Pane” and the animation is started immediately. The user can pause the animation and increase and decrease the timestep.

Please note that both visualizer plugins can only be used after the Replay a Log on a Process Tree for Quality and Performance Analysis plugin has been executed.
A.1.2 Manual

The usage of the plugin is described on the basis of the repairexample event log and a process tree discovered from this log with the Inductive Miner (Variant: Inductive Miner - infrequent, Noise threshold: 0.50). We assume the event log and the process tree discovered from it are stored on your system.

First, ProM needs to be started, then, in the workspace (see Figure A.1), click on “Import”, browse through the folders, select the event log, and click on “Open”. Then click on “Import” again, and select the process tree discovered from this log, and click on “Open”. Now your workspace should look like the one in Figure A.2.
Figure A.2: The event log and the process tree are imported in ProM.

Next, select the process tree resource in your workspace and click on the “play” button (highlighted in red in Figure A.2). Then, click on “Click to add input object”, select the event log resource from the list, and click on “Select”. Finally, select “Replay a Log on a Process Tree for Quality and Performance Analysis” in the list of plugins (see Figure A.3), and click on “Start”.

Figure A.3: The “Replay a Log on a Process Tree for Quality and Performance Analysis” plugin is selected.
The plugin is now running (see Figure A.4). When the plugin is finished, the process tree is visualized like in Figure A.5. In this screen click on “Create new...” and select either “Visualize Process Tree Animation” or “Visualize Process Tree Statistics”.

Figure A.5: Choose a visualizer from the dropdown menu.
When the former is selected, your screen should look like the one in Figure A.6, and the animation is started immediately. When the latter is selected, your screen should look like the one in Figure A.7. Initially, no visualizations are selected. By using the dropdown menus at the bottom right, one can select which metric is to be visualized on which process tree element. Furthermore, by using the dropdown menus at the bottom left, one can select which color map is to be used on nodes, borders, or edges. An example of a visualization is shown in Figure A.8.

Figure A.6: Visualization when “Visualize Process Tree Animation” is selected.

Figure A.7: Visualization when “Visualize Process Tree Statistics” is selected.
Figure A.8: An example of a visualization in the Static Visualization Framework. Fitness is projected on node label, average sojourn time on node size and color, useless node is projected on border width and color, and frequency is projected on edge label, size, and color.