

# Hierarchical process mining for scalable software analysis

**Citation for published version (APA):**

Leemans, M. (2018). *Hierarchical process mining for scalable software analysis*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science]. Technische Universiteit Eindhoven.

**Document status and date:**

Published: 06/12/2018

**Document Version:**

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

**Please check the document version of this publication:**

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

**General rights**

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

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

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

[www.tue.nl/taverne](http://www.tue.nl/taverne)

**Take down policy**

If you believe that this document breaches copyright please contact us at:

[openaccess@tue.nl](mailto:openaccess@tue.nl)

providing details and we will investigate your claim.

Hierarchical  
Process Mining  
for Scalable  
Software Analysis

Maikel Leemans

Copyright © 2018 by Maikel Leemans. All Rights Reserved.

Maikel Leemans

Hierarchical Process Mining for Scalable Software Analysis by Maikel Leemans  
Eindhoven: Eindhoven University of Technology, 2018. Thesis.

A catalogue record is available from the Eindhoven University of Technology  
Library.

ISBN 978-90-386-4618-3

Keywords: Process Mining, Reverse Engineering, Process Discovery, Performance Analysis, Software Analysis, Event Logs, Hierarchical Modeling, Hierarchical Discovery, Hierarchical Performance Analysis, Hierarchical Event Logs, Cancellation Modeling, Cancellation Discovery, Software Process Analysis Methodology, Statechart Workbench, Software Analysis Workbench



SIKS Dissertation Series No. 2018-27

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Printed by Ipskamp Printing

Cover design by Harald Pieper, In Zicht Grafisch Ontwerp

# Hierarchical Process Mining for Scalable Software Analysis

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische  
Universiteit Eindhoven, op gezag van de rector magnificus  
prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen  
door het College voor Promoties, in het openbaar te  
verdedigen op donderdag 6 december om 11:00 uur

door

Maikel Leemans

geboren te 's-Hertogenbosch

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter: prof.dr. Mark A. Peletier  
1e promotor: prof.dr.ir. Wil M. P. van der Aalst  
2e promotor: prof.dr. Mark G. J. van den Brand  
externe leden: prof.dr. Arie van Deursen (Delft University of Technology)  
prof. Serge Demeyer (Universiteit Antwerpen)  
prof. Alistair Barros (Queensland University of Technology)  
lid TU/e: prof.dr.ir. Jarke J. van Wijk

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

*In liefdevolle nagedachtenis aan mijn moeder (1959 – 2016)*

*In loving memory of my mother (1959 – 2016)*



# Abstract

In today's world, we increasingly rely on complex software-driven systems. Thus it comes as no surprise that software-related problems can have an incredible impact on society, organizations, and users. In practice, it is hard to address these problems *a priori* because software continues to evolve and grow in complexity and operates in an ever-changing environment. To address and possibly avoid these problems, one has to observe, log, monitor and study software systems "*on the run*", preferably in its natural, real-life production environment. *Process mining* techniques use logged event data, obtained from operational and software processes, to *discover* process models, to *check the conformance* of predefined process models, and to *extend* such models with information about bottlenecks, decisions, deviations, resource usage, and more.

In this thesis, we will show how we can use process mining for analyzing software systems. In addition, we will address the lack of support for *hierarchical subprocesses*, *recursive behavior*, and *cancelation behavior*, which is commonly found in software behavior. By including these constructs, we benefit from the extra information such constructs provide and improve the *algorithmic* and *visual scalability*. Finally, we will address the lack of support for *integrating* process mining in the software process analysis lifecycle.

To summarize, this thesis contains the following contributions:

- A detailed discussion of software event data, how to use such data for process mining, as well as tool support for logging such data.
- A modeling notation and discovery techniques for hierarchical and recursive behavior, exploiting hierarchical models with named submodels.
- A modeling notation and discovery techniques for cancelation behavior.
- A framework for visualization-independent performance analysis, supporting hierarchy, recursion, and cancelation.
- A family of model translations, supporting hierarchy, recursion, and cancelation, thereby separating visualization and representation bias.
- Extensive tool support for round-trip software analysis.

All methods have been implemented, systematically evaluated, and applied in real-life situations in the context of several case studies.





# Table of contents

<b>Abstract</b>	<b>vii</b>
<b>Table of contents</b>	<b>ix</b>

## I Introduction

<b>1</b>	<b>Overview</b>	<b>3</b>
1.1	Process Mining	4
1.1.1	Event logs	5
1.1.2	Process Discovery	6
1.1.3	Performance and Conformance Analysis	9
1.1.4	Enhancement	10
1.2	Software Engineering and Analysis	11
1.2.1	Software Analysis and Model Learning	11
1.2.2	Process Mining for Software Analysis	12
1.3	Open Challenges in Process Mining for Analyzing Software	14
1.4	Our Approach – Design Decisions and Foundation	16
1.5	Contributions and Structure of this Thesis	17
<b>2</b>	<b>Preliminaries</b>	<b>21</b>
2.1	Basic Notation	21
2.1.1	Sets, Sequences, and Functions	21
2.1.2	Graph theory	23
2.2	Process Model Notations	24
2.2.1	Petri nets	25
2.2.2	YAWL – Yet Another Workflow Language	33
2.2.3	BPMN – Business Process Modeling Notation	34
2.2.4	Statecharts	35
2.2.5	MSD – Message Sequence Diagrams	36
2.2.6	Process Trees	38

2.3	Event Logs	43
2.3.1	Structure of an Event Log	43
2.3.2	XES – Extensible Event Stream	46
2.3.3	Atomic Event Log	47
2.3.4	Non-Atomic Event Log	48
2.4	Directly-Follows Relation and the Directly-Follows Graph	49
2.4.1	Definition of the Directly-Follows Relation and Graph	50
2.4.2	Example Directly-Follows Graphs	50
2.4.3	Annotated Directly-Follows Graphs	52
2.5	Conclusion	53
<b>3</b>	<b>Related Work</b>	<b>55</b>
3.1	A Taxonomy covering Reverse Engineering and Process Mining	55
3.2	Discovery and Learning Techniques	57
3.2.1	Feature Criteria for Comparison	57
3.2.2	Discussion of Learning and Discovery Techniques	58
3.2.3	Discovery Assumptions and Model Expressivity	63
3.3	Scalability – Hierarchical Decomposition and Submodels	65
3.3.1	Hierarchical Decomposition for Algorithmic Scalability	66
3.3.2	Hierarchies and Submodels for Visual Scalability	68
3.4	Performance Analysis	68
3.4.1	Criteria for Comparison	69
3.4.2	Discussion of Related Work	69
3.5	Conclusion	71
<b>4</b>	<b>A Process Discovery Foundation</b>	<b>73</b>
4.1	Problem Statement	73
4.2	The Inductive Miner Framework	74
4.2.1	Algorithm Overview	74
4.2.2	Base Cases	75
4.2.3	Finding Cuts	76
4.2.4	Splitting Logs	79
4.2.5	Fallback Cases	81
4.3	Guarantees	84
4.3.1	Soundness and Termination	84
4.3.2	Perfect Fitness	84
4.3.3	Language Rediscoverability	84
4.3.4	Polynomial Runtime Complexity	86
4.4	Existing Behavioral and Scalability Extensions	86
4.5	Open Challenges and Our Approach	87

## II Hierarchical Process Discovery

<b>5</b>	<b>On Software Data and Behavior</b>	<b>91</b>
5.1	Software Data Sources and Case Identification	91
5.1.1	Log Files, Monitoring, Tracing, and Instrumentation	91
5.1.2	Software Environment and Stimuli	93
5.1.3	Event Structuring and Case Identification	94
5.2	Business Event Logs versus Software Event Logs	96
5.2.1	Log Size Comparison	96
5.2.2	Behavior Comparison and Accompanied Challenges	99
5.3	Software Event Data	102
5.3.1	Terminology	102
5.3.2	Event Type and Lifecycle Transactions	103
5.3.3	Event Location	105
5.3.4	Method Parameter Data	106
5.3.5	Application Metadata	106
5.3.6	Runtime Data	106
5.3.7	Exception Data	107
5.4	Conclusion	107
<b>6</b>	<b>Hierarchical and Recursion Aware Discovery</b>	<b>109</b>
6.1	Why We Need Hierarchy and Recursion – Reality Is Not Flat	109
6.2	Hierarchical Event Logs	111
6.2.1	Example Log of Executing a Program	111
6.2.2	The Hierarchical Event Log	111
6.2.3	The Atomic Hierarchical Event Log	114
6.2.4	Transformations – Heuristics for Hierarchy	116
6.3	Hierarchical Process Trees	117
6.3.1	Example Model of a Program Execution	117
6.3.2	Syntax and Semantics	119
6.4	Naïve Hierarchical Discovery	122
6.4.1	Algorithm Overview	122
6.4.2	Framework Extensions	123
6.4.3	Discovery Examples	125
6.4.4	Guarantees	126
6.5	Recursion Aware Discovery	128
6.5.1	Algorithm Overview	129
6.5.2	Framework Extensions	130
6.5.3	Discovery Examples	134
6.5.4	Guarantees	135

6.6	Compatibility with Other Extensions	137
6.7	Evaluation	138
6.7.1	Evaluation using Synthetic Logs	138
6.7.2	Performance and Scalability Evaluation	150
6.8	Conclusion and Open Challenges	159
<b>7</b>	<b>Cancelation Discovery</b>	<b>163</b>
7.1	Why We Need Cancelation – The Exceptional Case	163
7.2	Cancelation Process Trees	165
7.2.1	Example Model of a Program Execution	166
7.2.2	Syntax and Semantics	167
7.3	The Trigger Oracle	173
7.3.1	Trigger Oracles in Software	173
7.3.2	Non-software-specific Trigger Oracles	174
7.4	Cancelation Discovery	175
7.4.1	Algorithm Overview	175
7.4.2	Framework Extensions	175
7.4.3	Splitting Logs	181
7.4.4	Discovery Examples	182
7.4.5	Guarantees	183
7.5	Compatibility with Other Extensions	186
7.6	Evaluation	187
7.6.1	Evaluation using Synthetic Logs	187
7.6.2	Performance and Scalability Evaluation	196
7.7	Conclusion and Open Challenges	204

### III Beyond Model Discovery

<b>8</b>	<b>Hierarchical Performance Analysis</b>	<b>209</b>
8.1	Why We Need Hierarchical Performance Analysis	209
8.2	Introduction to Alignments	213
8.2.1	Unfolding Hierarchical Models	213
8.2.2	Alignments	215
8.2.3	Optimal Alignments	217
8.3	Analysis Framework Definition	219
8.3.1	Move Enablers and Execution Policies	219
8.3.2	Execution Intervals	223
8.3.3	Interval Correlation and Computation	225
8.3.4	Execution Subtraces	227

8.4	Metrics for Execution Intervals	228
8.4.1	Semantic-aware Filters	229
8.4.2	Metric Formalizations	230
8.4.3	Interpreting Metric Results	237
8.4.4	Example Metric Applications	238
8.5	Evaluation	239
8.5.1	Comparative Evaluation	239
8.5.2	Performance and Scalability Evaluation	253
8.6	Conclusion and Open Challenges	255
<b>9</b>	<b>Translations and Traceability</b>	<b>259</b>
9.1	The Translations and Traceability Framework	259
9.1.1	Translations	259
9.1.2	Traceability	261
9.2	Basic and Extended Petri net Interpretations	261
9.2.1	Basic Process Trees to Basic Petri nets	263
9.2.2	Mapping Named Submodels and Recursive References	263
9.2.3	Mapping Sequence Cancelations and Loop Cancelations	264
9.3	Model to Model Translations	266
9.3.1	Extended Process Trees to YAWL	266
9.3.2	Extended Process Trees to BPMN	266
9.3.3	Extended Process Trees to Statecharts	266
9.3.4	Extended Process Trees to Message Sequence Diagrams	267
9.4	Conclusion and Open Challenges	273

## IV Applications

<b>10</b>	<b>Tool Implementations</b>	<b>277</b>
10.1	Introduction	277
10.2	The Statechart Workbench	278
10.2.1	Overview and Design Decisions	278
10.2.2	Software Architecture	279
10.2.3	Tool Walkthrough	283
10.3	The Instrumentation Agent	289
10.3.1	Inside the Instrumentation Agent	289
10.3.2	Logging Software Events	293
10.4	The SAW Eclipse Plugin	294
10.4.1	The SAW Instrumentation Agent Integration	294
10.4.2	The ProM-Eclipse Link	294

10.5	User Experience Evaluation	296
10.5.1	Methodology	296
10.5.2	Evaluation Results	297
10.5.3	Threats to Validity	300
10.6	Conclusion	301
<b>11</b>	<b>The Software Process Analysis Methodology</b>	<b>303</b>
11.1	Introduction and Positioning	303
11.1.1	Why we Need a Software Process Analysis Methodology	303
11.1.2	Related Methodologies	304
11.2	The Software Process Analysis Methodology	304
11.2.1	Phase 1 – Planning and Scoping	306
11.2.2	Phase 2 – Instrumentation and Data Extraction	307
11.2.3	Phase 3 – Data Understanding and Preparation	309
11.2.4	Phase 4 – Mining, Modeling, and Analysis	311
11.2.5	Phase 5 – Evaluation and Rescoping	314
11.2.6	Phase 6 – Deployment	316
11.3	Practical Concerns	318
11.3.1	Completeness of Software Observations	318
11.3.2	Timings and Clock Considerations	319
11.3.3	Iterate and Iterate Often	320
11.3.4	Visual Analysis Workflow	320
11.4	Conclusion	321
<b>12</b>	<b>Case Studies</b>	<b>323</b>
12.1	Introduction	323
12.2	Open Source Software Case Study – The JUnit 4.12 Library	323
12.2.1	Case Description	324
12.2.2	Initial Data Extraction and Understanding	324
12.2.3	High-Level Process Understanding	327
12.2.4	Detailed Exploration and Design Patterns	330
12.2.5	Threats to Validity	333
12.3	Industrial Software Case Study – The Wafer Handling Process	335
12.3.1	Case Description	335
12.3.2	Initial Data Extraction and Understanding	336
12.3.3	Basic Process Understanding	339
12.3.4	Detailed Process Understanding	343
12.3.5	Follow-up Process Explorations	348
12.3.6	Threats to Validity	349
12.4	Conclusion	349

## V Closure

<b>13</b>	<b>Conclusion</b>	<b>353</b>
13.1	Contributions	353
13.1.1	Part II – Hierarchical Process Discovery	354
13.1.2	Part III – Beyond Model Discovery	355
13.1.3	Part IV – Applications	355
13.2	Limitations	356
13.2.1	Part II – Hierarchical Process Discovery	356
13.2.2	Part III – Beyond Model Discovery	357
13.2.3	Part IV – Applications	358
13.3	Future Work	359
13.3.1	Part II – Hierarchical Process Discovery	359
13.3.2	Part III – Beyond Model Discovery	360
13.3.3	Part IV – Applications	361
13.4	Broader Outlook on Process Mining and Software Engineering	361
<b>A</b>	<b>Proofs</b>	<b>365</b>
A.1	Proof for Chapter 4 – A Process Discovery Foundation	365
A.1.1	Soundness and Termination	365
A.1.2	Perfect Fitness	365
A.1.3	Language Rediscoverability	366
A.1.4	Polynomial Runtime Complexity	367
A.2	Proof for Section 6.4 – Naïve Hierarchical Discovery	368
A.2.1	Soundness and Termination	368
A.2.2	Perfect Fitness	369
A.2.3	Language Rediscoverability	370
A.2.4	Polynomial Runtime Complexity	371
A.3	Proof for Section 6.5 – Recursion Aware Discovery	371
A.3.1	Soundness and Termination	371
A.3.2	Perfect Fitness	372
A.3.3	Language Rediscoverability	373
A.3.4	Polynomial Runtime Complexity	374
A.4	Proof for Chapter 7 – Cancellation Discovery	375
A.4.1	Soundness and Termination	375
A.4.2	Perfect Fitness	376
A.4.3	Language Rediscoverability	377
A.4.4	Polynomial Runtime Complexity	380



Bibliography	381
Summary	401
Samenvatting	403
Acknowledgments	405
Curriculum Vitae	407
SIKS Dissertations	411
List of Figures	421
List of Tables	425
List of Symbols	427
Index	431

# I | Introduction

<b>1</b>	<b>Overview</b> .....	<b>3</b>
1.1	Process Mining	
1.2	Software Engineering and Analysis	
1.3	Open Challenges in Process Mining for Analyzing Software	
1.4	Our Approach – Design Decisions and Foundation	
1.5	Contributions and Structure of this Thesis	
<b>2</b>	<b>Preliminaries</b> .....	<b>21</b>
2.1	Basic Notation	
2.2	Process Model Notations	
2.3	Event Logs	
2.4	Directly-Follows Relation and the Directly-Follows Graph	
2.5	Conclusion	
<b>3</b>	<b>Related Work</b> .....	<b>55</b>
3.1	A Taxonomy covering Reverse Engineering and Process Mining	
3.2	Discovery and Learning Techniques	
3.3	Scalability – Hierarchical Decomposition and Submodels	
3.4	Performance Analysis	
3.5	Conclusion	
<b>4</b>	<b>A Process Discovery Foundation</b> .....	<b>73</b>
4.1	Problem Statement	
4.2	The Inductive Miner Framework	
4.3	Guarantees	
4.4	Existing Behavioral and Scalability Extensions	
4.5	Open Challenges and Our Approach	

---

## I Introduction

Chapter 1  
Overview

Chapter 2  
Preliminaries

Chapter 3  
Related Work

Chapter 4  
A Process Mining  
Foundation

---

## II Hierarchical Process Discovery

Chapter 5  
On Software Data  
and Behavior

Chapter 6  
Hierarchical and Recursion  
Aware Discovery

Chapter 7  
Cancellation Discovery

---

## III Beyond Model Discovery

Chapter 8  
Hierarchical Performance  
Analysis

Chapter 9  
Translations  
and Tracability

---

## IV Applications

Chapter 10  
Tool Implementations

Chapter 11  
The Software Process  
Analysis Methodology

Chapter 12  
Case Studies

---

## V Closure

Chapter 13  
Conclusion

Appendix A  
Proofs

In Part I, we set the scene for the rest of the thesis.

**Chapter 1** starts with an introduction to Process Mining, Software Analysis, and the research problems addressed in this thesis.

**Chapter 2** introduces preliminaries, basic notations and model languages, and some basic process mining concepts.

**Chapter 3** positions this thesis and discusses related work.

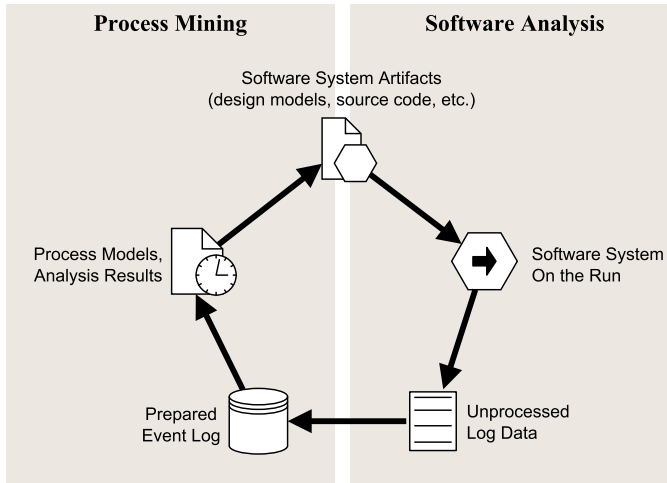
**Chapter 4** introduces a basic process mining foundation upon which the techniques in the rest of the thesis build.

# 1 | Overview

In today’s world, we increasingly rely on information technology. Complex software-driven systems can be found in all sectors: communication, production, distribution, healthcare, transportation, education, entertainment, government, trade, etc. Thus it comes as no surprise that software-related problems can have an incredible impact on society, organizations, and users. Specification, verification and testing techniques attempt to avoid such problems. However, in practice, these techniques fall short because software continues to evolve and grow in complexity, and operates in an ever-changing environment [7]. In short, one cannot anticipate all problems at design-time or during testing.

To date, the computer science discipline has tried to address such problems by proposing new design methodologies and reverse-engineering tools. However, these *a priori* techniques have inherent limitations: we cannot predict evolving requirements and circumstances at design time, and numerous examples show that traditional approaches cannot cope with the complexity of today’s information systems [7]. To address and possibly avoid these problems, one has to observe, monitor and study software systems “on the run”, preferably in its natural, real-life production environment.

Event log data capturing actual system behaviors are being recorded everywhere: in enterprise information systems and business transaction logs, in web servers, in high-tech systems such as X-ray machines and wafer scanners, in warehousing systems, etc. [8] Although considerable amounts of data are recorded by software, machines, and organizations, problems are typically only addressed in a trial-and-error and ad-hoc fashion. Much of the information in the recorded data is ignored in this way. Especially in the case of (legacy) system comprehension, analysis, maintenance, and evolution, this data could give us insight into the actual structure, behavior, operation, and usage of the observed systems. In short, there is a need to exploit this rich data in a more systematical fashion. Especially in the case of software systems, where malfunctions can and will happen, it is important to study these systems in their natural environment to better understand the problems and to minimize their impact [7].

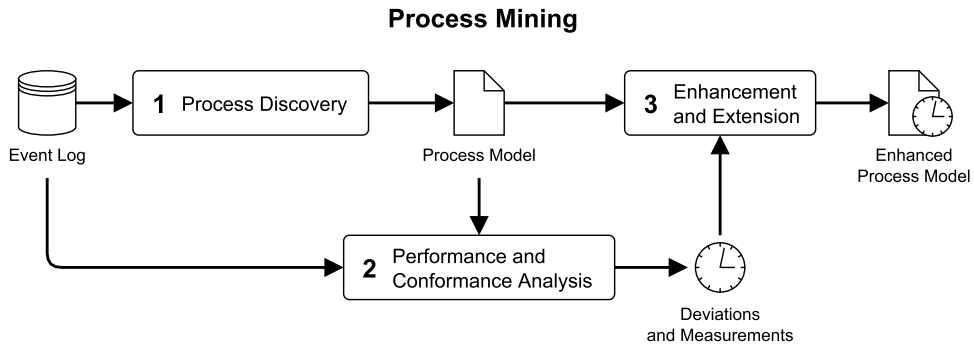


**Figure 1.1:** The software process analysis lifecycle, showing the relation between software artifacts, software analysis, and process mining. The arrows indicate the most important dependencies.

In the situation described above, *process mining* is a good candidate for the analysis of *actual system behavior*. Process mining techniques provide a powerful and mature way to discover formal process models and analyze and improve these processes based on the *event log* data that is already being recorded. In this thesis, we will show, amongst others, how we can use process mining for analyzing software systems. The software process analysis lifecycle, as depicted in Figure 1.1, illustrates the relation between software artifacts, software analysis, and process mining. That is, we will look at log data obtained from software systems on the run, capturing actual system behavior. Using this recorded behavior, we will apply process mining to gain insight into the software systems, and we will relate the results back to the underlying software system artifacts.

## 1.1 Process Mining

The field of *process mining* [8] aims to extract information from recorded *event log* data. Such an event log typically consists of *traces*, where each trace represents one end-to-end execution of a system or process. Each trace consists of records of all the steps executed (*events*), detailing what happened for a particular execution. Typically, each event relates to a particular process step, called an *activity*. A trace details, for example, how a visitor used a website, how a product is processed by a machine, how an insurance company processes a claim, or how a programming interface (API) is used in a software system.



**Figure 1.2:** Process mining and the three major classes of techniques focused on in this thesis: process discovery, performance and conformance analysis, and enhancement.

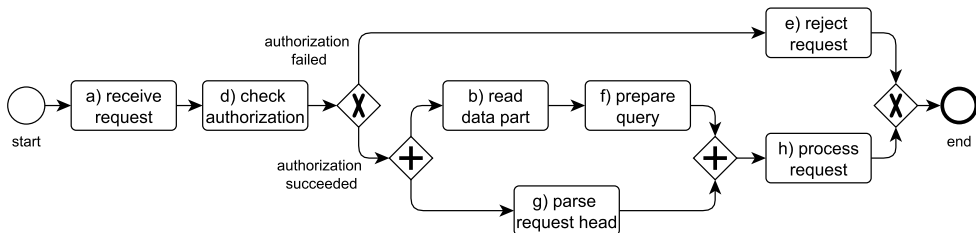
Process mining techniques use event logs to solve an array of different problems. In this thesis, we will address three major classes of techniques: 1) *Process discovery* tries to find new process models to describe the behavior in the event log. 2) *Performance and conformance analysis* aims to investigate and verify recorded behavior with a provided process model. 3) *Enhancement* attempts to combine the result of performance and conformance results with the process model, i.e., to project the measured results back onto the model at the right place. Figure 1.2 illustrates these techniques in the context of process mining.

### 1.1.1 Event logs

The starting point for process mining is an event log. Table 1.1 shows an example of such an event log for a web service provider process. It shows several traces, each with their own case id. Each row represents one event, detailing which activity was executed, and all the data associated with that event. In the first trace, with case id 1, we see that the process started with a *receive request* (events 1.1 and 1.2), followed by a *parse request head* (event 1.3) and a *check authorization* (events 1.4 and 1.5). Finally, at events 1.6 and 1.7, a *reject request* was recorded. Observe how for each step, both the *start* and *complete* (end) time was recorded, as well as which thread (which *resource*) executed the step. For example, from events 1.1 and 1.2 we know that the *main-thread* worked on *receive request* from 11:02:45.000 till 11:02:45.500. These are just some examples of the type of data one can find in an event log. Observe how each trace explains a different example run of the underlying process. For example, in *case 1*, the request was declined, and in *case 2*, the request was processed.

**Table 1.1:** Example snippet of an event log for a web service provider process. Each row is an event, and each column an attribute. Events are grouped into traces, as indicated by the horizontal lines and case id column.

Case id	Event id	Attributes					
		Activity	Lifecycle	Timestamp	Resource	...	
1	1.1	<i>a</i>	receive request	start	30-10-2017 11:02:45.000	main-thread	...
	1.2	<i>a</i>	receive request	complete	30-10-2017 11:02:45.500	main-thread	...
	1.3	<i>g</i>	parse request head	start	30-10-2017 11:02:45.650	worker-1	...
	1.4	<i>d</i>	check authorization	start	30-10-2017 11:02:45.651	worker-3	...
	1.5	<i>d</i>	check authorization	complete	30-10-2017 11:02:45.710	worker-3	...
	1.6	<i>e</i>	reject request	start	30-10-2017 11:02:45.820	main-thread	...
	1.7	<i>e</i>	reject request	complete	30-10-2017 11:02:45.870	main-thread	...
2	2.1	<i>a</i>	receive request	start	30-10-2017 11:03:12.150	main-thread	...
	2.2	<i>a</i>	receive request	complete	30-10-2017 11:03:12.450	main-thread	...
	2.3	<i>g</i>	parse request head	start	30-10-2017 11:03:12.670	worker-2	...
	2.4	<i>b</i>	read data part	start	30-10-2017 11:03:12.670	worker-1	...
	2.5	<i>g</i>	parse request head	complete	30-10-2017 11:03:13.110	worker-2	...
	2.6	<i>b</i>	read data part	complete	30-10-2017 11:03:13.160	worker-1	...
	2.7	<i>f</i>	prepare query	start	30-10-2017 11:03:13.320	worker-1	...
	2.8	<i>f</i>	prepare query	complete	30-10-2017 11:03:13.400	worker-1	...
	2.9	<i>h</i>	process request	start	30-10-2017 11:03:13.670	main-thread	...
	2.10	<i>h</i>	process request	complete	30-10-2017 11:03:14.220	main-thread	...
3	3.1	<i>a</i>	receive request	start	31-10-2017 09:43:16.030	main-thread	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮



**Figure 1.3:** A BPMN model discovered for the web service provider process. The X-diamond represents a choice; the +-diamond represents concurrency or parallel fork/join. Note that this model does *not* perfectly fit the example behavior from the event log in Table 1.1. For example, in case 1, activity *g* (event 1.3) was executed before activity *d* was performed.

### 1.1.2 Process Discovery

Within the area of process mining, the main focus of research has been on process discovery. The goal of process discovery is, given only an event log, construct a process model describing the behavior recorded in the event log. Such a process model is typically expressed in a formalism such as a Petri net or a process tree, and can be visualized using various notations, e.g., BPMN,

Statecharts, etc. For example, from the event log in Table 1.1, the model in Figure 1.3 could be discovered. In this model, after *receive request* and *check authorization*, there is a choice between failed and succeeded authorization. When authorization failed, a *reject request* is performed. When authorization succeeded, two paths are executed in parallel: 1) a *parse request head* is executed, and concurrently 2) a *read data part* and *prepare query* is executed. After both paths are completed, a *process request* is performed.

Numerous process discovery techniques and algorithms have been proposed in the literature. Despite all this effort, process discovery still remains challenging, with many open problems. In this section, we will briefly discuss the main challenges in process discovery.

### Sound Semantics

One challenge is that process discovery techniques should yield models with clear and well-defined semantics that are free of deadlocks and other anomalies. Clear semantics is a prerequisite for reliable interpretation by both man and machine. For example, without clear semantics, one cannot reliably perform performance and conformance analysis. The discovered models should also be *sound*, i.e., they should be free of deadlocks and other anomalies. Ambiguity and unclear behavior should be avoided, all process steps should be executable, and an end state should always be reachable. Without soundness, the model, and any conclusion derived from it, is unreliable.

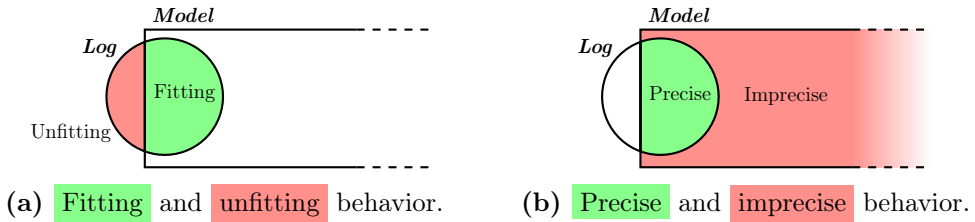
It should be noted that in certain cases it is beneficial to remain vague, especially when there is not enough “evidence” in the data or when the modeling constructs do not fit the observed behavior [11]. In addition, many commercial tools deliberately remain vague in order to scale well and produce simple models [51, 72, 76]. However, especially when analyzing software, precise and formal models are preferred for reliable reasoning and analysis.

### Fitness and Precision

Another challenge is that the discovered models should represent the behavior recorded in the event log. We can measure how much a model and log are related using various quality metrics. In this thesis, we will mainly focus on *fitness* and *precision*, but other metrics are defined as well [46]. We will explain both metrics using the Venn diagrams in Figure 1.4.

*Fitness* expresses the part of behavior in the event log that is also captured in the model. In Figure 1.4a, the green filled area indicates fitting behavior, whereas the red filled area indicates unfitting behavior. In order to draw conclusions based on a discovered model, that model should represent the event log well, i.e., the model should have a *high fitness*. With a low fitness, one cannot reliably analyze the recorded behavior with the discovered model: the model has too many *unfitting behavior*. Conclusions such as the *absence*





**Figure 1.4:** Fitness and precision metrics illustrated. The circles represent the finite behavior recorded in the event log. The squares represent the possibly infinite behavior captured by the model. The areas indicate the overlap between logged and modeled behavior.

of behavior or process rule violations would become unreliable when based on an unfitting model.

*Precision* expresses the part of the behavior in the model that is also present in the event log. In Figure 1.4b, the green filled area indicates precise behavior, whereas the red filled area indicates imprecise behavior. Since an event log only captures observed behavior over a period of time, and thus is likely incomplete, process discovery algorithms usually apply some generalizations. That is, the discovered model allows for behavior beyond what is recorded in the log. Although this imprecision can help in discovering a more fitting and simpler model, it is important that such imprecisions are limited. With a low precision, the modeled behavior is not supported by observations in the event log. Conclusions such as the *presence* of behavior or process rule violations would become unreliable when based on an imprecise model.

For example, the model in Figure 1.3 has a high fitness and precision but it is not perfectly fitting the example behavior from the event log in Table 1.1. For example, in case 1, activity *g* is executed too soon, and in case 2, activity *d* was illegally skipped.

### Supported Behavior

The model formalism and discovery algorithm should support the type of behavior that is recorded in the event log. For example, when one observes a web service provider process, and the whole application should be canceled once an authorization check fails, then the model formalism and discovery algorithm should support such cancelation behavior. Likewise, when one observes a recursive software process, then the model formalism and discovery algorithm should support such hierarchical and recursive process model constructs.

Any discovery technique must decide on which type of behavior and process model constructs it supports, i.e., it must decide on which class of behavior it supports and leverage this *representational bias*.

Lion's share of existing process discovery techniques assume that activity executions are instantaneous (*atomic*). Recent work started to explore also *non-atomic* event logs, amongst others to support various notions of concurrency. However, most of these approaches do not address notions like hierarchical subprocesses, recursive behavior, or cancelation behavior. Process mining techniques should be aware of such constructs in order to benefit from the extra information it provides.

### Scalability

Finally, the discovery algorithm and produced models should scale well. This challenge concerns both the algorithmic and visual aspects of scalability.

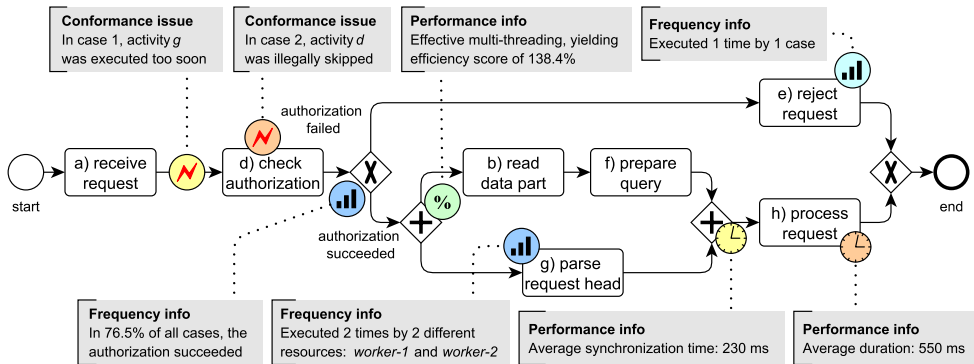
First of all, the process mining algorithms should scale well with large inputs. That is, these techniques should be able to handle large, real-life event logs within time and memory constraints. Current state-of-the-art process discovery techniques already scale well with a large number of events and traces, but are mainly limited by the number of activities or length of traces. Recent advances in distributed process mining split computationally challenging process mining problems into many smaller problems that can be analyzed easily and whose results can be combined into solutions for the original problems. In addition, state-of-the-art streaming process mining techniques enable the analysis of running systems and event logs that are too large to fit main memory. However, more support is needed to improve the *algorithmic scalability* in order to handle more complex event logs consisting of long traces with many activities and complex relations. This is especially true in the case of software systems, where the observed processes may produce extremely long traces with hundreds of thousands of events and hundreds of activities in each trace.

Secondly, scalability also concerns the presentation of the resulting models, i.e., the *visual scalability*. That is, even when a model consists of hundreds of activities, the user should be able to grasp the presented complexity. Clearly, this calls for proper process model constructs and tool support to aid the user. Modeling languages like BPMN have recognized this need for visual scalability by supporting process model constructs like subprocesses, cancelations, etc. However, very few discovery techniques support and use such constructs to reduce the modeled complexity.

### 1.1.3 Performance and Conformance Analysis

When both an event log and a process model are given, one can compare both to check the quality of the model and analyze performance information. The process model may have been constructed by hand (e.g., a reference model), or may have been discovered.

*Conformance checking* relates events in the event log to activities in the process model and compares both. The goal is to find commonalities and



**Figure 1.5:** An annotated model learned from the web service provider process. This example combined the event log in Table 1.1 and the model in Figure 1.3 to show performance and conformance information in the context of the model.

discrepancies between the modeled behavior and the observed behavior. When an event log and process model do not agree, these discrepancies might indicate undesirable deviations, fraud, inefficiencies, or other issues.

For example, the model in Figure 1.3 is not perfectly fitting the example behavior from the event log in Table 1.1. Conformance checking reveals that in case 1, activity *g* is executed too soon, and in case 2, activity *d* was illegally skipped. Figure 1.5 shows these deviations in the context of the model.

*Performance analysis* uses the conformance checking results to measure the performance of a process. Through conformance checking, events in the log are coupled to elements in the model. This coupling allows for precise performance analysis and diagnostics. Based on the event data linked to a model element, durations, service times, waiting times, resource usage, and more can be derived and evaluated.

For example, the model in Figure 1.5 shows various frequency and performance annotations in the context of the model. Such annotations could indicate bottlenecks or less frequently executed activities.

### 1.1.4 Enhancement

*Enhancement* techniques take both an event log and a process model to *extend or improve the model with information extracted from the event log*. There are various enhancements one can perform. Simple enhancements extend or annotate the process model with conformance information such as deviations and performance information such as frequency and timing information. More advanced enhancements animate event data over process models or repair process models to better reflect the recorded behavior.

For example, by annotating conformance and performance results onto the model in Figure 1.5, deviations and performance issues can be located in the context of the model. This allows us to identify *which parts of the process* are problematic in terms of, for example, deviations or durations.

Enhanced process models consolidate information from multiple data perspectives and provide a lot of insight. The insights gained from enhancement techniques may often lead to new analysis questions. In practice, this often leads to an explorative process mining approach, where insights from enhanced process models lead to new analyses using different data filters and different perspectives.

## 1.2 Software Engineering and Analysis

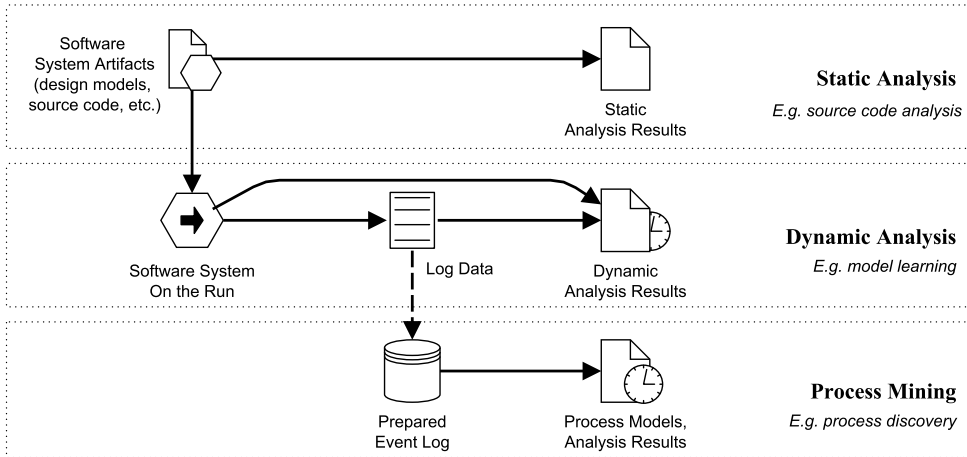
Over the last few decades, the complexity of software systems has increased considerably. These software systems have become an indispensable and integral part of everyday life and consist of millions of lines of code, written by thousands of different programmers over the last few decades. As these systems continue to grow and evolve, failures, performance issues, and downtime happen more often. Since these software systems are an accumulation of business logic and lessons learned in the field, simply fixing or rewriting the failing parts is not a trivial task. Even worse, in practice, knowledge and documentation of these ever-evolving systems are typically missing or outdated; we are dealing with *legacy software*. Such knowledge and documentation regarding the structure, behavior, operation, and usage of these software systems is crucial for system comprehension, analysis, maintenance, and evolution. When no complete and up-to-date information is available, one has to extract this information through software analysis techniques such as model learning.

### 1.2.1 Software Analysis and Model Learning

Many software analysis and reverse engineering techniques have been proposed in the software engineering domain. We typically can partition such techniques into static analysis and dynamic analysis.

*Static analysis* derives its analyses and models without actually executing the software (see the top row in Figure 1.6). It uses source code, object code, byte code, or other static sources as input. Instead of running the code, one tries to understand the internal logic of the software and how it (statically) connects with other parts. With static analysis, design models are inferred, and behavior is estimated and approximated.

In contrast, *dynamic analysis* techniques, such as *model learning*, derives its analyses and models by executing software programs on a real or virtual processor (see the middle row in Figure 1.6). Using information obtained from interactions with the software, log recordings, instrumentation, and debuggers,



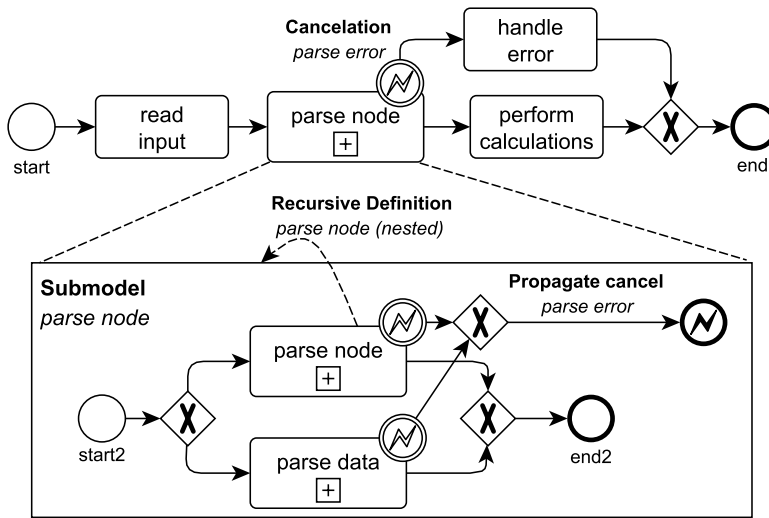
**Figure 1.6:** Analyzing software with static analysis, dynamic analysis, and process mining. Static analysis only uses the static software system artifacts. Dynamic analysis looks at the runtime behavior. Process mining extracts information from structured event log data.

runtime behavior is captured and modeled. To understand the operation and usage of a system, one has to observe and study the system “on the run”, preferably in its natural, real-life production environment. To understand and maintain the behavior of legacy systems when design and documentation are outdated, one can observe and study the system in a controlled environment using, for example, testing techniques.

### 1.2.2 Process Mining for Software Analysis

As noted before, in the process mining community, model (re)construction from logs is commonly referred to as *process discovery*. In contrast to many software analysis techniques, process mining provides a powerful and mature way of combining and integrating both model (re)construction and analysis. That is, process mining combines extracting (formal) enriched and annotated models and enabling performance and conformance analysis using these models. With the use of structured and well-defined event logs (see the bottom row in Figure 1.6), numerous existing techniques can be combined for advanced analyses, yielding results like in Figure 1.5. Chapter 3 elaborates more on the commonalities and differences between static analysis, dynamic analysis, model learning, and process mining.

When applying process mining on event data originating from software systems, new patterns and challenges pop up. For example, software event data contains structured hierarchies not commonly found in business process



**Figure 1.7:** Illustrative BPMN example of named submodels, recursion, and cancellation. The X-diamonds represent choice. This model shows a fictive process for parsing a recursive tree-like data structure.

event logs. Hierarchies can arise from structural relations such as dependencies in client-server relations, communicating software components, or object relations. When focusing on runtime structures, hierarchies can arise from call-relations amongst functions, methods, or co-routine invocations. These hierarchical structures should be exploited in all process mining tasks. This is exactly the core property we will investigate and exploit in the techniques in this thesis. Chapter 5 elaborates more on these patterns and challenges.

To illustrate some of the structured hierarchies that can be found in software, consider the example of a process for parsing a recursive tree-like data structure. Figure 1.7 presents a model for this example. At the top, the main process is shown: a *read input* is followed by a *parse node* and a *perform calculations*. The step *parse node* consists of multiple substeps, and is given in more detail in the **submodel** *parse node*. The *parse node* submodel has a choice between two steps: either the *parse data* submodel (not shown) is executed, or the *parse node* step is executed, which refers back to the *parse node* submodel. Hence, the *parse node* submodel is defined **recursively** in terms of itself, parsing our fictive tree-like data structure level by level, node by node. At any time, a *parse error* exception can occur. The submodel *parse node* propagates this error upwards, **canceling** any and all behavior. At the top, this *parse error* is **caught**, and the alternative *handle error* step is performed. Clearly, such behavior cannot (easily) be captured in a flat model.

### 1.3 Open Challenges in Process Mining for Analyzing Software

The application of process mining for analyzing actual software system behavior leads to new challenges. On the one hand, there is the lack of support for the type of behavior present in software system settings. On the other hand, there is a lack of support for integrating process mining in the software process analysis lifecycle (Figure 1.1). Below we discuss the open challenges addressed in this thesis.

**Challenge 1 — Lack of structured techniques for the recording and processing of software event data.** Process mining starts with an event log. Although event log data is captured and recorded everywhere nowadays, there is no structured format or standard detailing what type of information is recorded in which way. This lack of a structured standard and accompanied tool support limits the application of event data and process mining. Furthermore, the transformation of software logging data to event log traces is far from trivial. In practice, one has to decide on what constitutes a software event, and how cases/traces are defined in a software analysis setting.

**Challenge 2 — Lack of support for hierarchical models with named submodels.** Software behavior is, by design, hierarchical. Hierarchies can arise from structural relations such as dependencies in client-server relations, communicating software components, or object relations. When focusing on runtime structures, hierarchies can arise from, for example, call-relations amongst functions, methods, or co-routine invocations. See, for example, the *parse node* and *parse data* submodel in Figure 1.7. In addition, in the context of business processes, low-level events recorded by information systems may not directly match high-level activities that make sense to process stakeholders [139]. Often, also in these cases, a hierarchy of business activities can be identified. These hierarchical structures should be exploited in all process mining tasks. Moreover, in most cases, the data contains enough information to extract hierarchies with named submodels, which can be exploited for discovery and analysis purposes. However, there is little work in either process discovery or analysis that exploits hierarchical models with named submodels.

**Challenge 3 — Lack of support for recursive behavior.** Event logs are finite and thus often incomplete observations of system behavior. Hence, process discovery algorithms generalize over the behavior in the event log to deduce the behavior of the observed system. For example, when one observes a repeated occurrence of an activity  $a$ , one can deduce an iteration or repetition and generalize the observed behavior to a loop. Similarly, one can observe a repeated occurrence down a hierarchy. This repetition down a hierarchy can be seen as recursive behavior. See, for example, the recursive definition of the *parse node* submodel in Figure 1.7. Especially when modeling software behavior, proper

support for named submodels with recursion is a must. Current techniques lack support for modeling and discovering such recursive behavior.

**Challenge 4 — Lack of support for cancelation behavior.** The majority of real-life event logs contain some form of cancelation or error-handling behavior. A bank loan request may be canceled or declined, a web server needs to handle a connection error, an X-ray machine may detect a sensor problem, etc. See also, for example, the cancelation involving *parse error* in Figure 1.7. This type of cancelation behavior can easily be expressed in existing modeling formalisms, but few existing process discovery techniques actually take these cancelation features into account. Without cancelation support, discovery algorithms can produce needlessly complex and imprecise models.

**Challenge 5 — Separation of visualization and representational bias.** Process mining techniques require a representational bias to make assumptions about the modeled behavior. However, the modeled behavior can be visualized in many different formalisms. In traditional business process mining, visual notations like BPMN, EPC and Petri nets are common. In software engineering, visual notations like UML sequence diagrams, finite state machines, and call graphs are more common. In practice, a combination of different visual notations should be used for different process analysis questions. Ideally, process mining techniques should not be limited by the bias and constraints arising from these visual notations. Rather, they should use an internal representation whose bias is the “greatest common denominator” of the supported visual representations.

**Challenge 6 — Lack of support for integrating results with existing software artifacts.** Analyzing software starts with the actual, runnable software system itself. From this running software system, one needs to extract event data before any process mining can be applied. Before this thesis, there were no systematic approaches for extracting event logs from running software systems suitable for process mining. Most logging and tracing functionality use custom or proprietary logging formats that cannot directly be used for process mining. Secondly, for a true round-trip analysis, any insights gained from the process mining analysis need to be related back to the software system domain. Hence, there should be support for tracing process mining results back to software artifacts like the actual source code that generated the event data.

**Challenge 7 — Improve understandability for non-experts and software analysts.** Recent work in process mining has shifted from technical Petri net visualizations to more user-friendly BPMN and flow-map visualizations. By providing simple models enhanced with conformance and performance analysis results, non-experts can quickly see and interact with the results in a language they understand. Similarly, when applying process mining to software, suitable visualizations and interactions need to be chosen. This not only means choosing



suitable (domain) model visualizations, but also means that the results should be relatable and traceable to software domain concepts.

**Challenge 8 — Improve algorithmic and visual scalability.** The discovery algorithm and produced models should scale well. This challenge concerns both the algorithmic and visual aspects of scalability. That is, first of all, the techniques should be able to handle large, real-life event logs within time and memory constraints (i.e., the algorithmic scalability). Current state-of-the-art process discovery techniques already scale well with a large number of events and traces, but are mainly limited by the number of activities and trace length. Support is needed to handle more complex processes consisting of many activities. This is especially true in the case of software systems, where the observed processes can consist of long traces with hundreds of thousands of events and hundreds of activities in each trace. Moreover, scalability also concerns the presentation of the resulting models (i.e., the visual scalability). That is, even when a model consists of hundreds of activities, the user should be able to grasp the presented complexity. Clearly, this calls for proper process model constructs and tool support to aid the user.

## 1.4 Our Approach – Design Decisions and Foundation

In this thesis, we will present techniques and tools based on process mining for analyzing software behavior. Our work has been implemented as a plugin for the process mining framework ProM [187]. We will reuse the well-established IEEE XES event log standard [77, 187] to exchange software event data, see also Section 2.3.2. This way, all techniques and tools from this thesis can be used on any existing event log dataset, even if the dataset did not originate from software systems, and our output results can be used with other existing tools and ProM plugins.

At the core of this thesis, we will present hierarchical, recursion aware, and cancellation discovery techniques based on the *process tree* notation, as introduced in Section 2.2.6. For these techniques, we will build on an existing process discovery foundation to discover process trees, and extend the modeling notation and discovery technique where needed. Since process trees already capture a hierarchical relation, they naturally allow for our novel and explicit hierarchical extensions. We selected the well-known *Inductive Miner (IM) framework*, as described in [130] and summarized in Chapter 4, as our process discovery foundation. This framework offers good discovery guarantees, scales well, and provides clear extension points for our adaptations.

The discovered models can be exported as Petri net models to be used in other tools and ProM plugins. We reuse and build upon the mature *alignments* technique [21] to provide hierarchical performance analysis.

## 1.5 Contributions and Structure of this Thesis

This thesis presents hierarchical process mining techniques and shows how these techniques can be used for analyzing software systems in a scalable way. To summarize, this thesis contains the following contributions:

**Contribution 1** — **A detailed discussion of software event data, as well as tool and methodology support for logging such data.** Software behavior and event data differ from business process event logs in several ways. Software event data has different properties and contains different patterns and structures not commonly found in business process event logs. Chapter 5 discusses these properties, patterns, and the accompanied challenges. Chapter 11 presents a methodology for obtaining and analyzing software event log data in a structured way. This contribution aims to address Challenges 1 and 6.

**Contribution 2** — **A modeling notation and discovery techniques for hierarchical and recursive behavior.** As discussed in Challenges 2 and 3, software behavior contains hierarchical and recursive structures. Chapter 6 introduces the *hierarchical process tree* notation to model hierarchical and recursive behavior. In addition, we present several algorithms for discovering hierarchical and recursive models. Our tools leverage the hierarchy to reduce the problem space and modeled complexity, and provide the user with a model in terms of more high-level concepts (Challenge 8).

**Contribution 3** — **A modeling notation and discovery techniques for cancelation behavior.** As discussed in Challenge 4, operational and software processes can contain various forms of cancelation behavior. Chapter 7 introduces the *cancelation process tree* notation to model cancelation regions using process trees. In addition, we present an algorithm for discovering cancelation process trees. These cancelation regions help reduce the modeled complexity (Challenge 8).

**Contribution 4** — **A framework for visualization-independent performance analysis, supporting hierarchy, recursion, and cancelation.** Chapter 8 introduces a hierarchical approach to performance analysis. Our hierarchical performance analysis approach allows us to analyze performance while taking into account hierarchical, recursive, and cancelation behavior (Challenges 2, 3, 4, and 8). Furthermore, novel performance metrics bring process performance analysis more into the software engineering domain (Challenge 7).

**Contribution 5** — **A family of model translations, supporting hierarchy, recursion, and cancelation.** Leveraging the process tree extensions mentioned above, Chapter 9 provides an extensive model translation framework, taking into account the hierarchical, recursive, and cancelation semantics. This way, we separate visualization and representation bias and allow our results to be mapped to more user-friendly visualizations, as discussed in Challenges 5 and 7. In

addition, the translations maintain traceability across models and event logs, allowing to map the results back to the software artifacts, as discussed in Challenge 6.

**Contribution 6 — Extensive tool support for round-trip software analysis.** We provide a rich implementation of the concepts and algorithms presented in this thesis which allows the user to create event logs from their own software (Challenge 1), perform process mining, and link the results back to the source code. Chapter 10 discusses the implemented tools, the interactions amongst tools, the user interactions (Challenge 8), and how our tools allow for integration with existing software artifacts (Challenge 6).

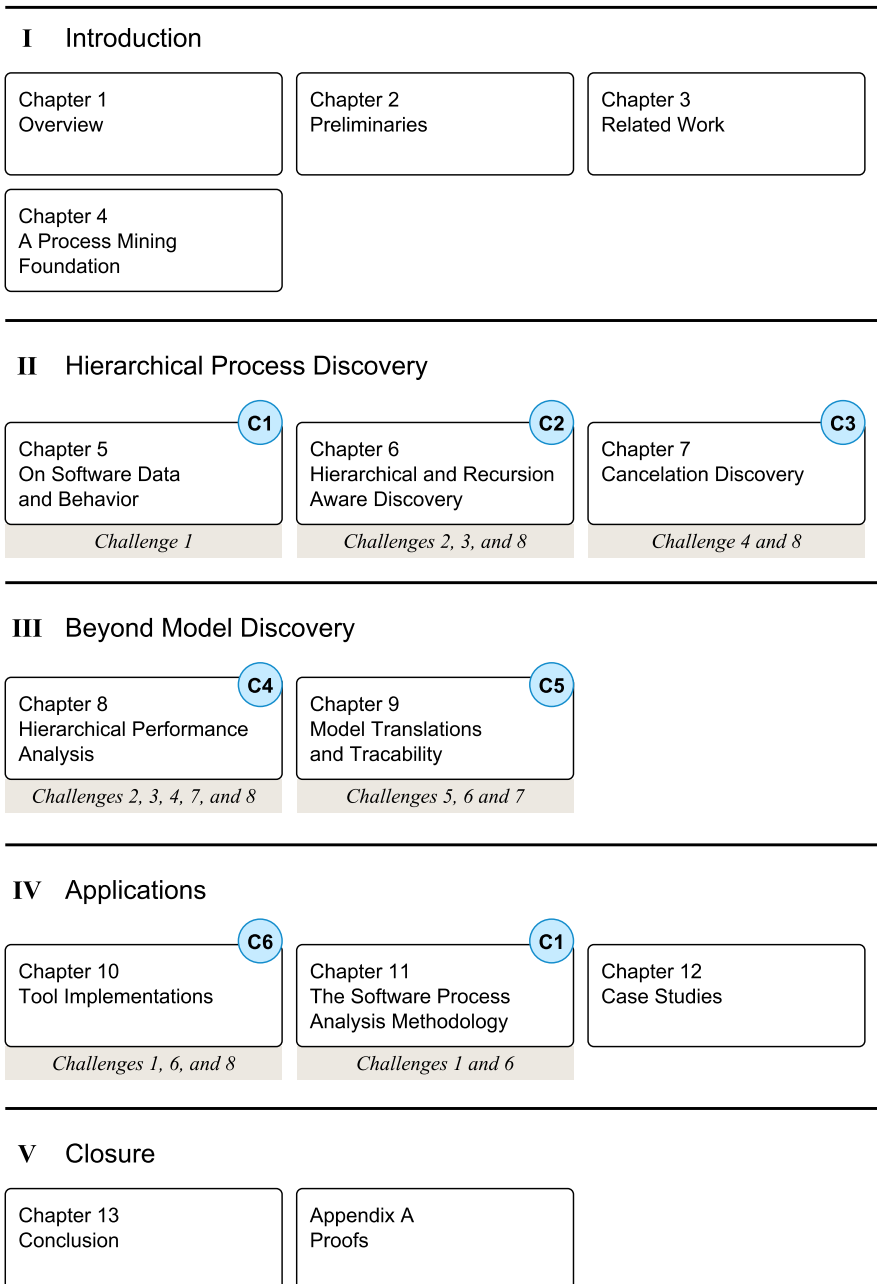
Figure 1.8 shows the structure of this thesis. Part I provides an introduction to process mining and software engineering. Chapter 2 introduces the basic concepts of process mining. Chapter 3 positions the thesis and discusses related work. Chapter 4 introduces a basic process mining foundation upon which the techniques in the rest of the thesis build.

Part II presents our novel hierarchical process discovery techniques. Chapter 5, discusses the properties, patterns, and the accompanied challenges of software event logs. Chapter 6 introduces the *hierarchical process tree* notation to model hierarchical and recursive behavior. In addition, we present several algorithms for discovering hierarchical and recursive models. Chapter 7 introduces the *cancellation process tree* notation to model cancellation regions using process trees. In addition, we present an algorithm for discovering cancellation process trees.

Part III further explores hierarchical process mining beyond model discovery. Chapter 8 introduces a hierarchical approach to performance analysis. Our hierarchical performance analysis approach allows us to analyze performance while taking into account hierarchical, recursive, and cancellation behavior. Chapter 9 provides an extensive model to model transformation framework, taking into account the hierarchical, recursive, and cancellation semantics.

Part IV discusses applications of the techniques and algorithms presented in this thesis. Chapter 10 presents the implemented tools, their interactions, and how our tools allow for integration with existing software artifacts. Chapter 11 presents a methodology for obtaining and analyzing software event log data in a structured way. Chapter 12 presents various case studies, showing how our techniques can be used in practice.

Part V concludes this thesis. Chapter 13 summarizes the main results and discusses possible research directions that build on the presented work. Appendix A presents the proof details for the various theorems and lemmas found throughout this thesis.



**Figure 1.8:** The structure of this thesis. The white boxes indicate the chapters, grouped into five parts. The blue circles above the chapters refer to the contributions and the grey boxes below the chapters refer to the challenges.



“The mark of a great man is one who knows when to set aside the important things in order to accomplish the vital ones.”

— Brandon Sanderson, *The Alloy of Law*

## 2 | Preliminaries

In this chapter, we will introduce the definitions, formalizations, and notations used throughout the rest of the thesis. We start by introducing some basic notations and terminology in Section 2.1. In Section 2.2, we discuss several process model notations used in later chapters for discovering and visualizing behavioral models. In Section 2.3 we will discuss event logs, the basic input for many process mining techniques. In Section 2.4 we will discuss the directly follows relation, an event log abstraction commonly used in process discovery.

### 2.1 Basic Notation

In this section, we start with sets, sequences, functions, and projections (Section 2.1.1), followed by some basic graph theory (Section 2.1.2).

#### 2.1.1 Sets, Sequences, and Functions

**Definition 2.1.1 — Sets and Multisets.** A set is an unordered collection of elements. We use uppercase letters to denote sets and lowercase letters to denote the elements. Given sets  $X = \{a, b\}$  and  $Y = \{b, c\}$ , we denote:

$\mathcal{P}(X)$	The <i>power set</i> over $X$ . For example: $\mathcal{P}(X) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$
$X \cup Y$	The <i>union</i> of $X$ and $Y$ . For example: $X \cup Y = \{a, b, c\}$
$X \cap Y$	The <i>intersection</i> of $X$ and $Y$ . For example: $X \cap Y = \{b\}$
$X \setminus Y$	The <i>set difference</i> , i.e., in $X$ but not in $Y$ . For example: $X \setminus Y = \{a\}$
$X \subseteq Y$	The <i>subset</i> relation, i.e., is $X$ a subset of $Y$ . For example: $\{a, b\} \subseteq \{a, b, c\}$ .
$\emptyset$	The <i>empty set</i>

A multiset is a set where each element can appear multiple times. Given multisets  $A = [a^2, b^1] = [a^2, b] \in \mathcal{B}(X)$ ,  $B = [b^1, c^1] = [b, c] \in \mathcal{B}(Y)$  and elements  $a, b, c \in X$ , we denote:

$\mathcal{B}(X)$	The set of <i>all multisets</i> of set $X$ . For example: $A \in \mathcal{B}(X) = \{[], [a^1], [b^1], [a^2, b^1], \dots\}$
$A(a)$	The <i>number of times</i> $a$ appears in multiset $A$ . For example: $A(a) = 2, A(b) = 1, A(c) = 0$
$A + B$	The <i>summation</i> of $A$ and $B$ . For example: $A + B = [a^2, b^2, c^1]$
$A - B$	The <i>difference</i> of $A$ and $B$ . For example: $A - B = [a^2]$
$A \leq B$	The <i>subbag</i> relation, i.e., is $A$ a subbag of $B$ . For example: $[a, b] \leq [a^2, b, c]$ .
$ B $	The <i>size of a multiset</i> is given by the total number of elements: $ B  = \sum_{a \in B} B(a) = 2$
$[]$	The <i>empty multiset</i>

**Definition 2.1.2 — Sequences.** A sequence is an ordered list of elements. Given a set  $X = \{a, b\}$ , a sequence over  $X$  of length  $n$  is denoted as  $\sigma = \langle a_1, \dots, a_n \rangle \in X^*$ . We denote:

$X^*$	The set of <i>all sequences</i> of $X$ . For example: $X^* = \{\varepsilon, \langle a \rangle, \langle b \rangle, \langle a, a \rangle, \langle b, a \rangle, \dots\}$
$\sigma \cdot \sigma'$	The <i>concatenation</i> of $\sigma$ and $\sigma'$ . For example: $\langle a, b \rangle \cdot \langle c, d \rangle = \langle a, b, c, d \rangle$ , and $\langle a, b \rangle \cdot \varepsilon = \langle a, b \rangle$
$\sigma \diamond \sigma'$	The <i>set of all interleavings (shuffles)</i> of $\sigma$ and $\sigma'$ . The order of elements in $\sigma$ and $\sigma'$ are respected in $\sigma \diamond \sigma'$ . For example: $\langle a, b \rangle \diamond \langle c, d \rangle = \{\langle a, b, c, d \rangle, \langle a, c, b, d \rangle, \langle a, c, d, b \rangle, \langle c, a, b, d \rangle, \langle c, a, d, b \rangle, \langle c, d, a, b \rangle\}$
$\sigma[i]$	Refer the $i$ th element: $\langle a_1, \dots, a_n \rangle [i] = a_i$
$head(\sigma)$	The <i>head (first element)</i> of $\sigma$ : $head(\langle a_1, \dots, a_n \rangle) = a_1$
$end(\sigma)$	The <i>end (last element)</i> of $\sigma$ : $end(\langle a_1, \dots, a_n \rangle) = a_n$
$\varepsilon$	The <i>empty sequence</i>

**Definition 2.1.3 — Set and Sequence Comprehension.** Set comprehension is a notation for describing a set by stating the properties that its elements must satisfy. For example, consider:

$$X = \{2 \cdot x \mid x \in \mathbb{N} \wedge x > 3\}$$

Here,  $X$  denotes the set containing all the numbers “2 times  $x$ ” such that  $x$  is an element of the natural numbers ( $\mathbb{N}$ ) and  $x$  is greater than 3.

The same idea can be extended to multisets and sequences as well, e.g.:

$$\sigma = \langle x \mid x \in \langle 1, 2, 3, 4 \rangle \wedge “x \text{ is even}” \rangle$$

Here,  $\sigma$  denotes the ordered sequence containing all the even numbers in the sequence  $\langle 1, 2, 3, 4 \rangle$  (the order is maintained). Hence, this sequence comprehension yields the sequence  $\sigma = \langle 2, 4 \rangle$ . Note that  $\langle \dots \mid \text{false} \rangle = \varepsilon$ .

**Definition 2.1.4 — Functions.** A function is a relation from a set  $X$  to a set  $Y$ , where every element in  $X$  is associated with an element of  $Y$ . Given set  $X$  and  $Y$ , we denote:

$f : X \mapsto Y$  The *function*  $f$  with domain type  $X$  and range type  $Y$

$dom(f)$  The *domain* of  $f$ :  $dom(f) = X$

$rng(f)$  The *range* of  $f$ :  $rng(f) = \{ f(x) \mid x \in X \} \subseteq Y$

A function  $f : X \mapsto Y$  can be applied to sequences over  $X$ , yielding sequences over  $Y$ . We define sequence function application, with  $\sigma \in X^*$ :

$$f(\sigma) = \langle f(x) \mid x \in \sigma \wedge x \in X \rangle$$

**Definition 2.1.5 — Projections.** A projection restricts or filters a function, multiset, or sequence to a given domain  $Z$ . We define projections as follows:

Projection  $f \upharpoonright_Z$  restricts function  $f$  to domain  $Z$ , with  $x \in dom(f)$ :

$$f \upharpoonright_Z(x) = f(x) \text{ for } x \in dom(f \upharpoonright_Z) = dom(f) \cap Z$$

Projection  $A \upharpoonright_Z$  restricts multiset  $A \in \mathcal{B}(X)$  to domain  $Z$ , with  $a \in X$ :

$$A \upharpoonright_Z(a) = \begin{cases} 0 & \text{if } a \notin Z \\ A(a) & \text{if } a \in Z \end{cases}$$

Projection  $\sigma \upharpoonright_Z$  restricts sequence  $\sigma \in Y^*$  to domain  $Z$ , with  $y \in Y$ : We define sequence function application, with  $\sigma \in X^*$ :

$$\sigma \upharpoonright_Z = \langle x \mid x \in \sigma \wedge x \in Z \rangle$$

For example, we have:

$$[a^2, b, c^3, d] \upharpoonright_{\{b,c\}} = [b, c^3]$$

$$\langle a, b, c, d, e, c, f \rangle \upharpoonright_{\{b,c\}} = \langle b, c, c \rangle$$

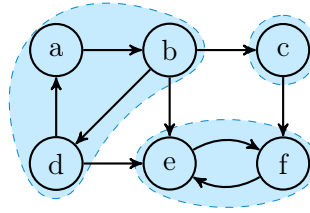
$$\langle a, b, c, d, e, c, f \rangle \upharpoonright_{\{a,d,e,f\}} = \langle a, d, e, f \rangle$$

### 2.1.2 Graph theory

A *graph* is a set of *nodes* combined with a set of *edges*, such that each edge connects two nodes. Edges and nodes can be labeled and annotated. For example, a node might have a name, and an edge can have a weight. If the edges of a graph have no direction, the graph is an *undirected graph*. If the edges do have a direction, the graph is a *directed graph*. The start of a directed edge is called the *source*, and the end is called the *target*. A directed graph can be projected onto an undirected graph by ignoring the direction of the edges. Figure 2.1 shows an example directed graph with 6 nodes and 9 edges.

We write a graph  $G$  with set of nodes  $V$  and set of edges  $E$  as  $G = (V, E)$ . We write  $(a, b) \in G$  to denote the existence of edge  $(a, b) \in E$ .





**Figure 2.1:** Example graph with strongly connected components indicated. The solid circles are the nodes and the arrows the edges. The dashed regions indicate the strongly connected components.

**Definition 2.1.6 — Path in a Graph.** A (directed) *path*, notation  $\rightsquigarrow$ , in a graph  $G$  is a non-empty sequence of nodes such that for all sequential pair of nodes there exists a corresponding edge (in the connected direction).

For example, in Figure 2.1, there exists a path between nodes  $a$  and  $c$  ( $a \rightsquigarrow c \in G$ ), but no path between  $f$  and  $b$  ( $a \rightsquigarrow c \notin G$ ) since the edges are connected in the wrong direction.

**Definition 2.1.7 — (Strongly) Connected Components.** A *connected component* in an undirected graph is a non-empty set of nodes such that for each pair of nodes  $u, v$  in the set, there is a path in the graph between  $u$  and  $v$  (in either direction).

A *strongly connected component* in a directed graph is a non-empty set of nodes such that for each pair of nodes  $u, v$  in the set, there is both a directed path from  $u$  to  $v$  and a directed path from  $v$  to  $u$  in the graph.

For example, the graph in Figure 2.1 consists of a single connected component and has 3 strongly connected components, which are indicated by the dashed regions.

**Definition 2.1.8 — Graph Cuts.** A  $n$ -ary *cut* is a partition of the nodes  $V$  of a graph  $G = (V, E)$  into pairwise disjoint non-empty sets  $\Sigma_1, \dots, \Sigma_n$ . A graph cut is a *non-trivial cut* if  $n > 1$  and no partition  $\Sigma_i$  is empty.

For example, the strongly connected components in Figure 2.1 describe a valid non-trivial cut yielding 3 partitions:  $\{a, b, d\}$ ,  $\{c\}$  and  $\{e, f\}$ .

## 2.2 Process Model Notations

Process models capture the behavior of a process. A plethora of modeling notations exist, some tailored to business processes, and some tailored to software processes. All these models describe, in different degrees of formality, if and in which order activities are to be executed. That is, process models describe a language in terms of activities. In this context, an activity is a well-defined

step in the process such as a user task, an external service, a software method or statement, etc. Process models can usually be represented in terms of a graph, but more complex notations also exist.

In this section, we describe process model notations from both the business process context and the software engineering context. We consider the following notations: Petri nets (Section 2.2.1), YAWL (Section 2.2.2), BPMN (Section 2.2.3), Statecharts (Section 2.2.4), Message Sequence Diagrams (Section 2.2.5), and process trees (Section 2.2.6).

### 2.2.1 Petri nets

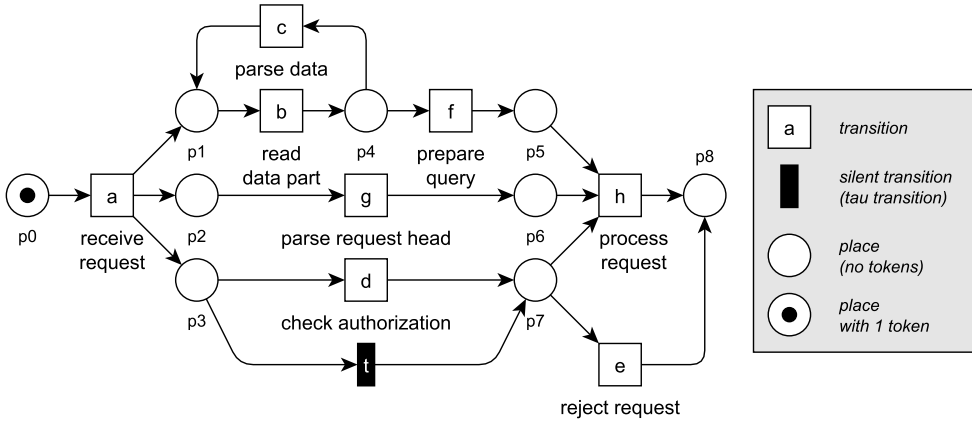
Petri nets are the oldest and best-investigated process modeling language that naturally supports concurrency [146, 163]. Although the graphical representation is intuitive and relatively simple, Petri nets are executable, and many analysis techniques can be used to analyze them. In this section, we introduce the basics of Petri nets and some subclasses and extensions to this formal notation.

#### Basic Petri net Notations and Definitions

A Petri net is a bipartite graph consisting of *places* and *transitions*, connected by *arcs*. An example of a Petri net is given in Figure 2.2. Circles represent places, and squares represent transitions. Arcs are arrows connecting places and transitions in a bipartite manner. Transitions can represent tasks or activities. A silent activity or transition, also called a  $\tau$ -transition, is represented by a filled black transition.

The state of a Petri net is determined by the distribution of *tokens* over places and is referred to as its *marking*. Black dots represent the tokens and arcs indicate the input and output places. When a transition is executed, it consumes one *token* from each input place, and produces a token in each of their output places. The start of the process is indicated by the *initial marking*, and the end or termination of the process is indicated by a *final marking*.

In the example Petri net from Figure 2.2, the initial marking is given by [p0] (i.e., it starts with one token in the place *p0*). The *firing rule* defines the dynamic behavior of this marked Petri net. A transition is *enabled* if each of its input places contains a token. An enabled transition can *fire* if it is enabled, thereby consuming one token from each input place, and producing one token for each output place. In the initial marking in Figure 2.2, the only transition that is enabled is *a*. After firing transition *a*, the token from place *p1* is consumed, and a token is placed in each of the three output places (*p1*, *p2*, and *p3*), enabling the three parallel branches. In this new marking, only transitions *b*, *g*, *d* and *t* (the black  $\tau$ -transition) are enabled. Note that transition *d* can be skipped via the  $\tau$ -transition. The final marking [p8] can be reached by firing, for example, the sequence *a*, *b*, *g*, *d*, *f*, *h*. Because of the



**Figure 2.2:** Example Petri net for a web service provider process. The initial marking is  $[p_0]$ , and the final marking is  $[p_8]$ . Note that this model is unsound, e.g., consider the firing sequence  $\langle a, b, f, g, d, e \rangle$  ending in the marking  $[p_5, p_6, p_8]$ . Possible sound variants are given in Figures 2.5 and 2.7.

loop construct involving  $c$  and  $b$ , there are infinitely many firing sequences. Observe that the sequence  $a, b, f, g, d, e$  results in the marking  $[p_5, p_6, p_8]$ , and transition  $h$  is not enabled in this marking. Hence, from this marking, we can no longer reach the final marking  $[p_8]$ . As we will explain below, this is an example of a (problematic) *unsound Petri net*. A solution could be, for example, to remove transition  $e$ .

To formalize Petri nets, we introduce some definitions and notations.

**Definition 2.2.1 — Petri net.** Let  $\mathbb{A}$  denote the activity alphabet and let  $\tau$  denote a special (silent) label such that  $\tau \notin \mathbb{A}$ . A Petri net is a tuple  $PN = (P, T, F, \ell)$ , where:

- $P$  is a finite set of places
- $T$  is a finite set of transitions, such that  $P \cap T = \emptyset$
- $F$  is a finite multiset of directed arcs:  $F \in \mathcal{B}((P \times T) \cup (T \times P))$
- $\ell$  is a transition labeling function:  $\ell : T \mapsto \mathbb{A} \cup \{\tau\}$

**Definition 2.2.2 — Pre set and Post set.** Let  $PN = (P, T, F, \ell)$  be a Petri net. For any  $x \in (P \cup T)$ , we denote:

- $\bullet x$  The *pre (multi)set* or input nodes of  $x$ :  $\bullet x = [y \mid (y, x) \in F]$
- $x \bullet$  The *post (multi)set* or output nodes of  $x$ :  $x \bullet = [y \mid (x, y) \in F]$

The state of a Petri net is determined by the distribution of *tokens* over places and is referred to as its *marking*. The firing rule defines the execution semantics of a Petri net and captures how tokens are consumed and produced.

**Definition 2.2.3 — Marking, Enabled, and the Firing Rule.** Given a Petri net  $PN = (P, T, F, \ell)$ , a marking  $M$  is a multiset of places, i.e.,  $M \in \mathcal{B}(P)$ . A transition  $t \in T$  is *enabled* in a marking  $M$ , denoted  $(PN, M)[t]$ , if and only if  $\bullet t \leq M$ , i.e., if in marking  $M$  there are enough tokens on all the input places of  $t$ .

*Firing* transition  $t$  in marking  $M$  results in a new marking  $M' = (M - \bullet t) + t\bullet$ , denoted  $M[t]M'$ , i.e., tokens are removed from the set of input places  $\bullet t$  and added to the set of output places  $t\bullet$ . The *firing rule*  $_{[_]_}$  is the smallest relation such that for any Petri net  $PN$ , any marking  $M \in \mathcal{B}(P)$  and any  $t \in T$  with  $\bullet t \leq M$ , we have:  $(PN, M)[t](PN, (M - \bullet t) + t\bullet)$ .

For a given Petri net in a given state or marking, we can now define which markings are reachable, and what sequences of transitions can be fired.

**Definition 2.2.4 — Firing Sequences and Reachability.** Let  $PN = (P, T, F, \ell)$  be a Petri net and let  $M$  be a marking in  $PN$ , i.e.,  $M \in \mathcal{B}(P)$ .

A sequence  $\sigma \in T^*$  is called a *firing sequence* of  $(PN, M)$ , if and only if it is a sequence of enabled transitions starting in  $M$ . More formally, assuming  $M = M_0$ , sequence  $\sigma \in T^*$  is a firing sequence iff, for some  $n \in \mathbb{N}$ , there exist markings  $M_1, \dots, M_n$  and transitions  $t_1, \dots, t_n \in T$  such that  $\sigma = \langle t_1, \dots, t_n \rangle$ , and for all  $i$  with  $0 \leq i < n$ , we have both:

- $(PN, M_i)[t_{i+1}]$ , i.e.,  $t_{i+1}$  is enabled in  $M_i$ ; and
- $(PN, M_i)[t_{i+1}](PN, M_{i+1})$ , i.e., firing  $t_{i+1}$  from  $M_i$  results in the new marking  $M_{i+1}$ .

A marking  $M'$  is *reachable* in  $PN$  from the marking  $M$ , notation  $M' \in [PN, M]$ , if and only if there exists a sequence of enabled transitions whose firing leads from  $M$  to  $M'$ , i.e., iff there is a sequence of firings  $\sigma = \langle t_1, t_2, \dots, t_n \rangle$  that transforms  $M$  into  $M'$ , denoted  $M[\sigma]M'$ .

The start of a process is indicated by an *initial marking* and the end or termination by a *final marking*. We capture a Petri net plus its initial and final marking in a *system net*. Using these initial and final markings, we define the language of a system net by capturing all full firing sequences.

**Definition 2.2.5 — System net.** A *system net* [1, 14] is a tuple  $SN = (PN, M_{ini}, M_{fin})$  where:

- $PN$  is a Petri net
- $M_{ini}$  is the initial marking:  $M_{ini} \in \mathcal{B}(P)$
- $M_{fin}$  is the final marking:  $M_{fin} \in \mathcal{B}(P)$

Using the firing rule, we can now define which sequence of transitions can fire in a system net. Moreover, we can capture all the reachable markings, and define the language of a system net.

**Definition 2.2.6 — Full Firing Sequences and Language.** Let  $SN = (P, T, F, M_{ini}, M_{fin}, \ell)$  be a system net.

A sequence  $\sigma \in T^*$  is called a *full firing sequence* of  $(SN, M_{ini})$ , if it leads from  $M_{ini}$  to  $M_{fin}$ , notation:  $M_{ini}[\sigma]M_{fin}$ . The set of full firing sequences, i.e., the set of sequences of enabled transitions leading from  $M_{ini}$  to  $M_{fin}$ , is denoted as:  $S_{SN} = \{ \sigma \in T^* \mid M_{ini}[\sigma]M_{fin} \}$

The *language* of a system net  $SN$ , notation  $\mathcal{L}(SN)$ , is given by the set of all full firing sequence labels:  $\mathcal{L}(SN) = \{ \ell(\sigma) \mid \sigma \in S_{SN} \wedge \ell(\sigma) \neq \tau \}$ .

In the Petri net of Figure 2.2 with  $M_{ini} = [p0]$ , the marking  $M_{fin} = [p8]$  is reachable through, for example, the firing sequences  $\langle a, b, g, d, f, h \rangle$  and  $\langle a, b, f, g, t, h \rangle$ . The language trace corresponding to the firing sequence  $\langle a, b, f, g, t, h \rangle$  equals  $\langle \text{receive request, read data part, prepare query, parse request head, process request} \rangle$ . Note that  $\tau$ -transition  $t$  has no label. The marking  $[p5, p6, p8]$  is reachable through, for example, the firing sequence  $\langle a, b, f, g, d, e \rangle$ . However, the marking  $[p7, p8]$  is not reachable from  $M_{ini}$ .

### Workflow Nets, Soundness, and Block-structured Workflow Nets

*Workflow nets* (WF-nets) are a subclass of system nets that have a dedicated source place where the process starts, and a dedicated sink place where the process ends [8]. Moreover, all nodes in a WF-net are on a path from source to sink. The subclass of workflow nets is a natural representation for process models that describe the *lifecycle of a case*.

**Definition 2.2.7 — Workflow net.** A *workflow net* (WF-net) [1, 14] is a system net  $SN = (P, T, F, M_{ini}, M_{fin}, \ell)$  such that:

- There is a single source place  $i \in P$  such that:  $\bullet i = \emptyset$
- There is a single sink place  $o \in P$  such that:  $o \bullet = \emptyset$
- All places and transitions are on a path from  $i$  to  $o$
- The source is the initial marking:  $M_{ini} = [i]$
- The sink is the final marking:  $M_{fin} = [o]$

Observe that the Petri net in Figure 2.2 is a workflow net. However, as noted before, there is a problem with this Petri net: the firing sequence  $\langle a, b, f, g, d, e \rangle$  results in the marking  $[p5, p6, p8]$ , and the final marking is no longer reachable. The Petri net in Figure 2.2 is an example of a (problematic) *unsound Petri net*. We call a workflow net *sound* if each transition can fire from the initial marking, and from each reachable marking, it is possible to reach the final marking [8]. More formally:

**Definition 2.2.8 — Soundness.** Let  $SN = (P, T, F, [i], [o], \ell)$  be a workflow net with source place  $i$  and sink place  $o$ . The WF-net  $SN$  is *sound* [1, 14] if and only if:

- *Safeness*: Places cannot hold multiple tokens at the same time:  
 $\forall M' \in [SN, [i]], p \in P : M'(p) \leq 1$
- *Absence of dead parts*: Every transition can be enabled:  
 $\forall t \in T : \exists M' \in [SN, [i]] : (SN, M')[t]$
- *Option to complete*: For any marking reachable from the initial marking, it is possible to reach the final marking:  
 $\forall M' \in [SN, [i]] : [o] \in [SN, M']$
- *Proper completion*: Every marking reachable from the initial marking with a token in  $o$  has no tokens left behind:  
 $\forall M' \in [SN, [i]] : o \in M' \Rightarrow M' = [o]$

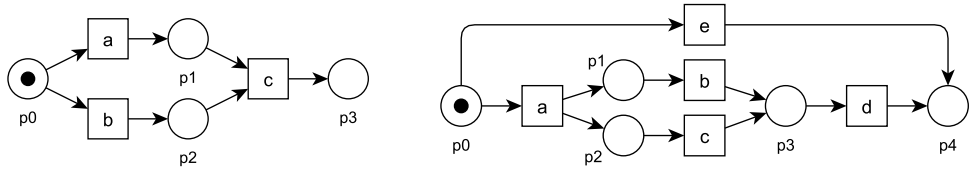
Note that the *option to complete* implies *proper completion*.

Soundness is an important property, and unsound models are undesirable. In an unsound model, some process steps might be inexecutable, one might end up in a deadlock, or there might be tokens remaining after a token reaches the sink place. In extreme cases, an unsound model might not even be able to complete, i.e., the final marking might not be reachable at all. Furthermore, when (automatically) analyzing unsound models, the analysis results can be unreliable and counter-intuitive.

There exist various other kinds of soundness [14]. For example, a weaker notion of soundness is *weak soundness*: a workflow net is weakly sound if there is at least one way to reach the final marking from the initial marking. Another example is *relaxed soundness*: a workflow net is relaxed sound if for every transition  $t$  there is at least one trace from the initial to the final marking. A sound workflow net is by definition also weak and relaxed sound.

Figure 2.3a shows an example of a workflow net in which the final marking is not reachable. After firing either  $a$  or  $b$ , the net is in a deadlock:  $c$  would require both  $a$  and  $b$  to be fired. Figure 2.3b shows an example of a workflow net in which there can be remaining tokens after the sink place is reached. After firing  $a$ ,  $b$ , and  $c$ , transition  $d$  can fire: a token is placed in the sink place, but there is a remaining token in the input place of  $d$ .

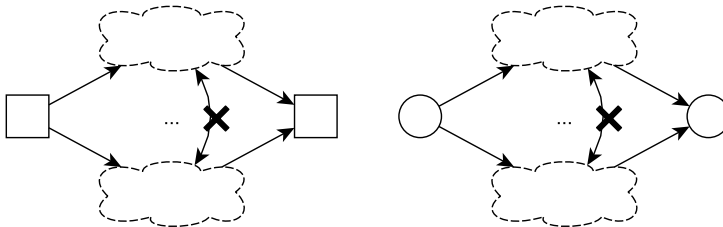
A subclass of workflow nets that are guaranteed to be sound is the *block-structured workflow net* [130]. A workflow net is *block-structured* if, for every place or transition with multiple outgoing arcs, there exists a corresponding place or transition with multiple incoming arcs, respectively. Moreover, the parts of the net between the outgoing and incoming arcs form regions, and no arcs may exist between these regions. That is, each region has a single entry and a single exit. Figure 2.4 demonstrates this single-entry-single-exit property. Note that dashed regions can be bound by either transitions or places and there are no arcs between the regions. The sound workflow net in Figure 2.5 is block structured. The filled regions denote the blocks of the block-structure, ensuring soundness.



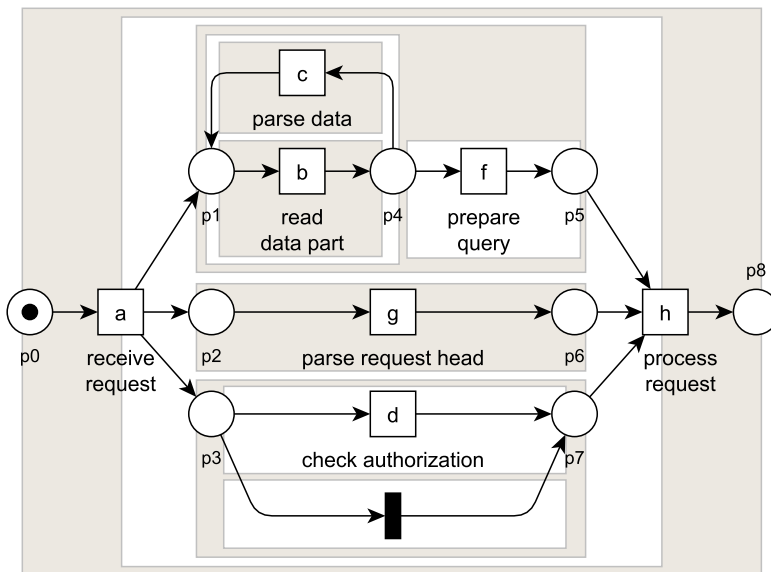
(a) A workflow net with an unreachable final marking.  $M_{ini} = [p_0]$  and  $M_{fin} = [p_3]$ .

(b) A workflow net with remaining tokens.  $M_{ini} = [p_0]$  and  $M_{fin} = [p_4]$ .

**Figure 2.3:** Two examples of unsound workflow nets. See also Definition 2.2.8.



**Figure 2.4:** Single-entry-single-exit regions. Note that dashed regions can be bound by either transitions or places and there are no arcs between the regions.



**Figure 2.5:** Example of a block-structured workflow net with initial marking  $M_{ini} = [p_0]$  and final marking  $M_{fin} = [p_8]$ . This net is a sound variant of the example Petri net in Figure 2.2, obtained by removing transition  $e$ . The filled regions denote the blocks of the block-structure.

### Reset arcs, Cancellation Regions, and Inhibitor arcs

Several extensions to Petri nets have been proposed to increase their expressivity. In this section, we briefly discuss two: *reset arcs* and *inhibitor arcs*.

A *reset arc* between a place and a transition removes all tokens from that place when the transition is fired. However, the place does not have to contain tokens for the transition to be enabled. That is, the reset arc does not influence the precondition of the transition. Reset arcs are typically used to ‘clean up’ tokens after a particular part of a process is canceled. As syntactic sugar, one can associate a *cancellation region* with a transition  $t$  to denote the set of places that have reset arcs to transition  $t$ . In Figure 2.6a two reset arcs are shown, and in Figure 2.6b the same behavior is modeled using a cancellation region. Figure 2.7 demonstrates how a cancellation region can be used to fix the soundness issue in Figure 2.2. For example, the firing sequence  $\langle a, b, f, g, d, e \rangle$  now results in the marking  $[p8]$ ; the tokens in  $p5$  and  $p6$  are removed after firing  $e$  due to the cancellation region.

An *inhibitor arc* between a place and a transition models the requirement for an empty place. That is, a transition with inhibitor arcs is only enabled when all the places indicated by the inhibitor arcs are empty/contain no tokens. In Figure 2.6c an inhibitor arc is shown.

A reset arc can always be transformed into an inhibitor arc, but the reverse is not true [8]. Note that inhibitor arcs makes Petri nets Turing complete [156].

**Definition 2.2.9 — Reset/Inhibitor net.** A Reset/Inhibitor net is a tuple  $PN = (P, T, F, R, I, \ell)$ , where:

- $P$  is a finite set of places
- $T$  is a finite set of transitions, such that  $P \cap T = \emptyset$
- $F$  is a finite multiset of directed arcs:  $F \in \mathcal{B}((P \times T) \cup (T \times P))$
- $R$  is a finite multiset of reset arcs:  $R : T \mapsto \mathcal{P}(P)$
- $I$  is a finite multiset of inhibitor arcs:  $I : T \mapsto \mathcal{P}(P)$
- $\ell$  is a transition labeling function:  $\ell : T \mapsto \mathbb{A} \cup \{\tau\}$

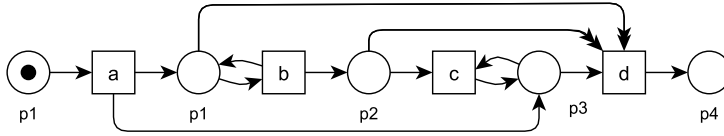
**Definition 2.2.10 — Enabled and the Firing Rule in a Reset/Inhibitor net.**

Let  $PN = (P, T, F, R, I, \ell)$  be a Reset/Inhibitor net and let  $M$  be a marking in  $PN$ , i.e.,  $M \in \mathcal{B}(P)$ .

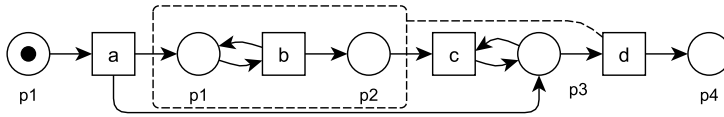
A transition  $t \in T$  is *enabled* in a marking  $M$ , denoted  $(PN, M)[t]$ , if and only if  $\bullet t \leq M$  and  $|M \upharpoonright_{I(t)}| = 0$ , i.e., if in marking  $M$  there are enough tokens on all the input places of  $t$  and there are no tokens on any of the inhibited places  $I(t)$ .

*Firing* transition  $t$  in marking  $M$  results in a new marking  $M' = (M - \bullet t) \upharpoonright_{(P \setminus R(t))} + t \bullet$ , denoted  $M[t]M'$ , i.e., tokens are removed from the set of input places  $\bullet t$ , any remaining tokens on the reset places  $R(t)$  are removed, and tokens are added to the set of output places  $t \bullet$ .

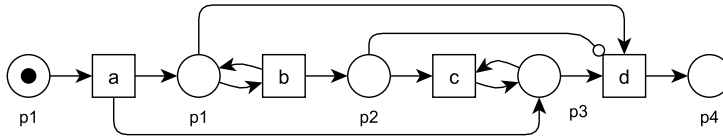




(a) A Petri net with two *reset arcs*, modeled as double headed arrows.

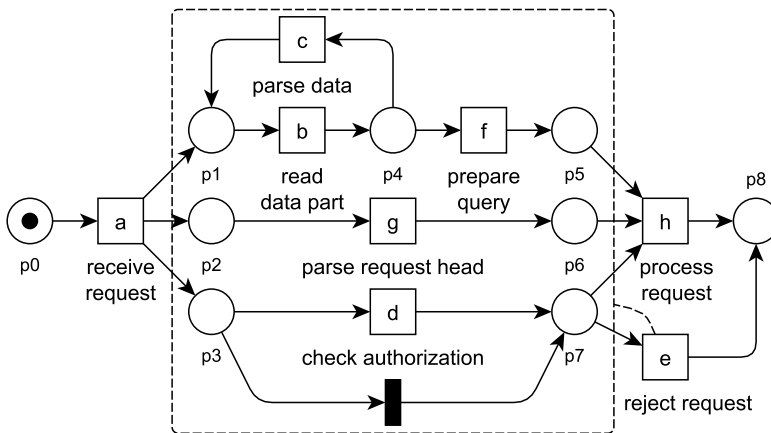


(b) The Petri net from Figure 2.6a modeled with a *cancellation region*, depicted as a dashed region.



(c) A Petri net with an *inhibitor arc*, modeled as an arrow with open circle.

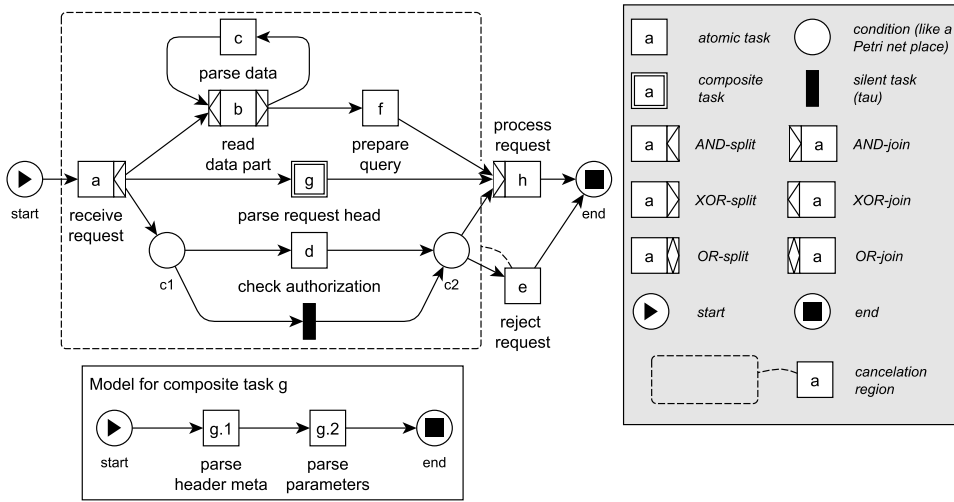
**Figure 2.6:** Examples of counting Petri nets using the reset and inhibitor extensions. The language of these nets consists of traces that start with an  $a$ , followed by  $b$ 's and  $c$ 's, and ending with a  $d$ . In the reset nets (Figures 2.6a and 2.6b), the number of  $c$ 's never exceeds the number of  $b$ 's. In the inhibitor net (Figure 2.6c), the number of  $c$ 's equals the number of  $b$ 's. Both constructs cannot be expressed in a classical Petri net.



**Figure 2.7:** Example of a reset workflow net with a cancellation region and with initial marking  $M_{ini} = [p0]$  and final marking  $M_{fin} = [p8]$ . This net is a sound variant of the example Petri net in Figure 2.2, obtained by associating a cancellation region with transition  $e$ .

### 2.2.2 YAWL – Yet Another Workflow Language

*YAWL* is both a workflow modeling language and an open-source workflow system [8, 86]. The acronym *YAWL* stands for “Yet Another Workflow Language”. The language was designed to provide an illustrative implementation of an important subset of the typical constructs to represent how cases flow through a process: the *workflow patterns* [15]. These workflow patterns were identified based on a systematic analysis. They cover, amongst others, control-flow, data, resource and exception patterns.



**Figure 2.8:** Example YAWL model for a loan application process. The start and end conditions are labeled as such. The main YAWL model, excluding the submodel for *g*, has the same language as the reset net model in Figure 2.7. Note that the cancellation region must include the arcs, which represent the ‘hidden places’ from the model in Figure 2.7.

The YAWL language extends Petri nets using syntactic sugaring and several constructs that increase the expressibility. Here we restrict ourselves to the control-flow perspective. An example of a YAWL model is given in Figure 2.8. Activities in YAWL are called *tasks*. Places are called *conditions* and are optional, i.e., tasks may be connected directly. Each task can be decorated with well-defined split and join semantics. The *AND-join/AND-split* tasks behave like a Petri net transition: it needs to consume one token via each input arc and produces a token via each output arc. In contrast, an *XOR-split* selects exactly one of its outgoing arcs, and an *XOR-join* is enabled once for every incoming token, without synchronization. The *OR-split* selects one or more of its outgoing arcs, i.e., it selects a subset of all outgoing arcs. The *OR-join* requires at least one input token, but also synchronizes tokens that

are “on their way”. That is, as long as another token may arrive, the OR-join waits. Furthermore, YAWL offers natural support for *cancellation regions*. The semantics of a YAWL cancellation region is exactly the same as the Petri net reset arc semantics mentioned before.

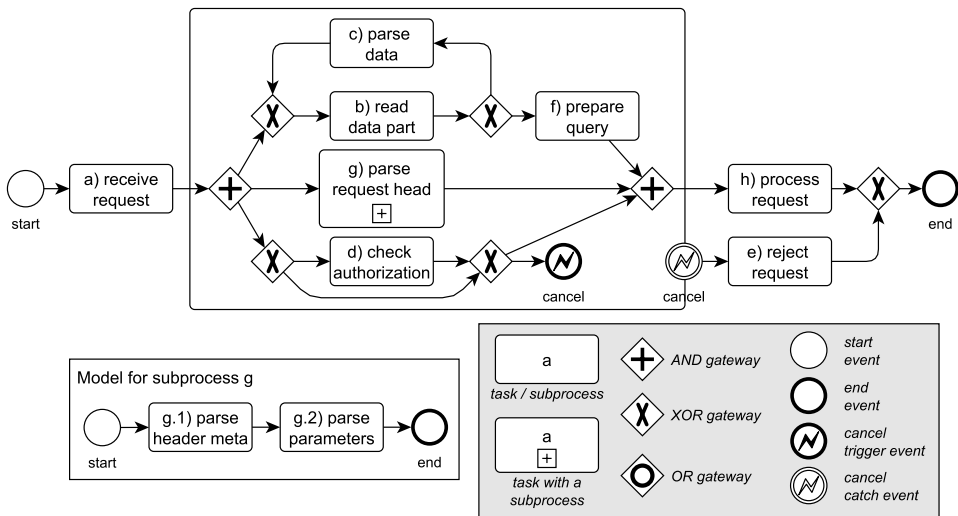
It should be noted the OR-join has non-local semantics: the decision to wait or not does not only depend on its direct predecessors but also on parts of the model that may lead to future triggers. Due to these non-local semantics, it is possible to create a paradox known as “the vicious circle” where we do not know whether the OR-joins should propagate or not [12, 98, 199].

Tasks in a YAWL model are either *atomic* or *composite*. A composite task refers to another YAWL model, and hence supports a notion of hierarchy. Via a composite task, a complex activity can be modeled in more detailed steps via a referenced subprocess. The main YAWL model in Figure 2.8, excluding the submodel for  $g$ , has the same language as the reset net model in Figure 2.7.

Note that YAWL models can have similar soundness issues as Petri nets. By mixing the AND/XOR/OR splits and joins, unsound models can be obtained.

### 2.2.3 BPMN – Business Process Modeling Notation

The *Business Process Modeling Notation* (BPMN) has become one of the most widely used languages to model business processes. BPMN is supported by many tool vendors and has been standardized by the OMG [8, 153].



**Figure 2.9:** Example BPMN model for a loan application process. The start and end events are labeled as such. This BPMN model has the same language as the YAWL model in Figure 2.8, including the submodel for  $g$ . Note that the cancellation region is modeled via a subprocess and cancel events.

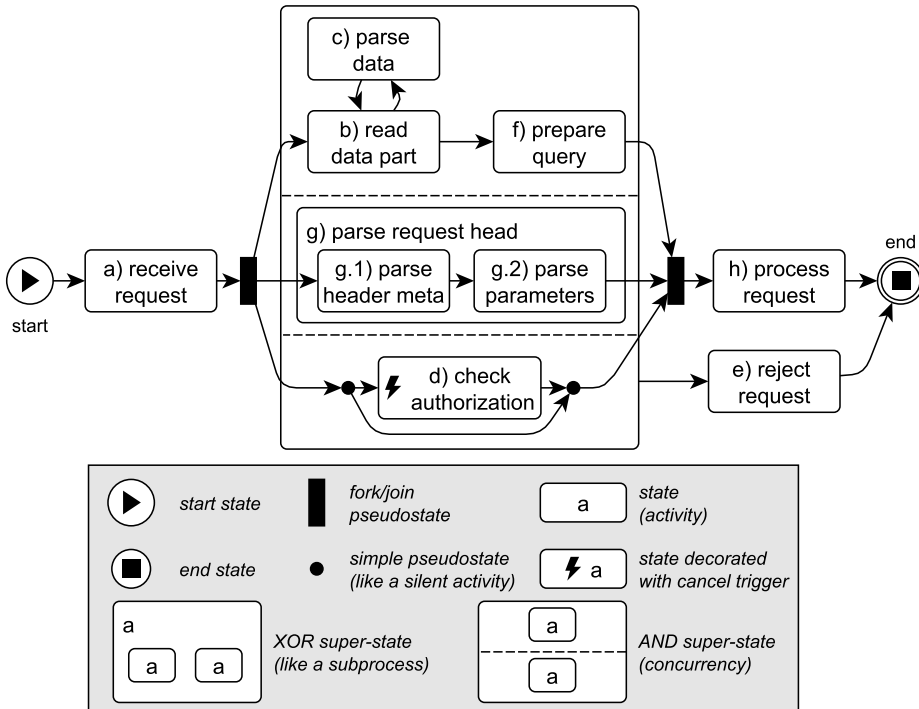
A BPMN model is similar to a Petri net or YAWL model, and most constructs can easily be understood in the same way. One notable difference is that the routing logic is not associated with tasks (as in YAWL), but with separate *gateways*. Like YAWL, there are routing constructs for AND, XOR, and OR. An example of a BPMN model is given in Figure 2.9. Atomic activities are called *tasks*. The notion of composite tasks is modeled via a *subprocess* task. A BPMN *event* is comparable to a transition in a Petri net with one incoming and one outgoing arc. There are two types of events: *catching events* and *throwing events*. Catching events wait for some trigger, for example, a timeout, an external message, or canceling the current task or subprocess. Throwing events are actively triggered during the process, for example, sending a signal or message to some catch event, or causing a cancelation. Observe that we can represent a cancelation region in BPMN through the use of a subprocess and cancel events. The BPMN model in Figure 2.9 has the same language as the YAWL model in Figure 2.8, including the submodel for *g*.

Note that BPMN models can have similar soundness issues as Petri nets. By mixing the AND/XOR/OR gateways, unsound models can be obtained.

#### 2.2.4 Statecharts

Statecharts were first introduced as a visual formalism by David Harel in 1987 [80]. The visual formalism was designed for specifying the behavior of complex reactive systems in the context of software and systems engineering. Statecharts extend classical state-transition diagrams with hierarchy, concurrency, and communication. Here, we will restrict ourselves to hierarchy and concurrency only. Later, Statecharts were incorporated in the Unified Modeling Language standard (UML) as StateMachines, adding an object-oriented extension [154]. Since the conception of Statecharts, there have been several works proposing associated formal semantics [81, 82, 83, 105, 184].

An example of a Statechart model is given in Figure 2.10. In Statecharts, activities are represented by *states*. Arcs called *transitions* connect states. Outgoing and incoming arcs follow the XOR semantics. States may be grouped in a *super-state*, creating a hierarchy of states. Super-states come in two flavors: *AND super-states* and *XOR super-states*. The XOR super-states are similar to YAWL composite tasks and BPMN subprocesses: they model a refinement of a particular state into smaller steps. The AND super-state models concurrency by capturing the orthogonal product of several XOR super-states, also known as *regions* in UML terminology. In Figure 2.10, the center AND super-state has three regions, separated by dashed lines: 1) the region containing states *b*, *c*, and *f*, 2) the region containing the *g* XOR super-state, and 3) the region containing the state *d*. For the sake of clarity, UML introduced *fork* and *join pseudo-states* to model the beginning and end of a concurrent control-flow.



**Figure 2.10:** Example Statechart model for a loan application process. The start and end states are labeled as such. This Statechart model has the same language as the YAWL model in Figure 2.8, including the submodel for *g*. Note that the cancellation region is modeled via a super-state arc.

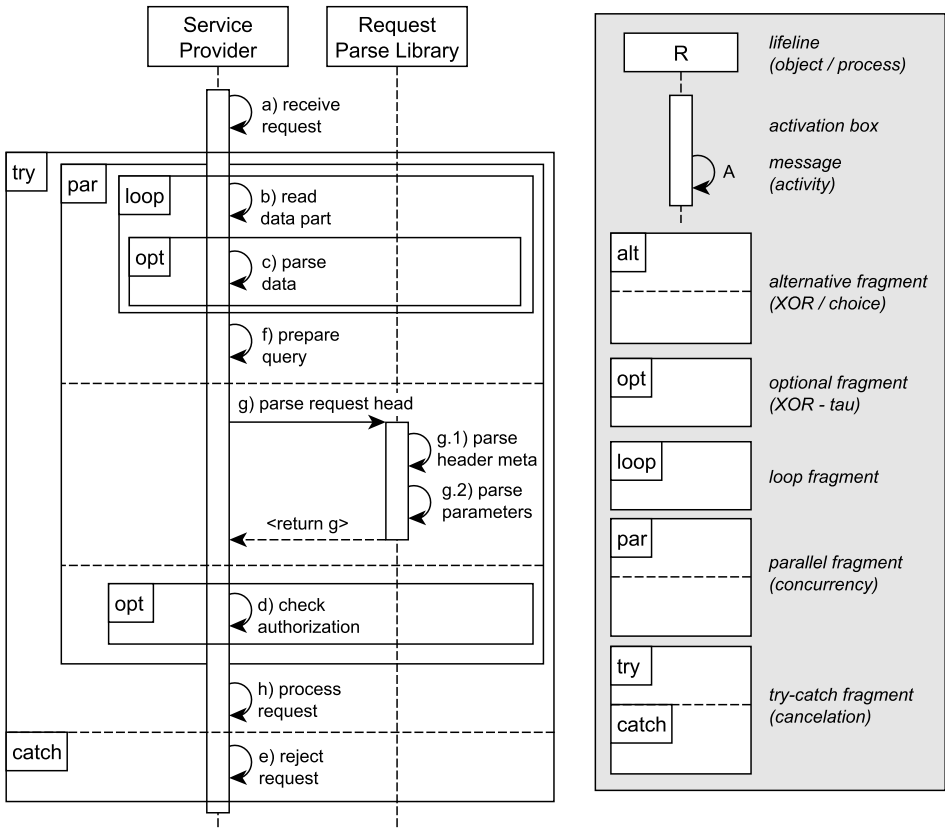
In the Statecharts language, one can also model an arc from a super-state. Such a super-state arc is short for an arc from every underlying sub-state. Note that such a super-state arc can be used to model a cancellation region. The Statechart model in Figure 2.10 has the same language as the YAWL model in Figure 2.8, including the submodel for *g*.

Note that, in Statecharts, both states and transitions can be labeled. The label on a transition is called an *event*; transitions are assumed to be instantaneous. In contrast, non-pseudo states are assumed to be non-instantaneous. In a process context, activities represent some amount of work being executed; they are not instantaneous. Hence, we chose to map activities to states, and not to events.

### 2.2.5 MSD – Message Sequence Diagrams

*Message Sequence Diagrams* (MSDs) are a popular visual formalism for describing scenarios in terms of partial orderings on events. Early standards

for MSDs appeared as Message Sequence Charts (MSCs) in a recommendation of the ITU [201]. This ITU standard defines the syntax of MSCs and is accompanied by formal semantics in terms of process algebra [202]. Other efforts at formal semantics have been made in terms of Petri nets and other formalisms [36, 74, 104]. Later, Message Sequence Diagrams were incorporated in the Unified Modeling Language standard (UML) [154].



**Figure 2.11:** Example Message Sequence Diagram (MSD) for a loan application process. The first message (seen from the top) is the start. This MSD has almost the same language as the YAWL model in Figure 2.8, including the submodel for g. The only difference is that the loop retry (activity c) could not be modeled exactly, and can also occur at the end of the loop in this MSD. The cancellation region is modeled via a try-catch fragment.

A Message Sequence Diagram shows how concurrent objects or processes interact over time. These object or processes are depicted as parallel vertical lines called *lifelines*. Horizontal arrows indicate the exchange of *messages* between any two objects or processes. Here, the exchanged messages are labeled

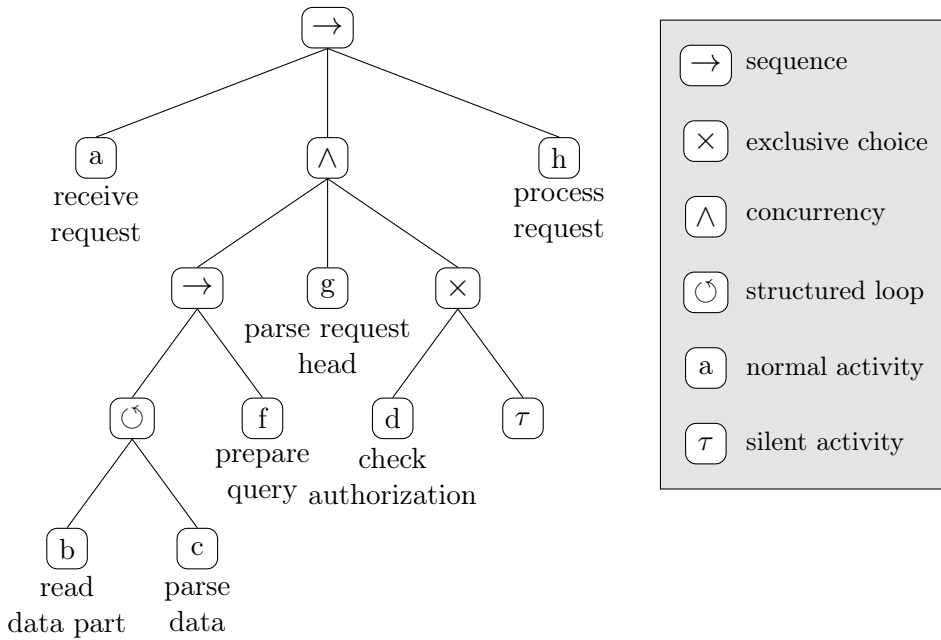
and represent activities. Messages with solid arrowheads represent *synchronous calls*; messages with dashed lines represent *reply* messages. When sending a synchronous message, the source lifeline must wait till the message is done, as indicated by the reply message. Hence, synchronous-reply message pairs model the invocation of subroutines, similar to YAWL composite tasks and BPMN subprocesses. *Activation boxes* are opaque rectangles drawn on top of lifelines to represent that an activity is being performed in response to a message. The vertical order of messages over the lifelines implies the partial order sequence in which activities must occur. Messages can be grouped into *fragments*, drawn as rectangular groups. These fragments can be used to model more advanced concepts such as alternatives (*alt*), optionals (*opt*), loops (*loop*) and parallelism (*par*). We will use the fragment labels *try* and *catch* to model cancelations. In Figure 2.10 an example MSD is given. This MSD has almost the same language as the YAWL model in Figure 2.8, including the submodel for g. The only difference is that the loop retry (activity c) could not be modeled exactly, and can also occur at the end of the loop in this MSD.

Note that Message Sequence Diagrams are traditionally used to describe scenarios, i.e., they typically describe a set of example behaviors. Extensions like *live sequence charts* (LSCs) [59] attempt to formally lift MSDs by explicitly modeling universal (must happen) and existential (can happen) views. For our purposes, we simply assume that the MSDs have a universal semantics; i.e., the diagrams model all the possible runs of a process.

Observe that, unlike the other models presented, Message Sequence Diagrams explicitly capture objects or processes via the lifeline notation.

### 2.2.6 Process Trees

Petri nets, WF-nets, BPMN models and YAWL models may be unsound, i.e., they may suffer from deadlocks, livelocks, and other anomalies. Soundness is an important property, and unsound models are undesirable, unreliable and counter-intuitive. One does not need to look at an event log to see that an unsound model cannot describe the observed behavior well. Process discovery approaches using any of the graph-based process notations mentioned may produce unsound models. In fact, the majority of models in the search space tend to be unsound [8]; this complicates discovery. One solution is to use block-structured models that are sound by construction due to the fact that each control-flow split has, by design, a corresponding join of the same type. In this section, we introduce *process trees* as a notation to represent such block-structured models. In contrast to other block-structured notations such as process algebras [66] and BPEL [48], process trees have been designed to facilitate process discovery and allow for easy manipulation of, and reasoning over, the model.



**Figure 2.12:** Example process tree for a loan application process. The top node is called the root. This process tree has the same language as the Petri net model in Figure 2.5.

A process tree is a hierarchical process model where the (inner) *nodes* are *operators* such as sequence and choice, and the *leaves* are activities. Such a process tree describes a language, and the operators describe how the languages of subtrees are to be combined. In its basic form, there are four types of operators that can be used in a process tree:  $\rightarrow$  *sequence* or sequential composition,  $\times$  *exclusive choice* or XOR choice,  $\wedge$  *concurrency* or parallel composition, and  $\odot$  *structured loop* or redo loop.

We will explain each operator using the example process tree given in Figure 2.12. This process tree has the same language as the Petri net model in Figure 2.5. We start with the *root* node. In Figure 2.12, the root node is a *sequence* operator ( $\rightarrow$ ) with three children: an activity  $a$ , a subtree, and an activity  $h$ . This means that every process instance or trace starts with activity  $a$ , followed by the subtree, and ends with activity  $h$ . The subtree rooted in the *concurrency* operator ( $\wedge$ ) executes all of its children concurrently, i.e., any interleaving or parallel execution is possible. The *exclusive choice* subtree ( $\times$ ) chooses one of its subtrees to execute: either  $d$  or the silent activity  $\tau$ . Since the silent activity  $\tau$  cannot be observed, this means that the activity  $d$  can either be executed or skipped. The subtree rooted in the *structured loop* operator ( $\odot$ ) starts with the leftmost child and may loop back through any of



its other children. This means we execute activity  $b$ , and if we choose to loop back by executing the “redo” activity  $c$ , then we can execute  $b$  again.

The process tree in Figure 2.12 can also be represented textually:

$$\rightarrow(a, \wedge(\rightarrow(\circ(b, c), f), g, \times(d, \tau)), h)$$

The same activity may appear multiple times in the same process tree. For example, the tree  $\rightarrow(a, a, a)$  models a sequence of three  $a$  activities. Since the silent activity  $\tau$  cannot be observed, we can use it to model various properties:  $\times(a, \tau)$  models an activity  $a$  that can be skipped;  $\circ(a, \tau)$  models the process that executes  $a$  at least once; and  $\circ(\tau, a)$  models the process that executes  $a$  any number of times.

To formalize process trees, we introduce the syntax, semantics, and functions defined below.

**Definition 2.2.11 — Process tree.** We formally define *process trees* recursively. We assume a finite alphabet  $\mathbb{A}$  of activities and a set  $\otimes$  of operators to be given. The symbol  $\tau \notin \mathbb{A}$  denotes the silent activity.

Any  $a \in (\mathbb{A} \cup \{\tau\})$  is a process tree. Let  $Q_1, \dots, Q_n$  with  $n > 0$  be process trees and let  $\otimes \in \otimes$  be a process tree operator, then  $\otimes(Q_1, \dots, Q_n)$  is a process tree. We consider the following process tree operators:

- $\rightarrow$  denotes a *sequence* or the sequential composition of all subtrees
- $\times$  denotes an *exclusive choice* or XOR choice between one of the subtrees
- $\wedge$  denotes *concurrency* or the parallel composition of all subtrees
- $\circ$  denotes the *structured loop* or redo loop with loop body  $Q_1$  and alternative loop back paths  $Q_2, \dots, Q_n$  (with  $n \geq 2$ )

We define the semantics and language of process trees directly by defining how each operator combines the languages of its subtrees. The leaves are defined directly, completing this recursive language definition. Recall that sequence concatenation ( $\cdot$ ) and shuffle ( $\diamond$ ) were explained in Definition 2.1.2 on page 22.

**Definition 2.2.12 — Process tree Semantics and Language.** Let  $Q$  be a process tree over the alphabet  $\mathbb{A}$  and let  $\tau \notin \mathbb{A}$  denote the silent activity. The *language* of a process tree  $Q$ , notation  $\mathcal{L}(Q)$ , is defined recursively:

The basis of the process tree language are the activity  $a \in \mathbb{A}$  and silent activity ( $\tau$ ) leaves. For the activity leaf we define the singleton trace and for the silent activity leaf the empty trace. For any process tree operator  $\otimes$ , we use the operator specific *language-join functions*  $\otimes_{\mathcal{L}}$  defined below:

$$\begin{aligned}\mathcal{L}(a) &= \{ \langle a \rangle \} \\ \mathcal{L}(\tau) &= \{ \varepsilon \} \\ \mathcal{L}(\otimes(Q_1, \dots, Q_n)) &= \otimes_{\mathcal{L}}(\mathcal{L}(Q_1), \dots, \mathcal{L}(Q_n))\end{aligned}$$

The *sequence* operator concatenates the language of all its subtrees in order. For  $\rightarrow(Q_1, Q_2, \dots, Q_n)$  we define the language-join function  $\rightarrow_{\mathcal{L}}$ :

$$\rightarrow_{\mathcal{L}}(L_1, \dots, L_n) = \{ \sigma_1 \cdot \sigma_2 \cdot \dots \cdot \sigma_n \mid \forall 1 \leq i \leq n \sigma_i \in L_i \}$$

The *exclusive choice* operator chooses one of its subtrees for execution. For  $\times(Q_1, Q_2, \dots, Q_n)$  we define the language-join function  $\times_{\mathcal{L}}$ :

$$\times_{\mathcal{L}}(L_1, \dots, L_n) = \bigcup_{1 \leq i \leq n} L_i$$

The *concurrency* operator yields the parallel or interleaved composition of all subtrees. For  $\wedge(Q_1, Q_2, \dots, Q_n)$  we define the language-join function  $\wedge_{\mathcal{L}}$  using the sequence shuffle operator ( $\diamond$ ) from Definition 2.1.2:

$$\wedge_{\mathcal{L}}(L_1, \dots, L_n) = \{ \sigma \in (\sigma_1 \diamond \sigma_2 \diamond \dots \diamond \sigma_n) \mid \forall 1 \leq i \leq n \sigma_i \in L_i \}$$

The *structured loop* starts with the leftmost child (the loop body) and may loop back through any of its other children (the redo paths). For  $\circ(Q_1, Q_2, \dots, Q_n)$  we define the language-join function  $\circ_{\mathcal{L}}$ :

$$\begin{aligned}\circ_{\mathcal{L}}(L_1, \dots, L_n) &= \{ \sigma_1 \cdot \sigma'_1 \cdot \sigma_2 \cdot \sigma'_2 \cdot \dots \cdot \sigma_{m-1} \cdot \sigma'_{m-1} \cdot \sigma_m \mid m \geq 1 \wedge \\ &\quad \forall 1 \leq i \leq m \sigma_i \in L_1 \wedge \forall 1 \leq i < m \sigma'_i \in \bigcup_{2 \leq j \leq n} L_j \}\end{aligned}$$

Observe that a loop indirectly models a XOR choice between the redo paths. See for example the language  $\mathcal{L}(\circ(a, b, c))$  in the example below. In addition, note that, based on the semantic definition, the order of children for the operators  $\times$ ,  $\wedge$ , and the order of non-first children for the operator  $\circ$  are arbitrary. For example,  $\circ(a, b, c)$  and  $\circ(a, c, b)$  are equivalent. We will address these properties in more detail in the rewrite rules for process on the next page.

■ **Example 2.1** The following examples further illustrate the process tree operators and their semantics:

$$\begin{aligned}\mathcal{L}(\tau) &= \{ \varepsilon \} \\ \mathcal{L}(a) &= \{ \langle a \rangle \} \\ \mathcal{L}(\rightarrow(a, b, c)) &= \{ \langle a, b, c \rangle \} \\ \mathcal{L}(\times(a, b, c)) &= \{ \langle a \rangle, \langle b \rangle, \langle c \rangle \}\end{aligned}$$

$$\begin{aligned}
\mathcal{L}(\wedge(a, b, c)) &= \{ \langle a, b, c \rangle, \langle a, c, b \rangle, \langle b, a, c \rangle, \langle b, c, a \rangle, \langle c, a, b \rangle, \langle c, b, a \rangle \} \\
\mathcal{L}(\odot(a, b, c)) &= \{ \langle a \rangle, \langle a, b, a \rangle, \langle a, c, a \rangle, \langle a, b, a, c, a \rangle, \langle a, c, a, b, a \rangle, \dots \} \\
\mathcal{L}(\odot(a, \times(b, c))) &= \{ \langle a \rangle, \langle a, b, a \rangle, \langle a, c, a \rangle, \langle a, b, a, c, a \rangle, \langle a, c, a, b, a \rangle, \dots \} \\
\mathcal{L}(\odot(\tau, a, b, c)) &= \{ \varepsilon, \langle a \rangle, \langle b \rangle, \langle c \rangle, \langle a, a \rangle, \langle a, b \rangle, \langle a, c \rangle, \langle b, a \rangle, \langle b, b \rangle, \dots \} \\
\mathcal{L}(\rightarrow(a, \times(b, c), \wedge(a, a))) &= \{ \langle a, b, a, a \rangle, \langle a, c, a, a \rangle \} \\
\mathcal{L}(\times(\tau, a, \tau, \rightarrow(\tau, b), \wedge(c, \tau))) &= \{ \varepsilon, \langle a \rangle, \langle b \rangle, \langle c \rangle \}
\end{aligned}$$

Observe that the model  $\odot(\tau, a, b, c)$  allows for any behavior over the activities  $a, b, c$ , i.e.,  $\mathcal{L}(\odot(\tau, a, b, c)) = \{a, b, c\}^*$ . This particular construction is known as a *flower model*, and this idea will be used again in Chapter 4. ■

Process trees are *sound by construction*, and each process tree is easily translatable to, amongst others, sound block-structured workflow nets. For example, compare Figure 2.12 and Figure 2.5.

Observe that multiple process trees can describe the same language. With the reduction rules described in Table 2.1, we can transform an arbitrary process tree into a normal form. It is not hard to reason that these rules preserve language. A process tree on which these rules have been applied exhaustively is a *reduced* process tree.

**Table 2.1:** Reduction rules for process trees. By applying these rules exhaustively, one obtains a reduced process tree in a normal form.

$$\begin{array}{ll}
\otimes(Q_1) = Q_1 & \text{for } \otimes \in \{\rightarrow, \times, \wedge\} \\
\otimes(\dots_1, \otimes(\dots_2), \dots_3) = \otimes(\dots_1, \dots_2, \dots_3) & \text{for } \otimes \in \{\rightarrow, \times, \wedge\} \\
\otimes(\dots_1, \tau, \dots_2) = \otimes(\dots_1, \dots_2) & \text{for } \otimes \in \{\rightarrow, \wedge\} \\
\times(\dots_1, \tau, \dots_2) = \times(\dots_1, \dots_2) & \text{if } \varepsilon \in \mathcal{L}(\dots_1 \cup \dots_2) \\
\odot(\odot(Q_1, \dots_1), \dots_2) = \odot(Q_1, \dots_1, \dots_2) & \\
\odot(\dots_1, \times(\dots_2), \dots_3) = \odot(\dots_1, \dots_2, \dots_3) & 
\end{array}$$

When calculating and reasoning with process trees, the utility functions defined below can come in handy.

**Definition 2.2.13 — Process Tree Functions.** We define the following utility functions for a process tree  $Q$  and a given node  $Q'$  somewhere in  $Q$ .

$children(Q')$  Node children: the set of children for a given node  $Q'$ :

$$children(Q') = \begin{cases} \{Q_1, \dots, Q_n\} & \text{if } Q' = \otimes(Q_1, \dots, Q_n) \\ \emptyset & \text{otherwise} \end{cases}$$

For example:  $children(a) = \emptyset$ ,  $children(\rightarrow(a, b)) = \{a, b\}$ , and  $children(\rightarrow(a, \times(b, c))) = \{a, \times(b, c)\}$ .

$enum(Q')$  *Enumerate tree*: the set of nodes for a given node  $Q'$ :

$$enum(Q') = \{Q'\} \cup \bigcup_{Q'' \in children(Q')} enum(Q'')$$

For example:  $enum(a) = \{a\}$ ,  $enum(\rightarrow(a, b)) = \{a, b, \rightarrow(a, b)\}$ , and  $enum(\rightarrow(a, \times(b, c))) = \{\times(b, c), b, c, \rightarrow(a, \times(b, c)), a\}$ .

$parent(Q, Q')$  *parent node*: the parent of  $Q'$  in the tree  $Q$ :

$$parent(Q, Q') = \begin{cases} Q'' & \text{if } \exists Q'' \in enum(Q) : \\ & Q' \in children(Q'') \\ \perp & \text{otherwise} \end{cases}$$

For example:  $parent(a, b) = \perp$ ,  $parent(\rightarrow(a, b), a) = \rightarrow(a, b)$ , and  $parent(\rightarrow(a, \times(b, c)), b) = \times(b, c)$ .

$path(Q, Q')$  *path sequence*: the sequence of nodes from root  $Q$  to node  $Q'$ :

$$path(Q, Q') = \begin{cases} path(Q, Q'') \cdot \langle Q' \rangle & \text{if } parent(Q, Q') \\ & = Q'' \neq \perp \\ \langle Q' \rangle & \text{otherwise} \end{cases}$$

For example:  $path(\rightarrow(a, b), a) = \langle \rightarrow(a, b), a \rangle$  and  $path(\rightarrow(a, \times(b, c)), b) = \langle \rightarrow(a, \times(b, c)), \times(b, c), b \rangle$ .

## 2.3 Event Logs

Process mining is impossible without proper event logs. An event log stores the execution history of a process. For discovery algorithms and many other process mining techniques, event logs are the main input. These event logs can be found everywhere: in enterprise information systems and business transaction logs, in web servers, in high-tech systems such as X-ray machines, in warehousing systems, and many other places [8].

In this section, we will first discuss the basic structure of an event log in Section 2.3.1. Next, we will discuss two commonly used views on event logs: atomic event logs (Section 2.3.3), and non-atomic event logs (Section 2.3.4).

### 2.3.1 Structure of an Event Log

We will explain the structure of an event log using the example dataset given in Table 2.2. This event log relates to the process model in Figure 2.7. An event log contains data related to a *single process*.

**Table 2.2:** Example event data related to the process model in Figure 2.7. Each line represents one event, each column an attribute. Case 1 records a rejected request (event 1.6 cancels activity *g*, hence no complete was recorded to match event 1.3), while case 2 records a processed request.

Case id	Event id	Attributes				
		Activity	Lifecycle	Timestamp	Resource	...
1	1.1	<i>a</i>	receive request	start	30-10-2017 11:02:45.000	main-thread ...
	1.2	<i>a</i>	receive request	complete	30-10-2017 11:02:45.500	main-thread ...
	1.3	<i>g</i>	parse request head	start	30-10-2017 11:02:45.650	worker-1 ...
	1.4	<i>d</i>	check authorization	start	30-10-2017 11:02:45.651	worker-3 ...
	1.5	<i>d</i>	check authorization	complete	30-10-2017 11:02:45.710	worker-3 ...
	1.6	<i>e</i>	reject request	start	30-10-2017 11:02:45.820	main-thread ...
	1.7	<i>e</i>	reject request	complete	30-10-2017 11:02:45.870	main-thread ...
2	2.1	<i>a</i>	receive request	start	30-10-2017 11:03:12.150	main-thread ...
	2.2	<i>a</i>	receive request	complete	30-10-2017 11:03:12.450	main-thread ...
	2.3	<i>g</i>	parse request head	start	30-10-2017 11:03:12.670	worker-2 ...
	2.4	<i>b</i>	read data part	start	30-10-2017 11:03:12.670	worker-1 ...
	2.5	<i>g</i>	parse request head	complete	30-10-2017 11:03:13.110	worker-2 ...
	2.6	<i>b</i>	read data part	complete	30-10-2017 11:03:13.160	worker-1 ...
	2.7	<i>f</i>	prepare query	start	30-10-2017 11:03:13.320	worker-1 ...
	2.8	<i>f</i>	prepare query	complete	30-10-2017 11:03:13.400	worker-1 ...
	2.9	<i>h</i>	process request	start	30-10-2017 11:03:13.670	main-thread ...
	2.10	<i>h</i>	process request	complete	30-10-2017 11:03:14.220	main-thread ...
3	3.1	<i>a</i>	receive request	start	31-10-2017 09:43:16.030	main-thread ...
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Each line in the table represents one *event*, and each column represents one *attribute* of this event. Each event is associated with a *case* or *process instance*. The sequence of events that are recorded for a case is called a *trace*. As a bare minimum for process mining, we also assume that events can be related to some *label*, e.g., an *activity*. The concept of a *classifier* maps the attributes of an event onto a label, i.e., given an event, a classifier returns the corresponding (activity) label based on the event attributes. In the example of Table 2.2, the events are grouped by case and sorted chronologically; each group is one trace. It is important that each event is grouped into a trace, and that events are sorted. For without ordering information, it is impossible to discover causal dependencies in process models.

In addition to a case and activity, events can be annotated with numerous other attributes. Typically, an event also has an associated *timestamp*, i.e., date and time information for when the event occurred. This information is useful when analyzing performance-related properties. In detailed event logs, events can be associated with a *lifecycle transition* such as *start* and *complete*. This way, the start and complete of an activity, and thus its duration, is recorded. In addition, *resources* can be recorded to indicate which person

executed the activities. Note that the *start* and *complete* lifecycle transitions are a subset of the *transactional lifecycle model* [8]. This model also supports more detailed transitions, such as *schedule*, *pause*, *resume*, *abort*, and *reassign*.

For example, consider the event log in Table 2.2. For case 1, we can see that an activity *g* was started in event 1.3, but never completed. This is due to event 1.6 and the associated cancelation behavior (recall the cancelation region modelled for activity *e* in the process model in Figure 2.7).

To recap, we assume the following general structure of an event log:

- An *event log* consists of *cases*.
- A case consists of *events* such that each event relates precisely to one case; no event is duplicated in a case or an event log.
- Events within a case are *ordered*.
- Events can have *attributes*. Examples of typical attributes are *activity*, *lifecycle*, *timestamp* and *resource*. Some attributes are optional; below we define the so-called standard attributes.
- A *classifier* maps the attributes of an event onto a label or name. For example, a typical classifier uses the *activity* attribute to label events.

The above structure and assumptions are captured in the following definition for *events*, *classifiers*, and an *event log*.

**Definition 2.3.1 — Event and Attributes.** Let  $\mathbb{E}$  be the *event universe*, i.e., the set of all possible event identifiers. Events may be characterized by various *attributes*, such as an activity name, a timestamp, etc. For any event  $e \in \mathbb{E}$  and attribute name  $n$ , the value of attribute  $n$  for event  $e$  is  $\#_n(e)$ . If event  $e$  does not have an attribute named  $n$ , then  $\#_n(e) = \perp$ , i.e., the null value.

For convenience, we assume the following standard attributes:

- $\#_{act}(e)$       the *activity* associated to event  $e$ .
  - $\#_{time}(e)$     the *timestamp* of event  $e$ .
  - $\#_{res}(e)$       the *resource* associated to event  $e$ .
  - $\#_{life}(e)$     the *lifecycle transaction type* associated to event  $e$ .
- For example: *start*, *complete*, *schedule*, etc.

**Definition 2.3.2 — Classifier.** A *classifier* is a function that maps the attributes of an event onto a label. For any event  $e \in \mathbb{E}$  and classifier  $\lambda_{\#}$ , the *name* of event  $e$  is given by  $\lambda_{\#}(e)$ .

If events are simply identified by their activity name, then  $\lambda_{\#}(e) = \#_{act}(e)$ . For example, event 1.1 in Table 2.2 would be mapped onto *a*.

If events are identified by their activity name and lifecycle transaction type, then  $\lambda_{\#}(e) = (\#_{act}(e), \#_{life}(e))$ . For example, event 1.1 in Table 2.2 would be mapped onto  $(a, start)$ .

**Definition 2.3.3 — Event Log.** Let  $\mathbb{E}$  be a set of events. Let  $L \subseteq \mathbb{E}^*$  be an event log, a set of traces such that no event is duplicated in  $L$ . That is, for any two traces  $\sigma, \sigma' \in L$  and any two numbers  $i, j$  such that  $1 \leq i \leq |\sigma|$  and  $1 \leq j \leq |\sigma'|$  we have  $(\sigma(i) = \sigma'(j)) \Rightarrow (\sigma = \sigma' \wedge i = j)$ .

### 2.3.2 XES – Extensible Event Stream

The *XES* format, or *eXtensible Event Steam* format, is the de facto standard to store and exchange event logs. XES is the successor of MXML and was adopted by the *IEEE Task Force on Process Mining* in September 2010 [8]. An XES document is an XML file that contains one log and consists of any number of traces. Each trace describes a sequential list of events corresponding to a particular case. The log, its traces, and its events can have any number of attributes. Attributes are typed key-value pairs and may be nested.

The XES standard does not prescribe a fixed set of mandatory attributes for each element. To provide semantics for attributes, the log refers to so-called *extensions*. An extension gives semantics to particular attributes. The XES standard extensions define the basic attributes mentioned in the previous section. For example, the activity of an event is specified via the *Concept* extension. The lifecycle transitions are detailed in the *Lifecycle* extension. And the timestamp and resource attributes are specified in the *Time* and *Organizational* extension respectively. In addition, XES allows new extensions to be defined and used by end users.

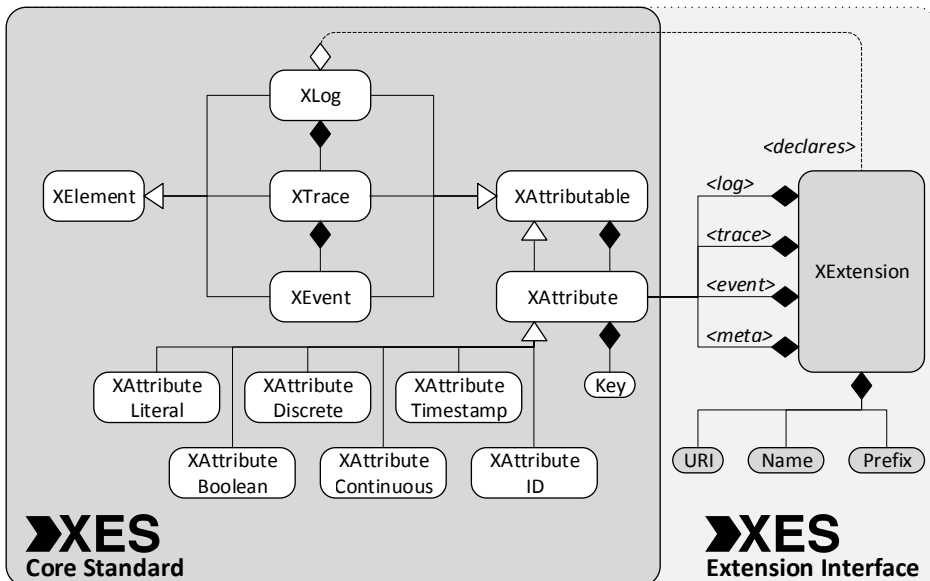


Figure 2.13: The XES metamodel, as originally defined in [77, 187].

Figure 2.13 presents the XES metamodel. For more detailed information on XES and the XES metamodel, we refer the reader to [77, 187]. We will introduce some novel XES extensions in Section 5.3 on page 102. For the basic discovery algorithms and their explanation, we will abstract from the rich event log structure and XES format.

### 2.3.3 Atomic Event Log

For the basic discovery algorithms and their explanation, the rich event log structure described above is a bit of overkill. If one is just interested in when which activity occurred, we can use a simplified view of the event log. In this section, we will define the *atomic event log* model and show how an event log can be transformed to an atomic event log using a classifier.

In the *atomic event log*, we assume that an event is just an atomic activity without any further attributes. Hence, we can describe an atomic event log simply as a multiset of traces, where each trace is a sequence of activities.

**Definition 2.3.4 — Atomic Event Log.** Let  $\mathbb{A}$  be a set of activities. Let  $L_A \in \mathcal{B}(\mathbb{A}^*)$  be an atomic event log, a multiset of traces. A trace  $\sigma \in L_A$  with  $\sigma \in \mathbb{A}^*$  is a sequence of activities.

**Definition 2.3.5 — Transforming an Event Log into an Atomic Event Log.** Let  $L \subseteq \mathbb{E}^*$  be an event log as defined in Definition 2.3.3. Assume a classifier  $\lambda_{\#}$  has been defined. Any trace  $\sigma \in L$  can be mapped onto a sequence of activities using classifier  $\lambda_{\#}$  as follows:  $\langle \lambda_{\#}(e) \mid e \in \sigma \rangle$ . The atomic event log  $L_A$  is derived as follows:  $L_A = [\langle \lambda_{\#}(e) \mid e \in \sigma \rangle \mid \sigma \in L]$

Note that, for the transformation in Definition 2.3.5, any classifier is valid. That is, we are not restricted to using only the activity name attribute.

■ **Example 2.2** The rich event log in Table 2.2 can be represented as different atomic event logs. For the sake of the example, we assumed that the activity sequence of case 1 occurred 12 times, and the activity sequence of case 2 occurred 23 times. The table below shows a few transformations using different classifiers and different filters applied before transformation.

atomic event log	filter and classifier
$L_A = [\langle a, a, g, d, d, e, e \rangle^{12}, \langle a, a, g, b, g, b, f, f, h, h \rangle^{23}, \dots]$	<i>Filter:</i> none <i>Classifier:</i> $\lambda_{\#}(e) = \#_{act}(e)$
$L_A = [\langle a, d, e \rangle^{12}, \langle a, g, b, f, h \rangle^{23}, \dots]$	<i>Filter:</i> remove start events <i>Classifier:</i> $\lambda_{\#}(e) = \#_{act}(e)$
$L_A = [\langle a, g, d, e \rangle^{12}, \langle a, g, b, f, h \rangle^{23}, \dots]$	<i>Filter:</i> remove complete events <i>Classifier:</i> $\lambda_{\#}(e) = \#_{act}(e)$



atomic event log	filter and classifier
$L_A = [\langle \text{main-thread, worker-3, main-thread} \rangle^{12}, \langle \text{main-thread, worker-2, worker-1, worker-1, main-thread} \rangle^{23}, \dots]$	<i>Filter:</i> remove <i>start</i> events <i>Classifier:</i> $\lambda_{\#}(e) = \#_{res}(e)$

Note the repetition of activities in the first example. This is due to the lifecycle information in Table 2.2. By filtering out start or complete events, different atomic event logs can be obtained, see the second and third example. By using a different classifier, different views can be obtained. The last example shows how the resource classifier can be used to get an atomic event log detailing the relation between resources/threads. ■

The atomic event log allows for some simple but rich event calculations. A common operation used in (discovery) algorithms is the activity projection.

**Definition 2.3.6 — Activity Log-Projection.** Given a set  $\Sigma \subseteq \mathbb{A}$  and trace  $\sigma \in L_A$ , we write  $\Sigma(\sigma) = \Sigma \cap \{a \in \sigma\}$  to denote the set of activities in the intersection of  $\Sigma$  and  $\sigma$ . We write  $\Sigma(L_A) = \Sigma \cap \{a \in \sigma \mid \sigma \in L_A\}$  to denote the set of activities in the intersection of  $\Sigma$  and  $L_A$ .

We write  $\mathbb{A}(\sigma)$  to get the complete set of activities in  $\sigma$ , and  $\mathbb{A}(L_A)$  to get the complete set of activities in  $L_A$ .

We can use the activity log-projection and normal multiset operations to derive various event log properties. For instance, the size of an event log  $L$ , notation  $|L|$ , equals the number of traces in that log. The size of the activity alphabet of an event log  $L$ , notation  $|\mathbb{A}(L)|$  equals the number of unique activities in that log.

For example, we have for the log  $L = [\langle a, b, a, c \rangle, \langle c, d, e \rangle]$ :

$$\begin{aligned}
 |L| &= 2 \\
 \mathbb{A}(L) &= \{a, b, c, d, e\} \\
 |\mathbb{A}(L)| &= |\{a, b, c, d, e\}| = 5
 \end{aligned}$$

### 2.3.4 Non-Atomic Event Log

In atomic event logs, an event denotes the execution of an activity, which is assumed to be atomic or instantaneous. For some cases, this view is a bit too simplistic. In a *non-atomic event log*, executions of activities are represented by two events instead of one: a start event and a complete event. That is, we include a basic version of the lifecycle transitions and indirectly capture the intervals during which activities are executed. Note that the full *transactional lifecycle model* supports more detailed transitions, such as schedule, pause, and

abort. However, for the purpose of capture the activity execution intervals, we do not need the full lifecycle model.

In this section, we define the *non-atomic event log* model and show how an event log can be transformed to a non-atomic event log.

**Definition 2.3.7 — Non-Atomic Event Log.** Let  $\mathbb{A}$  be a set of activities. Let  $LC = \{start, complete\}$  be the set of lifecycle transitions, denoting start and complete respectively. Let  $L_{LC} \in \mathcal{B}((\mathbb{A} \times LC)^*)$  be a non-atomic event log, a multiset of traces. A trace  $\sigma \in L_{LC}$  with  $\sigma \in (\mathbb{A} \times LC)^*$  is a sequence of activities annotated with start or complete.

We write  $a+s$  to denote the *start activity* pair  $(a, start)$ , with  $a \in \mathbb{A}$  and  $start \in LC$ . We write  $a+c$  to denote the *complete activity* pair  $(a, complete)$ , with  $a \in \mathbb{A}$  and  $complete \in LC$ .

**Definition 2.3.8 — Transforming an Event Log into an Non-atomic Event Log.**

Let  $L \subseteq \mathbb{E}^*$  be an event log as defined in Definition 2.3.3. Assume we have defined: 1) an activity label classifier  $\lambda_{\#}$ , and 2) a lifecycle transition attribute  $\#_{life}$ . Any trace  $\sigma \in L$  can be mapped onto a sequence of activity-lifecycle pairs as follows:  $\langle (\lambda_{\#}(e), \#_{life}(e)) \mid e \in \sigma \rangle$ . The non-atomic event log  $L_{LC}$  is derived as follows:

$$L_{LC} = [ \langle (\lambda_{\#}(e), \#_{life}(e)) \mid e \in \sigma \rangle \mid \sigma \in L ]$$

■ **Example 2.3** The rich event log in Table 2.2 can be represented as the following non-atomic event log, after filtering on start and complete events and using the classifier  $\lambda_{\#}(e) = \#_{act}(e)$ . Again, we assumed that the activity sequence of case 1 occurred 12 times, and the activity sequence of case 2 occurred 23 times.

$$L_{LC} = [ \langle a+s, a+c, g+s, d+s, d+c, e+s, e+c \rangle^{12}, \\ \langle a+s, a+c, g+s, b+s, g+c, b+c, f+s, f+c, h+s, h+c \rangle^{23}, \dots ]$$

■

## 2.4 Directly-Follows Relation and the Directly-Follows Graph

In this section, we introduce the *directly-follows relation* and its associated *directly-follows graph*. This relation expresses a causal language abstraction that is used in many process mining techniques. These abstractions are used as a basis for the discovery techniques described in this thesis. Both the directly-follows relation and graph can be defined over an (atomic) event log, but also over a process model [8, 130]. Hence, we will first explain these concepts over a language, and later give examples for both an event log and a process model. In this context, a language is simply a set of traces, where each trace is a sequence of symbols or activities.

### 2.4.1 Definition of the Directly-Follows Relation and Graph

The *directly-follows relation* specifies which activities can directly follow one another in a language. This relation is an abstraction over all the traces in a language. For example, in the language  $L = \{\langle a, b \rangle, \langle a, c \rangle\}$ , activity  $a$  is directly followed by  $b$  and  $a$  is directly followed by  $c$ , but there is no directly-follows relation between  $b$  and  $c$ .

The *directly-follows graph* captures the directly-follows relation, and explicitly tracks the start and end activities of a language. For example, in the language  $L = \{\langle a, b \rangle, \langle a, c \rangle\}$ , the directly-follows graph consists of three nodes and two edges: one edge from  $a$  to  $b$ , and one edge from  $a$  to  $c$ . In this language  $a$  is a start activity, and  $b$  and  $c$  are end activities.

We formally define the directly-follows relation and directly-follows graph below.

**Definition 2.4.1 — Directly-Follows Relation and Graph.** Let  $\mathbb{A}$  be a set of activities, and let  $L \subseteq \mathbb{A}^*$  be a language or atomic event log over  $\mathbb{A}$ . Let  $\triangleright \notin \mathbb{A}$  denote the start of a trace and let  $\square \notin \mathbb{A}$  denote the end of a trace. The *directly-follows relation* over  $L$ , notation  $\succrightarrow_L$ , is defined as:

$$\begin{aligned} \succrightarrow_L = & \{ (a, b) \in \mathbb{A} \times \mathbb{A} \mid \exists \sigma \in L : \sigma = \sigma_1 \cdot \langle a, b \rangle \cdot \sigma_2 \} \\ & \cup \{ (\triangleright, a) \mid a \in \mathbb{A} \wedge \exists \sigma \in L : \sigma = \langle a \rangle \cdot \sigma_2 \} \\ & \cup \{ (a, \square) \mid a \in \mathbb{A} \wedge \exists \sigma \in L : \sigma = \sigma_1 \cdot \langle a \rangle \} \\ & \cup \{ (\triangleright, \square) \mid \varepsilon \in L \} \end{aligned}$$

We write  $a \succrightarrow_L b$  to denote  $(a, b) \in \succrightarrow_L$ .

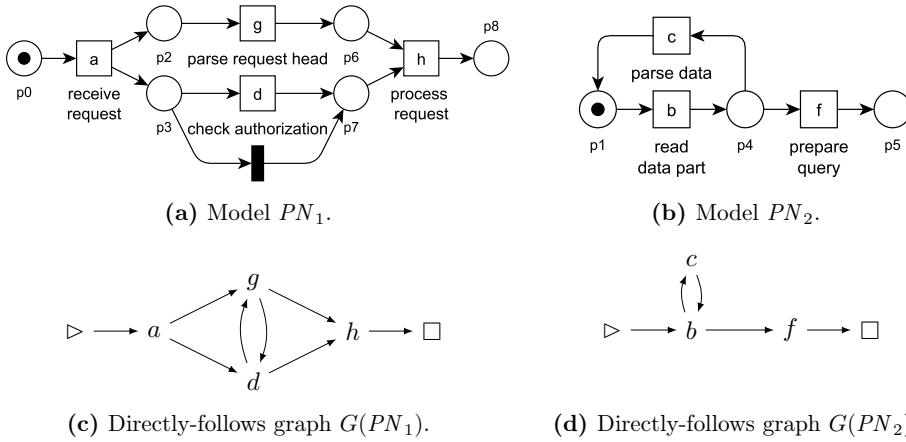
The *directly-follows graph* over  $L$  is the graph  $G(L)$  induced by  $\succrightarrow_L$ , i.e.,  $G(L) = (\bigcup_{a \succrightarrow_L b} \{a, b\}, \succrightarrow_L)$ . We define the following functions over directly-follows graphs:

$$\begin{aligned} Start(G) &= \{ a \in \mathbb{A} \mid \triangleright \succrightarrow_L a \} && \text{The set of all start activities in } G. \\ End(G) &= \{ a \in \mathbb{A} \mid a \succrightarrow_L \square \} && \text{The set of all end activities in } G. \end{aligned}$$

The set of start and end activities for a log  $L$  is defined as  $Start(L) = Start(G(L))$  and  $End(L) = End(G(L))$  respectively. The set of start and end activities for a model  $Q$  is defined as  $Start(Q) = Start(G(\mathcal{L}(Q)))$  and  $End(Q) = End(G(\mathcal{L}(Q)))$  respectively.

### 2.4.2 Example Directly-Follows Graphs

As stated before, the directly-follows relation and graph can be defined over an (atomic) event log, but also over a process model. The definition above assumed a generic language. Below, we will discuss concrete examples for both process models and event logs.



**Figure 2.14:** Example Petri nets and their corresponding directly-follows graphs. These models are subsets of the model in Figure 2.5. Note that the start activities are denoted by an incoming arc starting in  $\triangleright$ , and the end activities are denoted by an outgoing arc ending in  $\square$ . See also Example 2.4.

■ **Example 2.4 — Directly-Follows Graph for Process Models.** In Figure 2.14 two Petri net models and their corresponding directly-follows graphs are depicted.

In model  $PN_1$ , Figure 2.14a, we have an activity  $a$ , followed by activities  $d$  and  $g$  in parallel, followed by activity  $h$ . Note that in  $G(PN_1)$ , Figure 2.14c, the parallel behavior is expressed by the strongly connected component  $\{d, g\}$ , indicating these activities can happen in any interleaving. Observe that removing the silent transition ( $\tau$ ) in  $PN_1$  does not change its directly-follows graph.

In model  $PN_2$ , Figure 2.14b, we have an activity  $b$ , after which activity  $c$  allows for a redo of activity  $b$  (loop back), followed by activity  $f$ . In  $G(PN_2)$ , Figure 2.14d, we also observe a strongly connected component:  $\{b, c\}$ . Note that in  $G(PN_2)$  we only start in activity  $b$ , and can only continue with  $f$  after  $b$ , i.e., it is not possible to start or end with  $c$ . Hence, this strongly connected component captures the intended loop, and not some interleaving behavior. ■

Observe how, in the above example, patterns in the directly-follows graph are related to process model constructs. Likewise, as we will show in the example below, patterns in an event log can be related to patterns in the directly-follows graph.

■ **Example 2.5 — Directly-Follows Graph for Atomic Event Logs.** We will be reusing the directly follows graph from Figure 2.14. We have:

atomic event log	directly-follows graph
$L_1 = [\langle a, g, d, h \rangle, \langle a, d, g, h \rangle]$	$G(L_1) = G(PN_1)$ , Fig. 2.14c
$L_2 = [\langle b, f \rangle, \langle b, c, b, f \rangle, \langle b, c, b, c, b, f \rangle]$	$G(L_2) = G(PN_2)$ , Fig. 2.14d

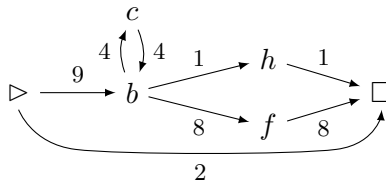
Observe that the atomic event log  $L'_1 = \{\langle a, g, h \rangle, \langle a, d, h \rangle\}$  would not yield  $G(PN_1)$  as a directly-follows graph. By removing the edges  $(d, g)$  and  $(g, d)$  from  $G(PN_1)$ , we would obtain the directly-follows graph  $G(L'_1)$ . ■

In the above examples, patterns in the event log can be abstracted and detected via the directly-follows graph, which in turn can be related to process models. Several process discovery algorithms have exploited these properties; see Chapter 3 for an extensive discussion. In Chapter 4, we will explore these directly-follows relations further for the purpose of process discovery.

### 2.4.3 Annotated Directly-Follows Graphs

In the above sections, we introduced the basic notion of directly-follows graphs. When deriving such graphs from event logs, additional information can be derived and annotated onto these graphs. For example, frequency information can be useful for discovery algorithms to determine which behavior occurs more frequently than other behavior [130]. In addition, likelihood or probability annotations could be used to evaluate incomplete behavior [130], and dependency annotations can be used to evaluate heuristic measures [192].

In order to extend directly-follows graphs with frequency information, the set of nodes become multisets and the edges become annotated with a weight denoting how often a relation happened. See also the example below.



**Figure 2.15:** Directly-follows graph with frequency annotations on the edge. The numbers on each arc indicate how often that relation happened in the following atomic event log:  $L = [\varepsilon^2, \langle b, f \rangle^5, \langle b, c, b, f \rangle^2, \langle b, c, b, c, b, f \rangle, \langle b, h \rangle]$ .

■ **Example 2.6 — Directly-Follows Graph with Frequency Annotations.** For example, consider the annotated directly-follows graph in Figure 2.15. The edge from  $\triangleright$  directly to  $\square$  captures that 2 empty traces were recorded in  $L$ . The edge from  $\triangleright$  to  $b$  captures that 9 traces started with activity  $b$  and the edge from  $b$  to  $h$  informs us that in only one trace activity  $b$  was followed by activity  $h$ .

Using a frequency threshold, a discovery algorithm could for example prune infrequent edges from the graph in Figure 2.15, yielding for example the reduced graph in Figure 2.14d. Similarly, using a frequency threshold, one could also prune infrequent nodes. ■

## 2.5 Conclusion

In this chapter, we introduced the definitions, formalizations, and notations used throughout the rest of the thesis. In the next chapter, we will describe a discovery algorithm based on the process tree notation (Section 2.2.6) and directly-follows graph (Section 2.4).



*“It is not only the violin that shapes the violinist, we are all shaped by the tools we train ourselves to use.”*

— Edsger W. Dijkstra

## 3 | Related Work

3

In this thesis, we address some of the challenges in applying process mining to analyze actual software system behavior. As a result, our work can be positioned in-between reverse engineering of software systems and process mining. In addition, this thesis focusses on investigating and exploiting hierarchical structures.

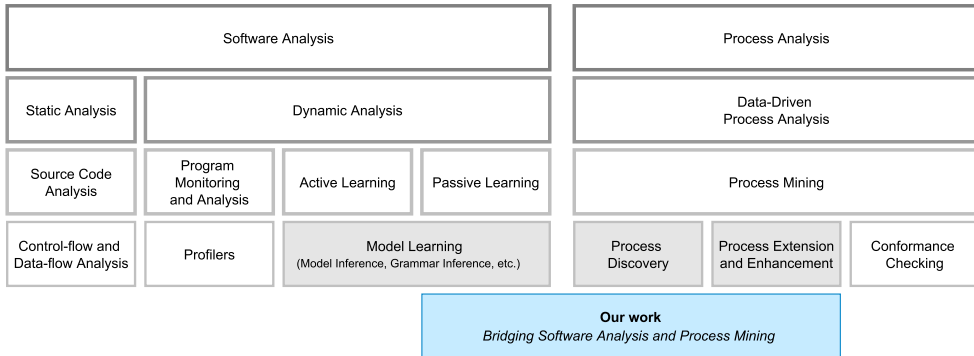
This chapter presents a broad and systematic survey of both the reverse engineering and process mining fields and touches upon related hierarchical techniques. The remainder of this chapter is organized as follows. Section 3.1 presents a taxonomy covering both fields. Section 3.2 discusses the similarity and differences between a selection of *model learning* and *process discovery*. Section 3.3 surveys scalability in related work and the use of hierarchies, hierarchical decomposition, and submodels in the broader *process mining* perspective. Section 3.4 will discuss various aspects of performance analysis in both fields, and investigate the use of hierarchies and submodels in the context of performance analysis.

### 3.1 A Taxonomy covering Reverse Engineering and Process Mining

Substantial work has been done on analyzing and (re)constructing models from software, example behavior, or (event) logs in a variety of research fields. Due to this variety and complexity, we will first establish the taxonomy and terminology used in the rest of the chapter.

The problem of (re)constructing models from systems, be it software systems, (business) processes, or any generic system with behavior, has been studied under different names in different communities [8, 183]: Control theorists refer to it as system identification, computational linguists speak about grammatical inference [90, 150]. Some papers use terms like regular inference [34], model inference [190], model identification [85], regular extrapolation [78], automata learning [91], or model learning [183]. Security researchers coined the term protocol state fuzzing [62]. The process mining community talks about process discovery [8]. In addition, the software engineering field also tried to (re)construct behavior by looking at static source code artifacts





**Figure 3.1:** Taxonomy positioning Model Learning and Process Discovery in the scope of Reverse Engineering of Software Systems and Process Mining.

(e.g., control-flow analysis [166]), or using online program analysis (e.g., profilers [73]). To make sense of this broad research topic, we classify these model (re)constructing techniques using the taxonomy presented in Figure 3.1.

In the software analysis and engineering community, we partition related work into static analysis and dynamic analysis. *Static analysis* derives its analyses and models without actually executing the software. It uses source code, object code, byte code, or other static sources as input. In addition, documentation and (design) models, both formal and informal, can be used. Instead of running the code, one tries to understand the internal logic of the software and how it (statically) connects with other parts. A typical example is the work on control-flow and data-flow analysis [100, 166]. *Dynamic analysis* derives its analyses and models by executing software programs on a real or virtual processor.

We further partition the school of dynamic analysis into program monitoring and analysis, active learning and passive learning. *Program monitoring and analysis* observe the software system on the run and present (aggregate) runtime information and notifications. These approaches have only a limited focus on constructing (formal) control flow models. Examples include profilers [73, 136, 191] and application performance management (APM) suites [31, 164, 182]. *Active learning* tries to learn or discover a behavioral model by actively interacting with the software system, i.e., it queries the system instead of relying on a log. That is, a special *learner* algorithm interacts with the software under study, and observes the responses and output. The learner metaphorically presses the buttons of the system and observes what happens. The most efficient learning algorithms used today all follow the seminal work on  $L^*$  by Angluin [28, 183]. *Passive learning* tries to learn or discover a behavioral model *without* interacting with the software system, i.e., it relies on some

form of logging or passive monitoring. In contrast to active learning, in passive learning, one only observes the software system. We will use the term *model learning* [183] to refer to the family of active and passive learning techniques in the software engineering community.

In the process mining community, model (re)construction from logs is commonly referred to as *process discovery* [8]. In contrast to many software analysis techniques, process mining provides a powerful and mature way of combining and integrating both model (re)construction and analysis. That is, process mining combines extracting (formal) enriched and annotated models and enabling performance and conformance analysis using these models. With the use of structured and well-defined event logs numerous existing techniques can be combined for advanced analyses.

We refer to Section 1.1 for an introduction to process mining techniques and concepts. In Section 3.2.3 we will discuss the subtle but important differences in assumptions between (software) model learning and process discovery.

## 3.2 Discovery and Learning Techniques

In this subsection, we provide a systematic comparison of existing reverse engineering and process discovery techniques. We will leverage the taxonomy of Section 3.1 and Figure 3.1 to classify the related work. The comparison (Section 3.2.2) will be based on the set of feature criteria (Section 3.2.1), as is summarized in Table 3.1. After that, Section 3.2.3 presents a more high-level comparison of the different schools of discovery and learning techniques.

### 3.2.1 Feature Criteria for Comparison

For the comparison, we defined a set of feature criteria, divided into two groups: 1) *guarantees and analysis support*, and 2) *discovery expressivity*.

To be able to perform meaningful analyses on a discovered or learned model, it is important that the technique and underlying model notation provide some *guarantees and analysis support*. First of all, the underlying model notation should have clear, well-defined, *formal execution semantics*. Clear semantics is a prerequisite for reliable interpretation by both man and machine. For example, without clear semantics, we cannot reliably perform performance and conformance analysis. Secondly, discovered models should be *sound and error-free*, i.e., they should be free of deadlocks and other anomalies. Although in some situations vagueness can be a feature [11], when analyzing software, ambiguity and unclear behavior should be avoided, all process steps should be executable, and an end state should always be reachable. Without soundness, the model, and any conclusion derived from it, is unreliable. Another criterion is that the discovered models should represent the behavior recorded in the

event log, i.e., the technique should be able to *guarantee fitness* for the discovered models. Real-life event logs are often messy and challenging for discovery algorithms. In these cases, it may be necessary to trade fitness for simplicity by filtering out *infrequent* behavior, i.e., discovering a so-called 80/20 model. An 80/20 model describes the mainstream (80%) behavior using a simple (20%) model [130]. In addition, the model should be used as the backbone for further analysis. At the very least, *frequency* (usage) and *performance* (timing) analysis should be supported.

Any discovery technique must decide on which type of behavior and process model constructs it supports, i.e., it must decide on which class of behavior it supports and leverage this *representational bias*. Learning and discovery techniques, and their underlying model notations, should support the type of behavior that is recorded in the event log. At the very least, *choice and loop* constructions should be supported, capturing branching and repetitive behavior. In software systems, typical examples include if-then-else and foreach/iterators respectively. *Concurrency and parallelism* support is equally essential. In reality, parts of a process or system may be executing in a non-deterministic order or simultaneously and potentially interacting with each other [8]. In addition, some behavior can only be captured by *non-local control-flow* constructs such as long-distance dependencies or global behavior such as cancellation. Given the Open Challenges from Section 1.3, we will discuss three types of behavior in more detail:

1. *Named Submodels (Hierarchy)* (see Challenge 2 on page 14),
2. *Recursion or Stack Behavior* (see Challenge 3 on page 14), and
3. *Cancellation or Reset Behavior* (see Challenge 4 on page 15).

### 3.2.2 Discussion of Learning and Discovery Techniques

Using the above criteria, we will now discuss a selection of related learning and discovery techniques as is summarized in Table 3.1.

#### Static Analysis

Static analysis focusses more on insights into relations and dependencies between various statements, and less on obtaining a fitting model with execution semantics. As a consequence, the models obtained have very little behavioral expressivity. Typical examples of static analysis include control-flow and data-flow analysis based on static artifacts like source code. The techniques from [27, 100, 101, 166, 181] discover branching behavior (choices and loops), but not much more. Obviously, since no actual software is executed, no 80/20 model or frequency and performance analysis is possible.

A consequence of static analysis is that these techniques have a difficulty capturing the actual, dynamic runtime behavior. Especially in the case of dynamic types and jumps (e.g., inheritance, method overloading, dynamic bind-

**Table 3.1:** Comparison of related model discovery and learning techniques.

	Author	Technique / Toolkit	Formalism	Execution Semantics	Sound and Error-free	Frequency and Infrequent (80/20 model)	Concurrency and Choice (Non-local control-flow)	Named Submodels and Loops	Cancellation or Reset Behavior (Hierarchy)	Recursion or Stack Behavior	
Static Analysis	[181] Tonella	Interaction control flow	UML Interaction	-	n/a	n/a	n/a	-	✓	-	
	[101] Korshunova	CPP2XMI, SQuADT	UML SD, AD	± <sup>sd</sup>	✓	✓	n/a	-	✓	± <sup>sh</sup>	
	[100] Kollmann	Control Flow	UML Collaboration	-	n/a	n/a	n/a	-	✓	-	
	[166] Rountev	Program control flow	UML SD	± <sup>sd</sup>	✓	✓	n/a	-	✓	± <sup>sh</sup>	
	[27] Amighi	Sawja framework	Control Flow Graph	-	n/a	n/a	n/a	-	✓	-	
Monitoring & Profiling	[73] Graham	gprof profiler	Call graphs	-	n/a	n/a	-	✓	-	± <sup>r</sup>	
	[89] Hoorn, van	Kieker Framework	Call graphs	-	n/a	n/a	-	✓	-	✓	
	[177] Szvetits	Reusable Event Types	UML, any	-	n/a	n/a	-	✓	-	✓	
	[136] Yourkit, LLC	Yourkit Profiler	Call graphs	-	n/a	n/a	-	✓	-	✓	
	[191] Weidendorfer	Callgrind, Kcachegrind	Call graphs	-	n/a	n/a	-	✓	-	✓	
Active Learning	[28] Angluin	L* (MAT approach)	Mealy machine	✓	✓	✓	-	-	✓	-	
	[92] Isberner	TTT algorithm	Mealy machine	✓	✓	✓	-	-	✓	-	
	[188] Volpato	Approximate L*	LTS	✓	✓	±	-	-	✓	-	
	[50] Cassel	Symbolic L*	Register automata	✓	✓	✓	-	-	✓	✓	
Dynamic Analysis Passive Model Learning	[24] Alalfi	PHP2XMI	UML SD	± <sup>sd</sup>	✓	✓	-	-	-	± <sup>sh</sup>	
	[152] Oechsle	JAVAVIS	UML SD	± <sup>sd</sup>	✓	✓	-	-	✓	± <sup>sh</sup>	
	[43] Briand	Meta models / OCL	UML SD	± <sup>sd</sup>	✓	✓	-	-	✓	± <sup>sh</sup>	
	[42] Briand	Meta models / OCL	UML SD	± <sup>sd</sup>	✓	✓	-	-	✓	± <sup>sh</sup>	
	[103] Labiche	Meta models / OCL	UML SD	± <sup>sd</sup>	✓	✓	-	-	✓	± <sup>sh</sup>	
	[176] Systä	Shimba	SD variant	± <sup>sd</sup>	✓	✓	-	-	✓	± <sup>sh</sup>	
	[190] Walkinshaw	MINT	EFSM	✓	✓	✓	-	-	✓	✓ <sup>d</sup>	
	[38] Beschastnikh	CSight	CFSM	✓	✓	✓	-	-	✓	✓	
	[20] Ackermann	Behavior extraction	UML SD	± <sup>sd</sup>	✓	n/a	-	± <sup>p</sup>	-	± <sup>sh</sup>	
	[61] De Pauw	Execution patterns	Execution Patterns	-	n/a	n/a	-	± <sup>f</sup>	± <sup>x</sup>	-	✓
	[85] Heule	DFAstat	DFA	-	n/a	✓	-	± <sup>f</sup>	-	-	-
	[37] Beschastnikh	Synoptic	FSM	✓	✓	✓	-	-	-	-	-
	[150] Nevill-Manning	Sequitur	Grammar	✓	✓	✓	-	-	-	± <sup>n</sup>	-
	[171] Siyari	Lexis	Lexis-DAG	✓	✓	✓	-	-	-	± <sup>n</sup>	-
	[96] Jonyer	SubdueGL	Graph Grammar	✓	✓	✓	-	-	± <sup>g</sup>	± <sup>n</sup>	± <sup>t</sup>
[54] Chow	Mystery Machine	Causal Model	✓	✓	✓	-	✓	-	-	-	
Process Mining Process Discovery	[18] Aalst, van der	Alpha algorithm	Petri net	✓	-	-	-	✓ <sup>a</sup>	✓	✓	-
	[16] Aalst, van der	2-step approach (Regions)	Petri net	✓	± <sup>s</sup>	-	-	✓ <sup>a</sup>	✓	✓	-
	[35] Bergenthum	Regions of Languages	Petri net	✓	± <sup>s</sup>	-	-	✓ <sup>a</sup>	✓	✓	-
	[192] Weijters	Flexible heuristics miner	Heuristics net	✓	± <sup>s</sup>	-	✓	✓ <sup>a</sup>	✓	✓	-
	[10] Alves de Medeiros	Genetic miner	Heuristics net	✓	± <sup>s</sup>	-	✓	✓ <sup>a</sup>	✓	✓	-
	[33] Augusto	Structured miner	BPMN	✓	-	-	✓	✓ <sup>a</sup>	✓	✓	-
	[193] Werf, van der	ILP miner	Petri net	✓	± <sup>s</sup>	✓	-	✓ <sup>a</sup>	✓	✓	-
	[204] Zelst, van	ILP with filtering	Petri net	✓	± <sup>s</sup>	✓	✓	✓ <sup>a</sup>	✓	✓	-
	[49] Carmona	Genet	Petri net	✓	-	✓	✓	✓ <sup>a</sup>	✓	✓	-
	[162] Redlich	Constr. Competition miner	BP(MN)	✓	✓	-	✓	✓ <sup>a</sup>	✓	✓	-
	[46] Buijs	ETM miner	Process tree	✓	✓	-	✓	✓ <sup>a</sup>	✓	✓	-
	[130] Leemans, S.J.J.	Inductive miner (IM)	Process tree	✓	✓	✓	✓	✓ <sup>a</sup>	✓	✓	-
	[75] Günther	Fuzzy miner	Fuzzy model	-	n/a	n/a	-	± <sup>m</sup>	✓	-	± <sup>n</sup>
	[94] Bose	Two-phase discovery	Fuzzy model	-	n/a	n/a	-	± <sup>m</sup>	✓	-	± <sup>n,u</sup>
	[76] Fluxicon / Günther	Disco	Fuzzy model	-	n/a	n/a	-	± <sup>m</sup>	✓	-	-
	[51] Celonis GmbH	Celonis	Fuzzy model	-	n/a	n/a	-	± <sup>m</sup>	✓	-	-
	[72] Gradient ECM	Minit	Fuzzy model	-	n/a	n/a	-	± <sup>m</sup>	✓	-	-
	[97] Kalenkova	TS with catching event	Reset WF-net	✓	± <sup>s</sup>	-	-	✓ <sup>a</sup>	✓	± <sup>c</sup>	± <sup>1</sup>
	[2] Aalst, van der	Generic post-processing	Reset WF-net	✓	-	-	-	✓ <sup>a</sup>	✓	± <sup>c</sup>	± <sup>1</sup>
[55] Conforti	BPMN miner	BPMN	✓	✓	-	✓	± <sup>b</sup>	✓	± <sup>c</sup>	± <sup>n</sup>	
[139] Mannhardt	Guided Process Discovery	Petri net	✓	✓	✓	✓	✓ <sup>a</sup>	✓	± <sup>u</sup>	-	
<b>This Thesis</b>		Hierarchical Discovery	Ex. Process Tree*	✓	✓	✓	✓	✓ <sup>a</sup>	✓	± <sup>c</sup>	✓

<sup>a</sup> Aligning an event log and a process model enables advanced performance and conformance analysis [21, 130].

<sup>b</sup> Analysis is based on alignments via a translation from BPMN to Petri nets, but subprocesses are not supported.

<sup>c</sup> Cancellation is a non-local control-flow construct.

<sup>d</sup> Data guards can indirectly express non-local constructs.

<sup>f</sup> Frequency-only; performance analysis is not supported.

<sup>g</sup> Loop constructs are not supported.

<sup>m</sup> Various log-based process metrics have been defined, such as frequency, significance, and correlation [75].

<sup>n</sup> Submodels have no names (i.e., only cluster hierarchy).

<sup>p</sup> Performance-only; frequency analysis is not supported.

<sup>q</sup> Cancellation not based on control-flow but on data perspective.

<sup>r</sup> Performance analysis is lost when recursion is modeled.

<sup>s</sup> Under certain conditions, soundness [33, 58] or relaxed soundness [206] can be guaranteed with pre- and post-processing.

<sup>sd</sup> Formal semantics are available for UML SD variants.

<sup>sh</sup> Properly nested SD activations can imply a hierarchy.

<sup>t</sup> Only tail recursion is supported.

<sup>u</sup> Hierarchy based on user-provided models.

<sup>x</sup> Choice constructs are not supported.

<sup>1</sup> Cancellation support is limited to one cancellation region; combinations of multiple constructs are not supported.

\* Extended Process Tree, as introduced in Part II.

ing, exception handling), it is difficult or impossible to statically determine the actual software behavior. There is little support for named submodels and cancelation patterns in static analysis. Well-structured sequence diagrams with properly nested activations could be seen as a named submodel hierarchy [101, 166], but this suggested hierarchy is not exploited further for either analysis or as an aid for the user to navigate the modeled complexity (e.g., zoom, collapse, expand, etc.). Although some related works, like [27], do consider exception handling, this information is not lifted to the concepts of cancelation behavior. In addition, since static techniques either (symbolically) unfold or do not explore references like function calls, these techniques lack support for recursive behavior.

### Monitoring and Profiling

Profilers are widely accepted and used in developing software and are extremely important for understanding program performance and behavior. Profiler-driven program analysis on Unix dates back to 1973 when Unix systems included a basic tool called *prof* [180], which listed each function and how much of program execution time it used. In 1982, *gprof* extended the concept to a complete call graph analysis [73]. In 2004, the *gprof* papers were amongst the 50 most influential PLDI (Programming Language Design and Implementation) papers for the 20-year period ending in 1999 [141, 197].

Despite their long history, popularity, and ease of use, profilers [73, 136, 191] have several shortcomings. The notion of call graphs used by profilers provides a hierarchical view of the executed program behavior, but no formal execution semantics support these models. That is, they give a hint or cross-section of the execution call stack, but provide little control-flow context (e.g., iterations, branches, etc.). Similarly, cancelation behavior (e.g., exception handling) is also not exploited. And despite the hierarchical view with named submodels, no recursion is supported, i.e., any recursive behavior is presented in an unfolded manner. In addition, only one execution or trace is considered, limiting the analysis capabilities. In particular, it is difficult to derive statistically sound conclusions about the observed performance.

In contrast to profilers, monitoring frameworks like Kieker [89] forgo detailed insights like execution call stacks. Instead, these monitoring techniques attempt to provide statistically sound performance insights by recording and using multiple executions or traces.

### Active Learning

The most efficient active learning algorithms used today are based on the seminal work on  $L^*$  by Angluin, published in 1987 [28, 183]. Angluin showed that finite automata can be learned using so-called *membership* and *equivalence queries*. Angluin's  $L^*$  approach is based on the *minimally adequate teacher*

(*MAT*) framework. In this framework, learning is viewed as a game in which a *learner* has to infer the behavior of an unknown state machine by asking queries to a *teacher* [28, 183]. The teacher knows the state machine, which is typically a *Mealy machine*, i.e., a deterministic finite-state transducer whose output values are determined both by its current state and the current inputs [196]. A learner uses *membership queries* to learn the output response to a given input sequence and build a hypothesis model, and uses *equivalence queries* to ask if a hypothesized (learned) Mealy machine is correct. If not, the teacher will respond with a counter example.

For a specific subset of behavior, active learning can potentially learn all the stateful behavior of an observed piece of software. Recent improvements, like the TTT algorithm [92], are much more efficient than the original  $L^*$ . However, in practice, active learning algorithms still have difficulty dealing with non-deterministic and timing-based behavior, i.e., non-Mealy machine behavior [183]. Recent advances tried to lift some of these restrictions by using *register automata*, a subclass of extended finite state machines [188], and approximating behavior to learn non-deterministic *labeled transition systems (LTS)* [50]. To the best of our knowledge, no support exists in the active learning community for named submodels, recursive behavior, or cancellation behavior. In addition, one should realize that, even with these recent advances, active learning: 1) requires a controlled and intrusive setup, where a learner algorithm can interact with the software, and 2) cannot learn the behavior and knowledge captured in the environment and clients using the software under study. In practice, this means that a learner algorithm cannot always be deployed, especially in an industrial (legacy) system, and that valuable context information from the environment and clients is lost.

### Passive Learning

In the school of passive learning, one passively observes the software system on the run, capturing tracing data, for example in a log. From this tracing data, an algorithm can extract a behavioral model. Many approaches use some kind of software instrumentation [24, 38, 42, 43, 103] or debugger interface [152, 176], although various monitoring and logging approaches [20, 38, 54, 61, 85, 96, 150, 171, 190], and even some supplementary source code analysis [103], are also used.

A large number of passive learning techniques focus on obtaining a rich but flat control-flow model, either in the form of an UML Sequence Diagram (UML SD), or a form of finite automata or state machine. A lot of effort has been put in enriching such models with more accurate choice and loop information [103], guards [42, 43, 190], and other predicates [20]. Well-structured sequence diagrams with properly nested activations could be seen as a named submodel hierarchy, but this suggested hierarchy is not exploited further for ei-

ther analysis or as an aid for the user to navigate the modeled complexity (e.g., zoom, collapse, expand, etc.). In the field known as grammar inference, there has been some work on inferring hierarchies and recursive patterns [96, 150, 171], however, these yield mostly nameless clusters and can handle only tail recursions. In short, there is little support for named submodels and recursive behavior, and no support for cancelation behavior.

Most passive learning techniques only focus on obtaining the control-flow model, and ignore the application of these models for tasks like analysis. Notable exceptions are the use of frequencies by [61], the analysis of network timings and performance by [20], and the extensive performance analysis and critical path or slack analysis by [54].

### Process Discovery

The idea to discover processes was first introduced in 1998 by Agrawal, who focused on processes in the context of (business) workflow management [23]. Around the same time, Cook investigated various notions of process discovery and model learning in the context of software engineering [56]. The seminal work on the  $\alpha$ -algorithm by Van der Aalst, published in 2004, can be seen as the first algorithm to truly capture concurrency in business processes [18]. The  $\alpha$ -algorithm exploits the directly-follows relation and derived log-based ordering relations to discover a Petri net. Much of the later work on process discovery is, at least in some way, inspired by the  $\alpha$ -algorithm and its use of Petri nets and directly-follows abstractions. In contrast to the variety of input sources found in software engineering, most process discovery techniques simply rely on an event log in the XES format.

Many different techniques have been proposed, based on, for example, various heuristics [2, 33, 55, 97, 162, 192], inductive logic programming [193, 204], state-based regions [16, 49], language-based regions [35], genetic approaches [10, 46], user-guided approaches [94, 139] and inductive discovery [130]. Most of these techniques yield models with formal execution semantics. In contrast, most commercial process discovery approaches [51, 72, 76] are based on the fuzzy miner approach [75], favoring a (visually) simple model by trading formal semantics for the less precise fuzzy semantics.

Despite the variety of approaches, soundness and error-free guarantees remain tricky. Region-based techniques, including inductive logic programming [193, 204], state-based regions [16, 49], and language-based regions [35], typically guarantee weak soundness (see page 29). For inductive logic programming, relaxed soundness (see page 29) can be guaranteed if there is a unique start and end symbol for all the traces in the event log [206]. The output of state- and language-based techniques can use the technique from [58] to generate a corresponding Petri net, which might be forced to be a sound workflow net under the unique start and end symbol assumption. The models produced

by heuristic miners [10, 192] can be restructured to sound models using the technique from [33]. In addition, process tree based approaches guarantee the discovery of sound and error-free models by construction [46, 130]. These approaches trade some expressivity for soundness by using process trees, i.e., the subclass of block-structured workflow nets. Often, simple block-structured models are too restrictive, and many techniques support the discovery of non-local control-flow like long-distance dependencies [10, 16, 33, 35, 49, 192, 193, 204]. These techniques typically introduce additional Petri net places, i.e., additional constraints, to capture non-local control-flow dependencies. However, the introduction of such constructs can result in unsound models.

An interesting type of non-local control-flow is the cancellation region, which allows to “cancel” or stop executing a part of the model (i.e., the part inside the cancellation region), and start executing an alternative control-flow. Although cancellation has been defined as an official workflow pattern [15], only a few approaches provide some support [2, 55, 97]. However, existing cancellation discovery is only limited to one cancellation region [2, 97], or is based on a data perspective instead of control-flow [55]. Hence, there is a lack of support for more general *cancellation and reset behavior discovery* (Challenge 4).

Other interesting modeling features are named hierarchies and recursion. Different approaches have recognized the need for layered models using some form of clustering, grouping, or subprocesses [55, 75, 94, 139]. These approaches use a form of hierarchy to reduce the modeled complexity and provide the user with a model in terms of more high-level concepts. However, these approaches either rely on user input to define the hierarchies [94, 139], or provide only nameless clusters [55, 75]. Hence, there is a lack of support to discover *named submodels* based on the available event data, without relying on manually created abstraction patterns (Challenge 2). In addition, although modeling languages like BPMN have recognized the need for reusable, recursively defined subprocesses, none of the discovery techniques support *recursion or stack behavior* (Challenge 3). Especially when modeling software behavior, proper support for named submodels and recursive behavior is a must.

With the exception of the fuzzy miner approaches, most of the process discovery techniques yield models that can be directly used for advanced performance, conformance, and enhancement via the work on *alignments* [21].

Our hierarchical performance analysis approach (Contribution 4, Chapter 8) is also based on alignments. See Chapter 8 for a detailed explanation.

### 3.2.3 Discovery Assumptions and Model Expressivity

There are some subtle differences in assumptions between the schools of model learning and process discovery and the type of models they can learn or discover. Table 3.2 summarizes the fundamental differences based on the *main-*



**Table 3.2:** Comparison of active learning, passive learning, and process discovery on the *mainstream* assumptions and expressivity.

	Non-determinism	Duplicate Labels	Noise Handling	Concurrency	Loops	Loop Identification
<b>Active Learning</b>	± <sup>a</sup>	✓	n/a	-	✓	Smallest hypothesis model
<b>Passive Learning</b>	✓	✓	-	± <sup>c</sup>	± <sup>1</sup>	Post processing
<b>Process Discovery</b>	± <sup>d</sup>	± <sup>d</sup>	✓	✓	✓	A Priori

**Note:** This table presents the mainstream assumption and expressivity, giving a feeling for how the schools of techniques differ. As noted below, there are always exceptions to the rule.

<sup>a</sup> Non-determinism is only supported by approximative extensions like [188].

<sup>c</sup> Concurrency is supported in some cases by relying on, for example, separate logs [38] or a posteriori SD fragment annotation [42].

<sup>d</sup> Non-determinism and duplicate labels are supported via preprocessing techniques like label-splitting [10, 16, 44, 47, 84, 137].

<sup>1</sup> Loops are supported in some cases by relying on, for example, a posteriori state merging [190] or a posteriori SD fragment annotation [42].

*stream* assumptions and expressivity. The goal of this discussion is to give a feeling of how the different schools approach the problems; there are always exceptions to the rule.

The main differences are based on alphabet abstractions. In process discovery, there is the general assumption that every activity should occur only once in the process model, i.e., every activity is one transition. The places between transitions represent the constraints on the ordering between the activities. This assumption is also reflected in the frequently used directly-follows graph, where each activity is a vertex. As a consequence, it is natural to detect concurrency and loops as different constraints on the ordering between the activities. The same assumption allows for effective noise filtering on the alphabet abstractions, i.e., (frequency-based) filtering is performed over structures like directly-follows relations, and not directly on the event log level. The cases where this “every activity is one transition” assumption does not hold are referred to as the *duplicate label problem* in process mining. Duplicate labels can cause inappropriate generalizations like loops, yielding unintuitive and imprecise models. Some discovery algorithms can refine labels during model construction to some extent [10, 16, 44, 47, 84]. In addition, event log preprocessing techniques like label-splitting can solve this duplicate label problem more generally by smartly relabeling the problematic events [137].

In contrast, passive learning does not know the duplicate label problem. In the case where multiple events with the same label occur one after another, passive learning algorithms will initially model the repetitive behavior in an unfolded manner, i.e., no loop abstractions are applied. After that, state merging algorithms like the red-blue fringe algorithm try to find recurring patterns and merge or collapse them, thereby introducing loops and generalizing the model [85, 190]. The upside of this approach is that these techniques do not suffer from the duplicate label problem. The downside is that they can suffer from long traces, as the generalization is only performed after the full model (usually a complete prefix tree) has been generated. Since passive learning focusses largely on sequential, state-based behavior, concurrency support is under-represented. There are some techniques for distributed software that rely on annotated or separated logs [38], but in other cases, one relies on a posteriori sequence diagram (SD) fragment annotation based on, for example, the source code [42].

In this comparison, active learning is a bit of a special case. Like passive learning, it does not introduce loops when that generalization seems inappropriate, i.e., it does not suffer from the duplicate label problem. Like process mining, it does try to introduce loops as early as possible. This “as early as possible” strategy is a direct consequence of the iterative query strategy and smallest hypothesis model employed by the  $L^*$  approach [28]. A side effect of the way this query plus hypothesis strategy work, is that the learning of non-determinism is only supported by approximative extensions like [188]. However, one should realize that active learning: 1) requires a controlled and intrusive setup, where a learner algorithm can interact with the software, and 2) cannot learn the behavior and knowledge captured in the environment and clients using the software under study. In practice, this means that a learner algorithm cannot always be deployed, especially in an industrial (legacy) system. In addition, valuable context information from the environment and clients may be lost.

The differences in assumptions and alphabet abstractions between process discovery and passive learning have a direct effect on the scalability of these techniques with respect to alphabet size, trace length, and number of traces (i.e., the algorithmic scalability). We will explore this scalability aspect in more detail in Chapter 5.

### 3.3 Scalability – Hierarchical Decomposition and Submodels

As argued in Section 1.3, there is a need to support hierarchical, recursive, and cancelation structures in process mining. This need is driven by both the type of behavior logged (Challenges 2 and 4), as well as the need for algorithmically and visually scalable techniques (Challenge 8). Moreover, as stated in

**Table 3.3:** Comparison of techniques for hierarchical decomposition and submodel identification in various process mining techniques targeted at algorithmic and visual scalability.

	Author	Algorithm / Technique	Submodels can be derived from								
			Log preprocessing	Process discovery	Model postprocessing	Conformance checking					
Algorithmic Scalability	[3] Aalst, van der	Decomposed, passages	-	✓	✓	-	Passages decomposition	-	-	-	✓
	[4] Aalst, van der	Decomposed, generic	-	✓	✓	-	Decomposition (log, model)	✓	-	✓	-
	[186] Verbeek	Decomposed ILP	-	✓	-	-	Decomposition (log, model)	✓	-	✓	-
	[88] Hompes	Decomposed, clusters	-	✓	-	-	Decomposition (log, model)	✓	-	✓	-
	[158] Polyvyanyy	Refined Process Structure Tree	-	-	-	✓	Graph structure (SESE, RPST)	-	-	-	✓
	[144] Munoz-Gama	Partitioned conformance	-	-	✓	✓	Partitions (topology, RPST)	-	-	-	✓
	[145] Munoz-Gama	SESE-based conformance	-	-	✓	✓	Graph structure (SESE, RPST)	-	-	-	✓
	[130] Leemans, S.J.J.	Inductive Miner family	-	✓	✓	-	Process tree (control-flow)	✓	-	✓	-
Visual Scalability	[75] Günter	Fuzzy miner	-	✓	-	-	Anonymous clusters	-	-	-	✓
	[93] Bose	Abstraction patterns	✓	-	-	-	Abstraction patterns	✓	-	-	-
	[94] Bose	Two-phase discovery	✓	✓	-	-	Cluster hierarchy	-	-	-	✓
	[55] Conforti	BPMN miner	-	✓	-	-	Subprocesses, cancelation	✓	✓	-	-
	[139] Mannhardt	Guided Process Discovery	✓	✓	-	-	Activity (abstraction) patterns	✓	-	-	-
	[97] Kalenkova	Cancelation region	-	✓	-	-	Cancelation regions	-	-	✓	✓
	[69] Gonzalez	Database process mining	✓	-	-	-	Business instance object relations	✓	✓	-	-
	<b>This Thesis</b>	Recursion Aware Discovery	✓	✓	-	-	Subprocesses, hierarchies	✓	-	✓	-
	Cancelation Discovery	-	✓	-	-	Cancelation regions	✓	-	✓	-	
	Distributed software	✓	✓	-	-	Distributed system submodels	✓	✓	✓	-	

Contribution 4, we not only consider process discovery, but also performance analysis based on a given hierarchy. Therefore, this section will survey scalability in related work and the use of hierarchies, hierarchical decomposition, and submodels in the broader process mining perspective.

Frequently, hierarchical decomposition is used to increase the algorithmic scalability of discovery and conformance techniques with respect to alphabet size, trace length, and number of traces (i.e., the algorithmic scalability). In other cases, hierarchical abstractions and submodels are used to reduce the modeled complexity and provide the user with a model in terms of more high-level concepts (i.e., the visual scalability). Below, we will discuss a selection of related work dealing with both algorithmic scalability (Section 3.3.1) and visual scalability (Section 3.3.2) as is summarized in Table 3.3.

### 3.3.1 Hierarchical Decomposition for Algorithmic Scalability

Related work has explored the use of various hierarchical decompositions to improve the algorithmic scalability of both process discovery and conformance checking. In fact, related work suggests that process discovery and conformance checking are related problems, and similar techniques could be applied to both problems [4]. In this section, we will first discuss several (generic) decomposition techniques. After that, we will discuss two more concrete techniques: the RPST-based techniques, and the Inductive Miner family.

Several decomposed process mining techniques have been proposed. One of the earlier decomposition techniques leveraged the notion of *passages* to

decompose both process mining and conformance checking [3]. A passage is a pair  $(X, Y)$  where  $X$  and  $Y$  are sets of activity nodes (i.e., events in a log or transitions in a model), such that: 1) the activity nodes in  $X$  influence the enabling of the activity nodes in  $Y$ , and 2) the activity nodes in  $Y$  are influenced by the activity nodes in  $X$ . Observe that this notion of passages is quite strict, hence the decomposition is fairly limited. Therefore, a generic approach was proposed to lift many of the restrictions imposed by the notion of passages [4]. This generic approach decomposes process discovery by *partitioning/clustering activities*, creating sublogs, and decomposes conformance checking by using *valid decompositions*. A decomposition of a Petri net is valid if: 1) the subnets “agree” on the original labeling function (i.e., the same transition always has the same label), 2) each place resides in just one subnet, 3) each invisible transition resides in just one subnet, and 4) if there are multiple transitions with the same label, they should reside in the same subnet. Both the passages and generic approaches rely on a decomposition or partitioning/clustering of transitions in the model c.q. activities in the log. Due to the focus on activities in the definition of passages and valid decompositions, these techniques rely mostly on the log causality and model semantics. Concrete clustering [88] and discover [186] approaches based on the generic framework from [4] have been proposed. However, despite the huge gain in algorithmic scalability and general applicability, the underlying hierarchical structure is not leveraged for the visual scalability of discovered models or detected conformance issues.

RPST-based approaches rely on the underlying graph structure to improve the algorithmic scalability of conformance checking. By analyzing (topological) graph structures, clusters of vertices can be derived and structured in a so-called Refined Process Structure Tree (RPST) [158]. Several improvements based on RPSTs have been realized over the years [144, 145]. However, these approaches only speed up conformance checking and provide no semantical hierarchical decomposition. In addition, the underlying hierarchical structure is not leveraged for the visual scalability of detected conformance issues.

The Inductive Miner family provided the first scalable process discovery technique that guaranteed soundness [130]. The Inductive Miner discovery algorithm leverages a relation between semantical structures in a directly-follows graph and the operators in a process tree. That is, a hierarchical decomposition of patterns in a directly-follows graph is related to the hierarchical structure of a process tree; each pattern corresponds to a process tree operator. Later, a novel, scalable conformance checking technique based on activity projections was introduced, leveraging the process tree structure [130, 134]. For the visual presentation, the underlying hierarchical structure of a process tree is indirectly leveraged to provide structured process models. That is, the approach yields visually simple models due to the restricted and well-structured process trees. Although this provides some visual scalability, this underlying hierarchi-

cal structure is not leveraged for the visual scalability of detected conformance issues or models with many (low-level) activities.

The Inductive Miner discovery approach will be discussed in Chapter 4. In Part II, we will further explore the notion of “patterns corresponding to process tree operators” to formulate novel process discovery algorithms.

### 3.3.2 Hierarchies and Submodels for Visual Scalability

Different approaches have recognized the need for visual scalability in discovered processes. One of the earlier approaches relied on the notion of user-defined abstraction patterns to avoid many low-level activities and reduce the visible complexity [93]. Various improvements have been proposed to help the user in detecting abstraction patterns [94]. The fuzzy miner automated this abstraction detection process and provided a slider-based approach to allow the user to zoom between different levels of abstractions, thus reducing the number of modeled activities [75]. In contrast, the BPMN miner attempts to automatically infer subprocesses based on relations in the data perspective instead of the control-flow perspective. However, an important downside of these techniques is that they provide only nameless clusters. That is, although the visible complexity is reduced through clustering, the understandability of the models is also reduced.

Recent work attempts to discover more meaningful hierarchical structures. In the Guided Process Discovery approach, user-defined *named* abstraction patterns are used to obtain a hierarchical model with named subprocesses [139]. Here, these named submodels reduce the visual complexity. In the technique from [97], the discovery algorithm attempts to find a cancellation region to reduce the modeled complexity.

Another approach is to employ domain knowledge to find relations in the event log. For example, in the technique from [69], business instance object relations are inferred from the underlying database structure. These relations can be used to construct a hierarchical decomposition of activities, and thus yield named submodels.

Conclusively, the need for visual scalability based on hierarchies has been recognized in various forms, but no existing work presented a generic and automatic discovery approach that yields understandable and visually scalable models with minimal user input (Challenge 8).

## 3.4 Performance Analysis

The area of performance analysis is very varied, both in the process mining and software engineering community. Our performance analysis approach (Contribution 4, Chapter 8) is especially useful for analyzing software system

processes, and is generic enough to apply to any kind of operational or software process. Therefore, we will compare a selection of both communities as is summarized in Table 3.4.

### 3.4.1 Criteria for Comparison

For the comparison, we defined a set of feature criteria. To be able to perform meaningful analyses on a discovered or learned model, it is important that the underlying model notation has clear, well-defined, *formal execution semantics*. Clear semantics is a prerequisite for reliable interpretation by both man and machine. Without clear semantics, we cannot reliably apply performance and conformance analysis. Secondly, the analysis should be (statistically) significant, i.e., the technique should be able to *aggregate* information over multiple execution *runs*. It is typically impossible to make reliable statements about normal and noisy behavior and performance given only a small set of observations [8]. At the very least, some support for *frequency* (usage) and *performance* (timing) analysis should be supported. Finally, the technique should support *submodels* and “drill-down” type of analysis [8]. That is, the techniques need to support frequency and performance analysis on various levels of abstractions, thereby increasing the visual scalability and aiding the user in understanding complex data (see Challenge 8 on page 16).

### 3.4.2 Discussion of Related Work

In the software engineering domain, profiler type of techniques [61, 73, 89, 136, 191] typically include some notion of submodels via call graphs. This gives a hint of total times and frequencies across a cross-section of the execution call stack. However, there is no control-flow context to put the numbers in perspective, i.e., there is no support for semantic-aware, model-based analysis. In addition, most profilers only look at one execution or trace. As noted before, by relying on only a single observation, one is at risk of making unreliable, statistically unsound statements.

Model-based techniques [20, 37, 54] typically focus more on the control-flow aspect. These techniques project performance results onto actual control-flow models. Multiple observations are aggregated, thereby supporting statistically significant performance analyses. However, these techniques have limited or no support for performance analysis on submodels and “drill-down” type of analysis. That is, they work well for small models, but are not visually scalable.

In the process mining domain, related work can be divided into three groups: 1) visual analytics without a model, 2) fuzzy model-based analysis, and 3) formal model-based analysis.

Performance analysis techniques based on visual analytics provide insights into the recorded behavior without using a model. Various event log based

**Table 3.4:** Comparison of related performance analysis techniques in software engineering and process mining.

	Author	Algorithm / Technique	Formal Semantics	Aggregate Values	Frequency Runs	Submodel Support	Performance Info
Software Engineering	[20] Ackermann	Behavior extraction	± <sup>1</sup>	✓	-	✓	-
	[37] Beschastnikh	Synoptic	✓	✓	✓	-	-
	[89] Van Hoorn	Kieker Framework	-	✓	✓	✓	-
	[54] Chow	Mystery Machine	✓	✓	-	✓	-
	[177] Szvetits	Reusable Event Types	-	✓	✓	✓	-
	[61] De Pauw	Execution patterns	-	-	✓	-	± <sup>2</sup>
	[73] Graham	gprof profiler	-	-	-	✓	± <sup>2</sup>
	[136] YourKit, LLC	YourKit Profiler	-	-	✓	✓	± <sup>2</sup>
	[191] Weidendorfer	Callgrind, KCachegrind	-	-	✓	✓	± <sup>2</sup>
Process Mining	[172] Song	Dotted Chart	-	✓	✓	✓	-
	[87] Hompes	Context-aware KPI	-	✓	✓	✓	-
	[157] Piessens	Replay Performance	✓	✓	✓	✓	-
	[75] Günther	Fuzzy miner	-	✓	✓	✓	-
	[76] Fluxicon / Günther	Disco	-	✓	✓	✓	-
	[51] Celonis GmbH	Celonis Process Mining	-	✓	✓	✓	-
	[72] Gradient ECM	Minit	-	✓	✓	✓	-
	[13] Van der Aalst	EMiT, timed WFnets	✓	✓	-	✓	-
	[165] Rogge-Solti	Stochastic Petri nets	✓	✓	-	✓	-
	[22] Adriansyah	Robust Performance	✓	✓	✓	✓	-
[21] Adriansyah	Alignments	✓	✓	✓	✓	-	
	<b>This Thesis</b>	Hierarchical Performance Analysis	✓	✓	✓	✓	✓

<sup>1</sup> Formal semantics are available for some UML sequence diagram variants.<sup>2</sup> Aggregate values at submodels only, no semantic-aware, model-based analysis.

techniques have been proposed, like, for example, the Dotted Chart [172] and the work on context-based KPIs [87]. The lack of a model can be both an advantage and a disadvantage. On the upside, performance analysis is not influenced by inaccurate or erroneous models. However, at the same time, there is little control-flow context to put the numbers in perspective.

Fuzzy model-based performance analysis relies on techniques like (approximative) replay [157] and various log-based process metrics such as frequency, significance, and correlation [75]. Many commercial tools provide fuzzy model-based performance analysis, favoring a (visually) simple model by trading formal semantics for the less precise fuzzy semantics [51, 72, 76]. However, fuzzy or imprecise semantics can yield unreliable performance and conformance analysis results. This is especially true in processes with concurrency, where the actual control-flow interpretation is often unclear [8].

Various formal model-based performance analysis techniques have been proposed. Some approaches focus on embedding performance characteristics into the model, yielding, for example, timed workflow nets [13] or stochastic

Petri nets [165]. Other techniques focus on robust performance analysis [22] and establishing a precise alignment between event logs and process models [21]. These techniques typically provide very precise performance insights and can handle complex, formal process models such as arbitrary Petri nets. However, these precise insights come at the cost of computation time (see also Section 3.3).

In all of the above techniques, the performance analysis focusses only on the level of individual activities or events. That is, they work well for small, flat models, but are not visually scalable. In contrast, our hierarchical performance analysis approach (Contribution 4, Chapter 8) aims at improving the visual scalability by leveraging hierarchies.

### 3.5 Conclusion

In this chapter, we presented a broad and systematic survey of both the reverse engineering and process mining fields and touches upon related hierarchical techniques. We have presented a taxonomy covering both fields, and discussed the similarity and differences between a selection of *model learning* and *process discovery*. In addition, we surveyed the scalability in related work and the use of hierarchies, hierarchical decomposition, and submodels in the broader *process mining* perspective. Furthermore, we discussed various aspects of performance analysis in both fields, and investigate the use of hierarchies and submodels in the context of performance analysis.

In the next chapter, we will look at the Inductive Miner process discovery technique from [130] in more detail.





“It’s still magic even if you know how it’s done.”

— Terry Pratchett, *A Hat Full of Sky*

## 4 | A Process Discovery Foundation

In this thesis, we will present hierarchical, recursion aware, and cancelation discovery techniques based on process trees. For these techniques, we will build on an existing process discovery foundation to discover process trees, and extend the modeling notation and discovery technique where needed. We selected the well-known Inductive Miner (IM) framework, as described in [130], as our process discovery foundation. This framework enables a family of concrete process discovery techniques. In addition, the IM framework offers good discovery guarantees, scales well, and provides clear extension points for our adaptations.

In this chapter, we will give an introduction to the IM framework, covering the problems this framework addresses (Section 4.1) the discovery algorithm (Section 4.2) and its properties and guarantees (Section 4.3). In addition, we will cover how this framework can be extended (Section 4.4) and some of the open challenges (Section 4.5).

### 4.1 Problem Statement

The goal of process discovery is, given only an event log, construct a process model describing the behavior recorded in the event log. As discussed in Section 1.1.2, process discovery is challenging for several reasons. For one, a process discovery technique should yield *sound* models with *clear and well-defined semantics*. For another, discovered models should represent the behavior recorded in the event log, i.e., the discovered model should have a high *fitness* and *precision*. In addition, the discovery algorithm should *support the type of behavior that is recorded in the event log*. E.g., when observing a process where everything is canceled upon a check fail, then the model formalism and discovery algorithm should support such behavior. Finally, the discovery algorithm should *scale well, allowing for the analysis of large event logs*.

The Inductive Miner framework addresses most of these challenges. By relying on process trees (see Section 2.2.6), the Inductive Miner discovers sound and well-defined models by construction. In addition, as we will discuss in detail in Section 4.3, the Inductive Miner guarantees perfect fitness and scales

well with large inputs. Naturally, by relying on process trees, the Inductive Miner is limited to the behavior that such trees can capture.

In Section 4.2, we will cover the basics of the Inductive Miner framework, summarizing the work in [130, 131]. However, there are several extensions to this framework [130]. The *incomplete* and *infrequent extensions* address event logs missing behavior (incomplete) or containing noise (infrequent). Special *construct extensions* allow for the discovery of interleaved behavior and the inclusive choice / OR split. Finally, the *lifecycle extension* is designed for non-atomic event logs, and the *directly-follows based approach* is designed to handle very large event logs.

## 4.2 The Inductive Miner Framework

This section covers the basics of the Inductive Miner discovery algorithm. Section 4.2.1 will give an overview of the algorithm and framework. The sections 4.2.2 through 4.2.5 will discuss key concepts and extension points in the Inductive Miner framework.

### 4.2.1 Algorithm Overview

The Inductive Miner framework discovers models using a divide and conquer approach. Given a log  $L$ , the framework searches for possible splits of  $L$  into sublogs  $L_1, \dots, L_n$ , such that these sublogs combined with a process tree operator  $\otimes \in \{ \rightarrow, \times, \wedge, \cup \}$  can (at least) reproduce  $L$  again. Later in this chapter, we will show that these operators are mutually exclusive. The framework then recurses on the corresponding sublogs, repeats the above process, and returns the discovered submodels. Empty logs, logs with empty traces or traces with only a single activity form the base cases for this framework. Note that we will assume atomic event logs for the remainder of this chapter.

The core of each recursion step in the Inductive Miner framework is the search for possible log splits and a corresponding process tree operator. The framework performs these searches using the directly-follows graph abstraction  $G(L)$  of the (sub)log  $L$ . For each process tree operator, a different graph cut or footprint is characterized based on the edges in  $G(L)$ . The framework searches for a valid non-trivial cut in the graph, selects the corresponding tree operator, and recurses. Recall from Definition 2.1.8, a cut is nontrivial if there are at least two partitions, and no partition is empty. If the log contains empty traces or no valid non-trivial cut can be found, a fallback solution will be used. In the worst case, the framework will fall back to a flower model, allowing for any behavior. Note that, by design, each activity appears only once in the produced process tree, and this tree can be a generalization of the original event log. For example, for the event log  $L = [\langle a, a, a \rangle]$ , the framework would return the model  $\cup(a, \tau)$ , and not  $\rightarrow(a, a, a)$ .

Algorithm 4.1 describes the above process using three steps:

1. Check for base cases (line 1).
2. If no base case was valid and the log does not contain empty traces, then find a valid non-trivial cut (line 5). If found, split the log (line 7) and recurse on the new sublogs (line 8).
3. If no valid base case or cut was found, apply a fallback solution (line 9).

Each of these steps is an extension point in the IM framework, allowing new base cases, cuts/splits, and fallback solutions to be defined and added to the discovery algorithm. Below, we will discuss all steps in detail. We will be using the example run in Table 4.1 as clarification. Section 4.2.2 details the standard base cases. Section 4.2.3 covers the standard cuts, and how they are related to the process tree operators. Section 4.2.4 defines log splits for each log cut. And Section 4.2.5 elaborates on the fallback options, when no valid base case or split can be found.

---

#### Algorithm 4.1: Inductive Miner (IM) Framework

---

**Input:** An event log  $L$ .

**Output:** A process tree  $Q$  such that  $L$  fits  $Q$ .

**Description:** The function `IMdiscover()` recursively tries to discover a process tree capturing (at least) the behavior in  $L$ . Note that we use  $\perp$  to model when no valid base case or valid cut was found for the given log.

```

IMdiscover( $L$ )
1   $Q_{base} = \text{BaseCase}(L)$                                      // Step 1 - Check base cases
2  if  $Q_{base} \neq \perp$  then
3      return  $Q_{base}$                                            // If a base case was found, then return
4  if  $\varepsilon \notin L$  then
5       $(\otimes, (\Sigma_1, \dots, \Sigma_n)) = \text{FindCut}(L)$          // Step 2.1 - Find a valid cut
6      if  $(\otimes, (\Sigma_1, \dots, \Sigma_n)) \neq \perp$  then
7           $(L_1, \dots, L_n) = \text{SplitLog}(L, (\otimes, (\Sigma_1, \dots, \Sigma_n)))$  // Step 2.2 - Split logs
8          return  $\otimes(\text{IMdiscover}(L_1), \dots, \text{IMdiscover}(L_n))$  // Step 2.3 - Recurse
9  return  $\text{Fallback}(L)$                                        // Step 3 - Apply fallback

```

---

### 4.2.2 Base Cases

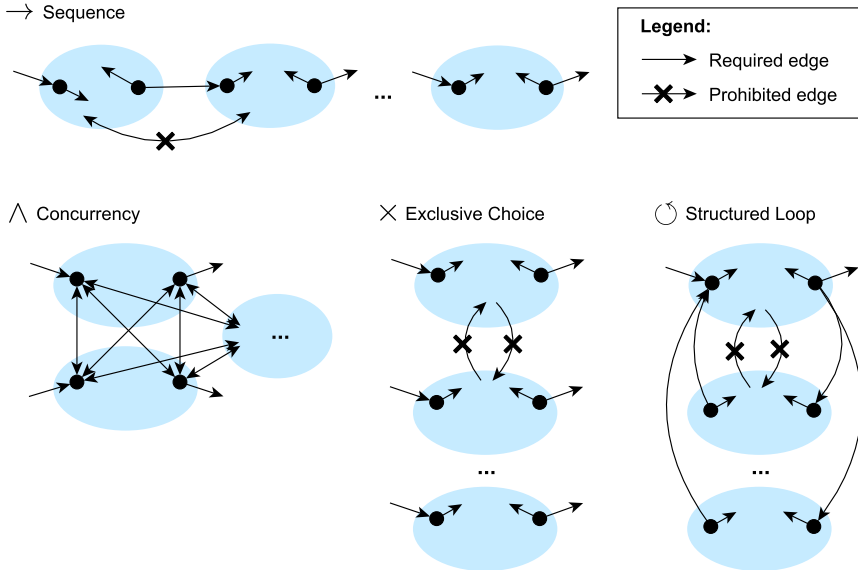
The first step in Algorithm 4.1 is to check for base cases using the function  $\text{BaseCase}(L)$ . This function returns the corresponding process tree when a base case applies or returns  $\perp$  to indicate no base case was found. These base cases provide an end to the recursion. We recognize the following base cases:

#### ■ Base Case 4.1 — Empty Log.

*Condition:*  $L = [] \vee (\forall \sigma \in L : \sigma = \varepsilon)$

*Return:*  $\tau$

*Description:* The *Empty Log* base case applies when the log contains no traces or only contains empty traces. This base case returns  $\tau$ , representing the model consisting of an empty step.



**Figure 4.1:** Cuts of the directly-follows graph for the basic Inductive Miner operators:  $\rightarrow$ ,  $\wedge$ ,  $\times$ , and  $\circlearrowleft$ . The grey areas indicate partitions; the arrows indicate required and prohibited edges characterizing the cut.

#### ■ Base Case 4.2 — Single Activity.

*Condition:*  $L \neq [] \wedge (\exists a \in \mathbb{A} : \forall \sigma \in L : \sigma = \langle a \rangle)$

*Return:*  $a$

*Description:* The *Single Activity* base case applies when the log contains only traces with a single activity  $a$ . This base case returns  $a$  as a leaf node.

### 4.2.3 Finding Cuts

The second step in Algorithm 4.1 is to search for possible log splits and a corresponding process tree operator using the function  $\text{FindCut}(L)$ . The function searches for non-trivial cuts in the directly-follows graph abstraction  $G(L)$  of the (sub)log  $L$  or returns  $\perp$  to indicate no valid non-trivial cut was found. For each process tree operator, a different cut or footprint is characterized based on the edges between nodes in  $G(L)$ . Figure 4.1 informally depicts cuts of the directly-follows graph for the basic Inductive Miner operators:  $\rightarrow$ ,  $\wedge$ ,  $\times$ , and  $\circlearrowleft$ . We formally define these directly-follows graph cuts below, assuming the event log has no empty traces (i.e.,  $\varepsilon \notin L$ ). Note that these cuts are mutually exclusive (see also Lemma 4.3.6 later in this chapter), and can be detected in any order. Recall that graph cuts, partitions  $(\Sigma_i)$  and path in a graph  $(\rightsquigarrow)$  were explained in Section 2.1.2 on page 23.

**Table 4.1:** Example Inductive Miner discovery on the log  $L = [\langle a, b, c, d, g \rangle, \langle a, b, c, e, f, e, g \rangle, \langle a, c, b, d, g \rangle, \langle a, c, b, e, g \rangle]$ . The rows illustrate how the discovery progresses step by step. The highlights indicate the parts of the log and directly-follows graph used, and relate them to the corresponding partial process tree model that is discovered. The dashed lines indicate the cuts. The corresponding Petri net model is shown at the bottom.

Event Log	Directly-Follows Graph	Discovered Model
<div style="display: flex; flex-direction: column; align-items: flex-start;"> <div style="display: flex; border: 1px solid black; padding: 2px; margin-bottom: 2px;">a b c d g</div> <div style="display: flex; border: 1px solid black; padding: 2px; margin-bottom: 2px;">a b c e f e g</div> <div style="display: flex; border: 1px solid black; padding: 2px; margin-bottom: 2px;">a c b d g</div> <div style="display: flex; border: 1px solid black; padding: 2px;">a c b e g</div> </div>		
<div style="display: flex; flex-direction: column; align-items: flex-start;"> <div style="display: flex; border: 1px solid black; padding: 2px; margin-bottom: 2px;">a b c d g</div> <div style="display: flex; border: 1px solid black; padding: 2px; margin-bottom: 2px;">a b c e f e g</div> <div style="display: flex; border: 1px solid black; padding: 2px; margin-bottom: 2px;">a c b d g</div> <div style="display: flex; border: 1px solid black; padding: 2px;">a c b e g</div> </div>		
<div style="display: flex; flex-direction: column; align-items: flex-start;"> <div style="display: flex; border: 1px solid black; padding: 2px; margin-bottom: 2px;">a b c d g</div> <div style="display: flex; border: 1px solid black; padding: 2px; margin-bottom: 2px;">a b c e f e g</div> <div style="display: flex; border: 1px solid black; padding: 2px; margin-bottom: 2px;">a c b d g</div> <div style="display: flex; border: 1px solid black; padding: 2px;">a c b e g</div> </div>		
<div style="display: flex; flex-direction: column; align-items: flex-start;"> <div style="display: flex; border: 1px solid black; padding: 2px; margin-bottom: 2px;">a b c d g</div> <div style="display: flex; border: 1px solid black; padding: 2px; margin-bottom: 2px;">a b c e f e g</div> <div style="display: flex; border: 1px solid black; padding: 2px; margin-bottom: 2px;">a c b d g</div> <div style="display: flex; border: 1px solid black; padding: 2px;">a c b e g</div> </div>		

### ■ Cut Detection 4.1 — Sequence ( $\rightarrow$ ).

*Description:* The directly-follows graph  $G$  can be partitioned with an ordered cut such that edges only flow to succeeding partitions, and not back.

*Definition:* A *sequence* ( $\rightarrow$ ) *cut* is an ordered cut  $\Sigma_1, \dots, \Sigma_n$  of a directly follows graph  $G$  such that:

1. There is a path from every  $\Sigma_i$  to  $\Sigma_j$  for  $i < j$ :

$$\forall 1 \leq i < j \leq n \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : a_i \rightsquigarrow a_j \in G$$

2. There is no path back from any  $\Sigma_j$  to  $\Sigma_i$  for  $i < j$ :

$$\forall 1 \leq i < j \leq n \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : a_j \rightsquigarrow a_i \notin G$$

3. There is an edge from every ‘exit node’  $a_i \in E(\Sigma_i)$  of partition  $\Sigma_i$  to every ‘entry node’  $a_j \in S(\Sigma_{i+1})$  of partition  $\Sigma_{i+1}$  (if such nodes exist):

$$\forall a_i \in E(\Sigma_i) \wedge a_j \in S(\Sigma_{i+1}) : (a_i, a_j) \in G$$

$$\text{Where } S(\Sigma_i) = \{ a_i \in \Sigma_i \mid j \neq i \wedge a_j \in \Sigma_j \wedge (a_j, a_i) \in G \}$$

$$\text{and } E(\Sigma_i) = \{ a_i \in \Sigma_i \mid j \neq i \wedge a_j \in \Sigma_j \wedge (a_i, a_j) \in G \}$$

*Example:* In step 1 in Table 4.1, a sequence cut is detected in the directly-follows graph, yielding four partitions:  $\Sigma_1 = \{ a \}$ ,  $\Sigma_2 = \{ b, c \}$ ,  $\Sigma_3 = \{ d, e, f \}$ , and  $\Sigma_4 = \{ g \}$ . Partitions  $\Sigma_1$  and  $\Sigma_4$  are handled by the *single activity* base case, the other partitions are handled in recursive steps 2 and 3.

### ■ Cut Detection 4.2 — Concurrency ( $\wedge$ ).

*Description:* Directly follows graph  $G$  can be partitioned with a cut such that there are edges between any two nodes of any two partitions.

*Definition:* A *concurrency* ( $\wedge$ ) *cut* is a cut  $\Sigma_1, \dots, \Sigma_n$  of a directly follows graph  $G$  such that:

1. Every partition  $\Sigma_i$  has some start and end activities:

$$\forall 1 \leq i \leq n : \Sigma_i \cap \text{Start}(G) \neq \emptyset \wedge \Sigma_i \cap \text{End}(G) \neq \emptyset$$

2. There are edges between every node  $a_i \in \Sigma_i$  and node  $a_j \in \Sigma_j$ , in both directions, for  $i \neq j$ :

$$\forall i \neq j \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : (a_i, a_j) \in G \wedge (a_j, a_i) \in G$$

*Example:* In step 2 in Table 4.1, a concurrency cut is detected in the directly-follows graph, yielding two partitions:  $\Sigma_1 = \{ b \}$  and  $\Sigma_2 = \{ c \}$ . Both partitions are handled by the *single activity* base case.

### ■ Cut Detection 4.3 — Exclusive Choice ( $\times$ ).

*Description:* Directly follows graph  $G$  can be partitioned with a cut such that there are no edges between any two partitions.

*Definition:* A *choice* ( $\times$ ) *cut* is a cut  $\Sigma_1, \dots, \Sigma_n$  of a directly follows graph  $G$  such that:

1. There are no edges between any  $\Sigma_i$  and  $\Sigma_j$  with  $i \neq j$ :

$$\forall i \neq j \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : (a_i, a_j) \notin G$$

*Example:* In step 3 in Table 4.1, a choice cut is detected in the directly-follows graph, yielding two partitions:  $\Sigma_1 = \{d\}$  and  $\Sigma_2 = \{e, f\}$ . Partition  $\Sigma_1$  is handled by the *single activity* base case, partition  $\Sigma_2$  is handled in recursive step 4.

#### ■ Cut Detection 4.4 — Structured Loop ( $\odot$ ).

*Description:* Directly follows graph  $G$  can be partitioned with a partially ordered cut such that the first partition represents the loop body, and the non-first partitions represent the loop redo options.

*Definition:* A *loop* ( $\odot$ ) *cut* is a partially ordered cut  $\Sigma_1, \dots, \Sigma_n$  of a directly follows graph  $G$  such that:

1. All start and end activities are in the body  $\Sigma_1$ :

$$Start(G) \cup End(G) \subseteq \Sigma_1$$

2. There are only edges from the end of  $\Sigma_1$  to  $\Sigma_i$  for  $i > 1$ :

$$\forall i > 1 \wedge a_i \in \Sigma_i \wedge a_1 \in \Sigma_1 : (a_1, a_i) \in G \Rightarrow a_1 \in End(G)$$

3. There are only edges to the start of  $\Sigma_1$  from  $\Sigma_i$  for  $i > 1$ :

$$\forall i > 1 \wedge a_i \in \Sigma_i \wedge a_1 \in \Sigma_1 : (a_i, a_1) \in G \Rightarrow a_1 \in Start(G)$$

4. There are no edges between nodes in  $\Sigma_i$  and  $\Sigma_j$  for  $i > 1 \wedge j > 1 \wedge i \neq j$ :

$$\forall i > 1 \wedge j > 1 \wedge i \neq j \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : (a_i, a_j) \notin G$$

5.  $\Sigma_i$ , with  $i > 1$ , has an edge to  $\Sigma_1$  iff it connects to all start activities:

$$\begin{aligned} \forall i > 1 \wedge a_i \in \Sigma_i \wedge a_1 \in Start(G) : (a_i, a_1) \in G \\ \Leftrightarrow (\exists a'_1 \in \Sigma_1 : (a_i, a'_1) \in G) \end{aligned}$$

6.  $\Sigma_i$ , with  $i > 1$ , has an edge from  $\Sigma_1$  iff it connects from all end activities:

$$\begin{aligned} \forall i > 1 \wedge a_i \in \Sigma_i \wedge a_1 \in End(G) : (a_1, a_i) \in G \\ \Leftrightarrow (\exists a'_1 \in \Sigma_1 : (a'_1, a_i) \in G) \end{aligned}$$

*Example:* In step 4 in Table 4.1, a loop cut is detected in the directly-follows graph, yielding two partitions: body  $\Sigma_1 = \{e\}$  and redo  $\Sigma_2 = \{f\}$ . Both partitions are handled by the *single activity* base case.

#### 4.2.4 Splitting Logs

Once a valid cut  $\Sigma_1, \dots, \Sigma_n$  has been found for a given process tree operator  $\otimes$ , Algorithm 4.1 splits the log according to the cut using the function  $\text{SplitLog}(L, (\otimes, (\Sigma_1, \dots, \Sigma_n)))$ . This function splits the log  $L$  into sublogs  $L_1, \dots, L_n$  such that these logs combined with the operator  $\otimes$  can (at least) reproduce  $L$  again. We formally define these log splits below for each operator cut. Recall that projection ( $\sigma|_X$ ) was explained in Definition 2.1.5 on page 23 and activity log-projection ( $\mathbb{A}(\sigma)$ ) was explained in Definition 2.3.6 on page 48.



■ **Log Split 4.1 — Sequence ( $\rightarrow$ ).**

*Description:* Each trace in the log is split into  $n$  substraces such that each subtrace corresponds to one of the  $n$  sequence cut partitions.

*Definition:* Given a sequence cut  $\Sigma_1, \dots, \Sigma_n$  and event log  $L$ :

- Sublog  $L_i$  consists of all maximal substraces with activities in  $\Sigma_i$ :

$$L_i = [\sigma_i \mid \sigma_1 \cdot \sigma_2 \cdot \dots \cdot \sigma_n \in L \wedge (\forall 1 \leq j \leq n : \mathbb{A}(\sigma_j) \subseteq \Sigma_j)]$$

*Example:* In step 1 in Table 4.1, we have the input log  $L = [\langle a, b, c, d, g \rangle, \langle a, b, c, e, f, e, g \rangle, \langle a, c, b, d, g \rangle, \langle a, c, b, e, g \rangle]$  and the sequence cut  $\Sigma_1 = \{a\}$ ,  $\Sigma_2 = \{b, c\}$ ,  $\Sigma_3 = \{d, e, f\}$ , and  $\Sigma_4 = \{g\}$ . As a result,  $L$  is split into the sublogs  $L_1 = [\langle a \rangle^4]$ ,  $L_2 = [\langle b, c \rangle^2, \langle c, b \rangle^2]$ ,  $L_3 = [\langle d \rangle^2, \langle e, f, e \rangle, \langle e \rangle]$ , and  $L_4 = [\langle g \rangle^4]$ .

■ **Log Split 4.2 — Concurrency ( $\wedge$ ).**

*Description:* Each trace in the log is projected  $n$  times, once for each of the  $n$  partitions c.q. sets of activities.

*Definition:* Given a concurrency cut  $\Sigma_1, \dots, \Sigma_n$  and event log  $L$ :

- Sublog  $L_i$  consists of all activities in  $\Sigma_i$ :

$$L_i = [\sigma \upharpoonright_{\Sigma_i} \mid \sigma \in L]$$

*Example:* In step 2 in Table 4.1, we have the input log  $L = [\langle b, c \rangle^2, \langle c, b \rangle^2]$  and the concurrency cut  $\Sigma_1 = \{b\}$  and  $\Sigma_2 = \{c\}$ . As a result,  $L$  is split into the sublogs  $L_1 = [\langle b \rangle^4]$  and  $L_2 = [\langle c \rangle^4]$ .

■ **Log Split 4.3 — Exclusive Choice ( $\times$ ).**

*Description:* Each trace in the log is assigned to the sublog corresponding to the matching choice cut partition, based on the activities in the trace.

*Definition:* Given a choice cut  $\Sigma_1, \dots, \Sigma_n$  and event log  $L$ :

- Sublog  $L_i$  consists of all the traces with only activities in  $\Sigma_i$ :

$$L_i = [\sigma \mid \sigma \in L \wedge \mathbb{A}(\sigma) \subseteq \Sigma_i]$$

*Example:* In step 3 in Table 4.1, we have the input log  $L = [\langle d \rangle^2, \langle e, f, e \rangle, \langle e \rangle]$  and the choice cut  $\Sigma_1 = \{d\}$  and  $\Sigma_2 = \{e, f\}$ . As a result,  $L$  is split into the sublogs  $L_1 = [\langle d \rangle^2]$  and  $L_2 = [\langle e, f, e \rangle, \langle e \rangle]$ .

■ **Log Split 4.4 — Structured Loop ( $\odot$ ).**

*Description:* Each trace in the log is split into substraces of the loop body and the loop redo partitions.

*Definition:* Given a loop cut  $\Sigma_1, \dots, \Sigma_n$  and event log  $L$ :

- Sublog  $L_i$  consists of all maximal substraces with activities in  $\Sigma_i$ :

$$\begin{aligned} L_i = & [\sigma_2 \mid \sigma_1 \cdot \sigma_2 \cdot \sigma_3 \in L \wedge \mathbb{A}(\sigma_2) \subseteq \Sigma_i \\ & \wedge (\sigma_1 = \varepsilon \vee (\sigma_1 = \langle \dots, a_1 \rangle \wedge a_1 \notin \Sigma_i)) \\ & \wedge (\sigma_3 = \varepsilon \vee (\sigma_3 = \langle a_3, \dots \rangle \wedge a_3 \notin \Sigma_i))] \end{aligned}$$

*Example:* The structured loop can split a trace into multiple subtraces. For example, in step 4 in Table 4.1, we have the input log  $L = [\langle e, f, e \rangle, \langle e \rangle]$  and the loop cut  $\Sigma_1 = \{e\}$  and  $\Sigma_2 = \{f\}$ . As a result,  $L$  is split into the sublogs  $L_1 = [\langle e \rangle^3]$  and  $L_2 = [\langle f \rangle]$ . Note that the first trace in  $L$  yields two subtraces for sublog  $L_1$ .

#### 4.2.5 Fallback Cases

Algorithm 4.1 should return a process tree under all circumstances. But for some input event logs, no base case can be applied and no valid cut can be found. Therefore, the third step in Algorithm 4.1 is to apply a fallback solution if all other options failed. Such a fallback solution is triggered when, for example, the directly-follows graph is not complete, the system cannot be accurately described by a process tree, or the log contains empty traces. To improve precision in fallback cases, multiple fallback patterns are defined in [130], which are checked in the order given below. When no pattern applies, a so-called flower model will be returned, allowing for any behavior. Table 4.2 provides illustrative examples for each of the fallback pattern detailed below.

##### ■ Fallback 4.1 — Empty Traces.

*Applies when:* The event log contains empty traces, i.e.,  $\varepsilon \in L$ .

*Solution:* To preserve fitness, a skip is modeled using  $\times(\tau, \dots)$ , and the discovery continues with a recursion on the log without empty traces, i.e., on the log  $L_1 = [\sigma \in L \mid \sigma \neq \varepsilon]$ .

*Example:* In the example from Table 4.2, the empty traces are removed. Upon recursion, a sequence cut can now be detected in the sublog  $L_1$ .

##### ■ Fallback 4.2 — Activity Once Per Trace.

*Applies when:* An activity  $a$  appears precisely once in every trace of the log  $L$ .

*Solution:* This fallback discovers that activity  $a$  can be put concurrent to the event log without  $a$ , i.e.,  $a$  is concurrent to the log  $L_1 = L \downarrow_{\mathbb{A}(L) \setminus \{a\}}$ . The discovery continues with a recursion on the sublog  $L_1$ . In case multiple activities  $a$  are valid for this fallback, an arbitrary one is chosen.

*Example:* In the example from Table 4.2, activity  $d$  is selected to be put concurrent to the remaining sublog. Upon recursion, a sequence cut can now be detected in the sublog  $L_1$ .

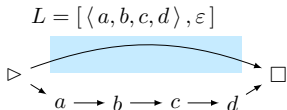
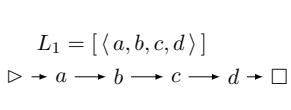

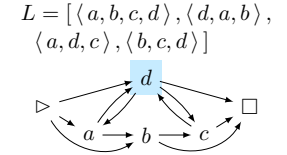
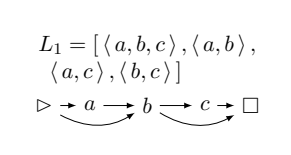

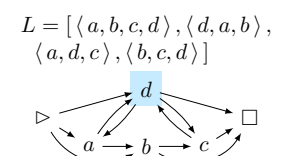
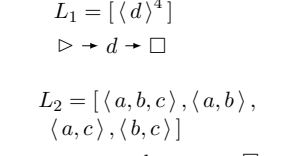
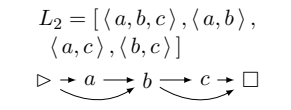

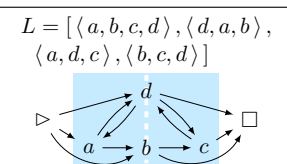
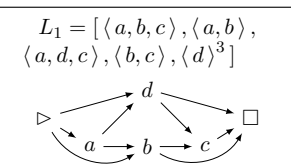

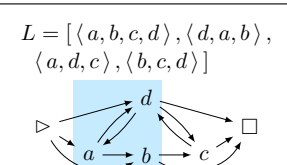
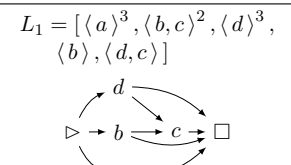

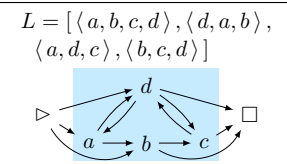
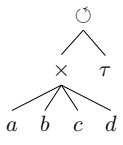
##### ■ Fallback 4.3 — Activity Concurrent.

*Applies when:* Leaving out an activity  $a$  allows a valid cut to be detected.

*Solution:* This fallback chooses an arbitrary activity  $a$ , filters it from the log  $L$ , and tries to detect a valid cut. If this succeeds,  $a$  is put concurrently to the filtered log. This fallback recurses on two sublogs: one with only  $a$ , i.e.,  $L_1 = L \uparrow_{\{a\}}$ , and one without  $a$ , i.e.,  $L_1 = L \downarrow_{\mathbb{A}(L) \setminus \{a\}}$ .

*Example:* In the example from Table 4.2, activity  $d$  is selected to be put in

**Table 4.2:** Example Inductive Miner fallback cases, illustrating how the different fallback solutions work. Each row illustrates an independent fallback example, showing the fallback input, the recursive call input, and the intermediate discovered model. The highlights in the Input column indicate the parts of the directly-follows graph used in the fallback conditions. The highlights in the Discovered Model column indicate where in the resulting model a recursive discovery is applied.

Fallback Case	Input	Resulting Recursion Input	Discovered Model
Empty Traces	$L = [\langle a, b, c, d \rangle, \varepsilon]$ 	$L_1 = [\langle a, b, c, d \rangle]$ 	
Activity Once Per Trace	$L = [\langle a, b, c, d \rangle, \langle d, a, b \rangle, \langle a, d, c \rangle, \langle b, c, d \rangle]$ 	$L_1 = [\langle a, b, c \rangle, \langle a, b \rangle, \langle a, c \rangle, \langle b, c \rangle]$ 	
Activity Concurrent	$L = [\langle a, b, c, d \rangle, \langle d, a, b \rangle, \langle a, d, c \rangle, \langle b, c, d \rangle]$ 	$L_1 = [\langle d \rangle^4]$  $L_2 = [\langle a, b, c \rangle, \langle a, b \rangle, \langle a, c \rangle, \langle b, c \rangle]$ 	
Strict Tau Loop	$L = [\langle a, b, c, d \rangle, \langle d, a, b \rangle, \langle a, d, c \rangle, \langle b, c, d \rangle]$ 	$L_1 = [\langle a, b, c \rangle, \langle a, b \rangle, \langle a, d, c \rangle, \langle b, c \rangle, \langle d \rangle^3]$ 	
Tau Loop	$L = [\langle a, b, c, d \rangle, \langle d, a, b \rangle, \langle a, d, c \rangle, \langle b, c, d \rangle]$ 	$L_1 = [\langle a \rangle^3, \langle b, c \rangle^2, \langle d \rangle^3, \langle b \rangle, \langle d, c \rangle]$ 	
Flower Model	$L = [\langle a, b, c, d \rangle, \langle d, a, b \rangle, \langle a, d, c \rangle, \langle b, c, d \rangle]$ 	No recursive call	

parallel. Upon recursion, a sequence cut can now be detected in the sublog  $L_2$ . Note that, in this example, recursive discovery on sublog  $L_1$  yields a leaf with activity  $d$ . However, since a recursive discovery is performed on sublog  $L_1$ , activity  $d$  could also end up in a structured loop or a skip construction.

■ **Fallback 4.4 — Strict Tau Loop.**

*Applies when:* Looping behavior is present.

*Solution:* This fallback checks for looping behavior by splitting each trace in the event log  $L$  on each occurrence of an end activity  $a \in \text{End}(L)$  followed by a start activity  $a \in \text{Start}(L)$ . If resulting sublog  $L_1$  has more traces than the original input log  $L$ , i.e., at least one trace was split, then the new sublog is used in the recursive discovery and a tau loop is modeled using  $\odot(\dots, \tau)$ .

*Example:* In the example from Table 4.2, the start activities of  $L$  are  $\{a, b, d\}$  and the end activities are  $\{b, c, d\}$ . After splitting, the resulting sublog  $L_1$  has more traces. Upon recursion, a sequence cut can now be detected in the sublog  $L_1$ .

■ **Fallback 4.5 — Tau Loop.**

*Applies when:* Looping behavior is present.

*Solution:* This fallback checks for looping behavior by splitting each trace in the event log  $L$  on every occurrence of a start activity. If the resulting sublog  $L_1$  has more traces than the original input log  $L$ , i.e., at least one trace was split, then the new sublog is used in the recursive discovery and a tau loop is modeled using  $\odot(\dots, \tau)$ .

*Example:* In the example from Table 4.2, the start activities of  $L$  are  $\{a, b, d\}$ . Upon splitting, the resulting sublog  $L_1$  has more traces. Upon recursion, a sequence cut can now be detected in the sublog  $L_1$ .

■ **Fallback 4.6 — Flower Model.**

*Applies when:* The event log contains no empty traces.

*Solution:* This fallback returns a model that allows for any behavior without the empty trace. The so-called flower model is constructed by looping over a big choice over all the activities in  $L$ , i.e., the model  $\odot(\times(a_1, \dots, a_n), \tau)$  where  $\mathbb{A}(L) = \{a_1, \dots, a_n\}$ .

*Example:* In the example from Table 4.2, we have the set of activities  $\mathbb{A}(L) = \{a, b, c, d\}$ . The flower model is instantiated and no recursion is performed.

The above fallback cases handle several interesting edge cases, for example:

$$\begin{aligned}
 \text{IMDiscover}([\langle a, a \rangle]) &= \odot(\text{IMDiscover}([\langle a \rangle^2]), \tau) && \text{Apply Fallback 4.4} \\
 &= \odot(a, \tau) && \text{Apply Base Case 4.2} \\
 \text{IMDiscover}([\langle a \rangle, \varepsilon]) &= \times(\tau, \text{IMDiscover}([\langle a \rangle])) && \text{Apply Fallback 4.1} \\
 &= \times(\tau, a) && \text{Apply Base Case 4.2}
 \end{aligned}$$

## 4.3 Guarantees

The Inductive Miner (IM) framework, as introduced in the previous section, offers good discovery guarantees, scales well, and provides clear extension points for our adaptations. In this section, we will briefly cover the discovery guarantees provided by the basic framework as described above. We refer the reader to Section A.1 on page 365 for the proofs.

### 4.3.1 Soundness and Termination

We start with two general properties of the IM framework: soundness is guaranteed (Theorem 4.3.1), and termination is guaranteed (Theorem 4.3.2).

**Theorem 4.3.1 — IM guarantees soundness.** All models returned by the IM framework are guaranteed to be sound.

*Sketch of Proof.* See page 365. □

**Theorem 4.3.2 — IM guarantees termination.** The IM framework is guaranteed to always terminate.

*Sketch of Proof.* See page 365. □

### 4.3.2 Perfect Fitness

As stated in Section 1.1.2, we want the discovered model to fit the actual behavior. That is, we want the discovered model to at least contain all the behavior in the event log. The perfect fitness guarantee in Theorem 4.3.3 states that all the log behavior is in the model discovered by the IM framework.

**Theorem 4.3.3 — IM guarantees fitness.** The IM framework returns a model that fits the log. That is, given an event log  $L$ , the IM framework returns a model  $Q$  such that  $L \subseteq \mathcal{L}(Q)$ .

*Sketch of Proof.* See page 366. □

### 4.3.3 Language Rediscoverability

The language rediscoverability property tells whether and under which conditions a discovery algorithm can discover a model that is language-equivalent to the original process. That is, given a ‘system model’  $Q$  and an event log  $L$  that is complete with respect to  $Q$  (for some notion of completeness), then we rediscover a model  $Q'$  such that  $\mathcal{L}(Q') = \mathcal{L}(Q)$ .

We will show language rediscoverability in several steps. First, we will define the notion of language complete logs. Then, we define the class of models

that can be language-rediscovered. And finally, we will show the language rediscoverability theorem.

### Language Completeness

Language rediscoverability holds for directly-follows complete logs.

**Definition 4.3.1 — Directly-follows Completeness.** Recall the directly-follows relation ( $\mapsto_L$ ) and graph functions ( $Start(L)$ ,  $End(L)$ ) from Definition 2.4.1 on page 50 as well as the activity projection ( $\mathbb{A}(L)$ ) from Definition 2.3.6 on page 48. A log  $L$  is directly-follows complete to a model  $Q$ , denoted as  $L \diamond_{df} Q$ , if and only if:

1.  $\forall a, b \in \mathbb{A}(Q) : a \mapsto_{\mathcal{L}(Q)} b \Rightarrow (a, b \in \mathbb{A}(L) \wedge a \mapsto_L b)$ ;
2.  $Start(\mathcal{L}(Q)) \subseteq Start(L)$ ;
3.  $End(\mathcal{L}(Q)) \subseteq End(L)$ ; and
4.  $\mathbb{A}(Q) \subseteq \mathbb{A}(L)$ .

### Class of Language-Rediscoverable Models

We will prove language rediscoverability for the following class of models.

**Definition 4.3.2 — Class of Rediscoverable Process Trees.** Let  $\mathbb{A}(Q)$  denote the set of activities in  $Q$ . A model  $Q$  is in the class of language rediscoverable models iff for all nodes  $\otimes(Q_1, \dots, Q_n)$  in  $Q$  we have:

1. No duplicate activities:  $\forall i \neq j : \mathbb{A}(Q_i) \cap \mathbb{A}(Q_j) = \emptyset$ ;
2. In the case of a loop, the sets of start and end activities of the first branch must be disjoint:  
 $\otimes = \circ \Rightarrow Start(Q_1) \cap End(Q_1) = \emptyset$
3. No  $\tau$ 's are allowed:  $\forall i : Q_i \neq \tau$ .

Note that when referred to a *reduced model*, we refer back to the normal form and the reduction rules detailed in Table 2.1 on page 42.

### Language-Rediscoverable Guarantee

Using the above language completeness definition and class of language-rediscoverable models, we can now proof language-rediscoverability. The proof for this property is divided into four steps:

1. In Lemma 4.3.4 we show that the base cases can be rediscovered.
2. In Lemma 4.3.5 we show that any root process tree operator can be rediscovered, proving that the cut criteria are correct.
3. In Lemma 4.3.6 we show that for all process tree operators, the graph cut yields the correct activity division and the log is correctly subdivided.
4. Finally, Theorem 4.3.7 uses the above lemmas to prove language rediscoverability using induction on the model size.

**Lemma 4.3.4 — IM rediscovered base cases.** Let  $Q = a$  for some  $a \in \mathbb{A}$  or let  $Q = \tau$ ; let  $L$  be a log such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ . Then  $\mathbf{IMdiscover}(L) = Q$ .

*Sketch of Proof.* See page 366. □

**Lemma 4.3.5 — IM selects right tree operator.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the model restrictions in Definition 4.3.2 and let  $L$  be a log such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ . Then  $\text{FindCut}(L)$  selects  $\otimes$ .

*Sketch of Proof.* See page 366. □

**Lemma 4.3.6 — IM splits logs correctly.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the model restrictions in Definition 4.3.2 and let  $L$  be a log such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ . Let  $\otimes$  be the result of  $\text{FindCut}(L)$  and let  $(L_1, \dots, L_n)$  be the corresponding result of  $\text{SplitLog}$ . Then for the resulting sublogs  $L_i$  we have  $\forall i : L_i \diamond_{df} Q_i \wedge L_i \subseteq \mathcal{L}(Q_i)$ .

*Sketch of Proof.* See page 366. □

Finally, using the above lemmas, we prove language rediscoverability using induction on the model size.

**Theorem 4.3.7 — IM guarantees language rediscoverability.** If the model restrictions in Definition 4.3.2 hold for a process tree  $Q$ , then  $\text{IMdiscover}$  language-rediscoveres  $Q$ , i.e.,  $\mathcal{L}(Q) = \mathcal{L}(\text{IMdiscover}(L))$  for any log  $L$  such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ .

*Sketch of Proof.* See page 367. □

#### 4.3.4 Polynomial Runtime Complexity

The IM framework is implemented as a polynomial algorithm and scales well with large event logs. Due to the used directly-follows abstraction, the algorithm is able to handle a large number of events and traces and is mainly limited in the number of activities or the alphabet size.

**Theorem 4.3.8 — IM has polynomial runtime complexity.** The runtime complexity of the IM framework is bounded by  $O(|\mathbb{A}(L)|^5 + |\mathbb{A}(L)| \cdot |L|)$ .

*Proof.* See page 367. □

### 4.4 Existing Behavioral and Scalability Extensions

The Inductive Miner (IM) framework, as introduced in the above sections, provides a good basis for process discovery. In addition, the base cases, cuts and splits, and the fallbacks provide easy extension points. Below, we discuss some existing extensions using these extension points, illustrating the various

ways in which the framework can be extended while maintaining (some of) the guarantees from the previous section.

In [130, Section 6.2], the *Inductive Miner – infrequent (IMf)* algorithm is presented, handling deviating and infrequent behavior. This algorithm extends the cut detection (`FindCut` in Algorithm 4.1) as follows: if no cut was detected, filter the directly-follows graph according to some user-chosen frequency threshold and repeat the cut detection. Likewise, the log splitting is modified to take this filtering into account. The user can set this frequency threshold by using a so-called “path” slider in the user interface implementations. In addition, the single activity base case and empty traces fallback are slightly altered to take into account infrequent behavior. Naturally, since IMf excludes behavior of the event log from the discovered model, perfect fitness is no longer guaranteed. However, the rediscoverability guarantee is maintained.

In [130, Section 6.3], the *Inductive Miner – incompleteness (IMc)* algorithm is presented, handling incompleteness. By using probabilistic activity relations in the cut detection, this algorithm accounts for missing behavior. Naturally, since IMc does not necessarily honor all observed activity relations, perfect fitness is no longer guaranteed. However, the rediscoverability guarantee is maintained.

In [130, Section 6.4] it is shown how the list of process tree operators can easily be extended. Here, new cases are added to the cut detection (`FindCut` in Algorithm 4.1) and log splitting (`SplitLog` in Algorithm 4.1) to support interleaved behavior and the inclusive choice/OR split. By construction of the new operators and language join-functions, perfect fitness is maintained (Theorem 4.3.3 can be proved independent of the actual tree operators). By covering these new cases in extensions of Lemmas 4.3.5 and 4.3.6, the rediscoverability guarantee is maintained.

In [130, Section 6.5], a *lifecycle aware* extension is presented, which adds support for non-atomic event logs by adapting the single activity base case, the cut detection, and some of the fallback solutions. And in [130, Section 6.6], support is added for very large event logs by replacing the sublogs with sub directly-follows graphs. That is, the framework no longer recurses using event log as input. Instead, we *recurse on a directly-follows abstraction*, such that in each recursion, only a directly-follows relation needs to be copied instead of an event log. Hence, we save memory otherwise needed for all the sublogs.

## 4.5 Open Challenges and Our Approach

In this chapter, we presented the well-known Inductive Miner (IM) framework, as described in [130], as our process discovery foundation. This framework offers good discovery guarantees, scales well, and provides clear extension points for our adaptations. However, existing extensions do not cover the need for



discovering models with hierarchy (Challenge 2), recursion (Challenge 3), and cancelation (Challenge 4). In addition, as noted in Section 4.3.4, the IM framework is mainly limited in the number of activities or the alphabet size.

In Chapters 6 and 7, we will use the extension points provided by the IM framework to add support for hierarchical, recursive, and cancelation behavior. Using the basis provided in Section 4.3, we will show that our extensions maintain many of the original guarantees. In addition, we will show how the hierarchical discovery approach has the potential to significantly improve the running time in the presence of a large number of activities.

# II | Hierarchical Process Discovery

<b>5</b>	<b>On Software Data and Behavior</b> .....	<b>91</b>
5.1	Software Data Sources and Case Identification	
5.2	Business Event Logs versus Software Event Logs	
5.3	Software Event Data	
5.4	Conclusion	
<b>6</b>	<b>Hierarchical and Recursion Aware Discovery</b>	<b>109</b>
6.1	Why We Need Hierarchy and Recursion – Reality Is Not Flat	
6.2	Hierarchical Event Logs	
6.3	Hierarchical Process Trees	
6.4	Naïve Hierarchical Discovery	
6.5	Recursion Aware Discovery	
6.6	Compatibility with Other Extensions	
6.7	Evaluation	
6.8	Conclusion and Open Challenges	
<b>7</b>	<b>Cancellation Discovery</b> .....	<b>163</b>
7.1	Why We Need Cancellation – The Exceptional Case	
7.2	Cancellation Process Trees	
7.3	The Trigger Oracle	
7.4	Cancellation Discovery	
7.5	Compatibility with Other Extensions	
7.6	Evaluation	
7.7	Conclusion and Open Challenges	

---

## I Introduction

Chapter 1  
Overview

Chapter 2  
Preliminaries

Chapter 3  
Related Work

Chapter 4  
A Process Mining  
Foundation

---

## II Hierarchical Process Discovery

Chapter 5  
On Software Data  
and Behavior

Chapter 6  
Hierarchical and Recursion  
Aware Discovery

Chapter 7  
Cancellation Discovery

---

## III Beyond Model Discovery

Chapter 8  
Hierarchical Performance  
Analysis

Chapter 9  
Translations  
and Tracability

---

## IV Applications

Chapter 10  
Tool Implementations

Chapter 11  
The Software Process  
Analysis Methodology

Chapter 12  
Case Studies

---

## V Closure

Chapter 13  
Conclusion

Appendix A  
Proofs

In Part II, we introduce the challenges of software process discovery and present our novel hierarchical process discovery techniques.

**Chapter 5** discusses the properties, patterns, and the accompanied challenges of software event logs.

**Chapter 6** introduces the *hierarchical process tree* notation and discovery techniques for hierarchical and recursive behavior.

**Chapter 7** introduces the *cancellation process tree* notation and discovery techniques for cancellation behavior.

*“The purpose of a storyteller is not to tell you how to think, but to give you questions to think upon.”*

— Brandon Sanderson, *The Way of Kings*

## 5 | On Software Data and Behavior

In this chapter, we will discuss software event logs, how to obtain such logs, and how software event logs compare to traditional business event logs (Contribution 1). Section 5.1 discusses various sources of software event data and different techniques for obtaining and structuring such data. After that, Section 5.2 compares software event logs to traditional business event logs. Section 5.3 proposes some common elements and structures for software event data. Finally, Section 5.4 concludes this chapter.

5

### 5.1 Software Data Sources and Case Identification

Event log data capturing actual system behavior can be found at many places: in enterprise information systems and business transaction logs, in web servers, in high-tech systems such as X-ray machines and wafer scanners, in warehousing systems, etc. [8] In this section, we will look into these various data sources (Section 5.1.1), and discuss different techniques for obtaining (Section 5.1.2) and structuring (Section 5.1.3) such data.

#### 5.1.1 Log Files, Monitoring, Tracing, and Instrumentation

There are many different sources for obtaining software data. Below, we will discuss the most common sources of data by dividing the discussion into four data source types: *log files*, *monitoring*, *tracing*, and *instrumentation*.

##### Log Files

The most obvious source for software data are *log files already provided by existing systems*. For example, the Apache web server [30], which serves 43% of all active websites [149], records *error logs* and *access logs*. The *error logs* record diagnostic information and any errors that Apache encounters in processing requests. And the *access logs* records all (HTTP) requests processed by the server. Based on these logs, one could derive how users use a website, which pages are accessed in which order, how individual web pages perform, where and when errors occur, etc. As another example, consider the various transaction logs at Adyen, a large service-provider company in the payment

industry [195]. Whenever Adyen handles a payment service request, several log entries are produced in multiple different systems. The above are just a few examples of the many software systems and companies that have logging in place, usually for debugging and troubleshooting purposes. However, these examples also illustrate some of the challenges associated with using such existing log files. For one, relevant and interesting log entries may be hidden among all the log entries that are less relevant. For another, large-scale logging is usually deployed using so-called *rotating logs*, i.e., log files are periodically archived or deleted in order to deal with the large amounts data being produced.

### Monitoring

Another common source for software data is *external monitoring*. There exist many monitoring frameworks geared to various purposes. For example, the Kieker framework [89] is designed to monitor and analyze a software system's internal runtime behavior. In contrast, application performance management (APM) suites like AppDynamics [31], New Relic [164], and XRebel [182] are focussing on cross-stack monitoring and analysis, showing how a collection of programs, servers, databases, and more are linked together and perform. Like APM suites, network monitoring tools like Wireshark [179] and UNIX Snoop [175] allow for cross-stack monitoring of large software systems. As another example, Google Analytics [70], the most widely used web analytics service on the internet [189], monitors and tracks users across websites or within applications like ProM 6.7 [185]. These monitoring frameworks typically hook into a target software system, record all events to an internal database or a structured format, and present the analyst with a dashboard-like summary of (recent) activities. Monitoring information, especially when stored in an internal database, is typically well structured and documented.

### Tracing

Many of the more complex software systems usually have some kind of on-demand *tracing infrastructure*. For example, ASML, a large high-tech company producing lithography machines, employs a tracing infrastructure in their component-based software systems for debugging and troubleshooting purposes (see also Chapter 12). As another example, Google's Dapper is a production distributed systems tracing infrastructure. Dapper aims to obtain more information for developers about the behavior of complex distributed systems [170]. Similarly, Facebook's ÜberTracing was designed to trace the interaction of software components during the end-to-end processing of a Facebook request [54]. In contrast to log files and monitoring, tracing infrastructures are a high-precision tool and designed to be configurable. That is, they do not log everything all the time. Rather specific behavior and data is traced upon request, and such data is often of high quality and designed with various analyses in

mind. For example, Facebook's ÜberTracing was designed to unify various tracing sources for analysis. In particular, these ÜberTraces enable request analyses in Facebook's Mystery Machine, a causal graph discovery tool [54].

### Instrumentation

The most obvious examples of *instrumentations* are *software profilers*. Tools like the Yourkit Profiler [136] and our solution presented in Chapter 10 instrument software by altering the program source code or its binary executable to sample or trace behavior and performance information. The key feature here is that the instrumentation by such tools is fully automatic and non-invasive. That is, the original software source code is not changed and the sampling or tracing code is not added manually by a developer. Although the information recorded by profilers is usually not (directly) accessible, similar instrumentation techniques can easily be employed for streaming or logging event data. For many target programming languages, instrumentation techniques can easily be built using one of the various Aspect-Oriented Programming (AOP) tools [65], such as AspectJ [71] for Java, AspectC++ [173] and AOP++ [200] for C++, and the TXL processor [57] for many other languages. In addition, many interpreted languages provide code-manipulation hooks such as, for example, Java Agents and Javassist [52, 53] for the Java programming language. The benefit of instrumentation approaches is the very high-quality, precise and fine-grained information that is available. However, such instrumentation can produce a noticeable overhead and as a result the behavior of the instrumented software system may be different from the unmodified software system, especially in the context of deadlines and race conditions. This is an unavoidable consequence as observing a system changes the system [168]. However, one should nevertheless strive to minimize the impact of the instrumentation code on the software system's behavior and performance.

#### 5.1.2 Software Environment and Stimuli

Next to the software data sources discussed above, the software systems also need an environment and external stimuli to trigger actual behavior and generate software event data. Below, we will discuss various approaches for triggering behavior.

For detailed and controlled software behavior analyses, a *development or test environment* can be used. In a development environment, one typically uses tracing and instrumentation techniques to observe a software system under carefully controlled setups. The software can be triggered via various means, such as: unit and integration test suites, simulation suites, scenario-based testing, random stimuli, etc. Such a setup is especially useful for obtaining a detailed and fine-grained observation of a piece of software. When an analysis focusses on the internal behavior of a specific part of a software system, such

detailed and fine-grained observation can be very useful. In addition, observing a software system in a development environment, possibly in a mock-up or simulation setting, is usually cheaper and more accessible than other setups. The downside of using such a controlled setup is the lack of any real-life behavior (e.g., lack of usage patterns, tests cannot cover all behavior, etc.) or real-life conditions (e.g., high load or resource demand).

For a more real-life analysis, an *acceptance (test) environment* can be used. An acceptance environment usually resembles a production environment more closely, and usually involves customers or users testing the software system or product. The upside is of using acceptance testing for triggering behavior is the more realistic usage patterns. As a result, more unusual behavior may be triggered, covering more corner cases in the software. The downside is that acceptance testing is more costly and thus limited. In addition, certain real-life conditions usually cannot be reproduced during testing (e.g., high load or resource demand).

Finally, the *production environment* can be used to observe real-life behavior and investigate the software system under real-life conditions. Especially the monitoring tools like the APM suites (see Section 5.1.1) are designed to be used in a production environment. Most of the performance problems, unusual usage patterns, and exceptional errors become only visible once a software system is in production. In addition to observing the behavior of a software system itself, a production environment can also be used for observing how a software system is used. However, due to the costly and sensitive nature of a production environment, any behavior observation has to be non-intrusive with respect to both the software behavior as well as the performance. Therefore, it is unlikely that one would obtain any detailed and fine-grained observation in a production environment.

### 5.1.3 Event Structuring and Case Identification

Once a dataset has been obtained from a software data source, this data has to be interpreted and structured. Following the structure of an event log (see Section 2.3.1 on page 43), there are various interpretation questions to be answered in parsing software data and mapping such data to an event log:

- What is the main subject being observed? I.e., what is the case notion?
- What are the activities describing the behavior?
- Which data represents lifecycle transition types?
- What timestamp data will be used?
- Which resource notions are associated with the activities?
- What other data exist, and what are the associated semantics?

The most important decision here is the *case notion*. That is, the case notion determines the view on the dataset, how traces are constructed, and

the subject of further process analysis efforts. Often, in practice, a dataset can be interpreted in many different ways, yielding different case notions. For example, when considering the Apache access logs, one can adopt a user-centric view such that each trace represents the activities of a user. An equally valid choice would be a page-centric view such that each trace represents the visit of a particular page by various users. Below, we will further detail three common approaches for determining a case notion: *instance identifiers*, *business transactions*, and *windowing/protocol runs*.

### Instance Identifiers

The simplest approach to construct a case notion is to *use a set of existing attributes as instance or case identifiers*. In many (software) event datasets various context attributes are present. A web server access log may log page names, request identifiers, session identifiers, host names, and service, client or user ids. A cyber-physical system may log product numbers, batch identifiers, or location names. A detailed execution log of internal software behavior may log process and thread instance ids, class object addresses, or communication port numbers. Sometimes, such data can be used directly for case identification. In other cases, only a little preprocessing is needed for normalizing and combining attribute names and values.

### Business Transactions

Another common approach is to identify *business transactions*. A business transaction is a sequence of related events, which together detail a specific (user/service) request. To construct a business transaction, data from several software systems, usually across a specific software stack, is combined based on certain identifiers. For example, connection information from communication channels can be used to correlate events from a server and a client. In other cases, session, request, or span identifiers are passed across different systems. In addition, timestamps and (interval) containment relations can be analyzed to form business transactions [119]. Business transactions are typically used for a user-centric or request-centric analysis of a distributed system. Especially in APM suites [31, 164, 182] and approaches like Dapper [170], ÜberTrace [54], service mining [6], and distributed software mining [119], business transaction notions are used for case identification.

### Windowing/Protocol Runs

In situations where a case notion cannot be constructed from data attributes alone, one can resort to *windowing and protocol run techniques*. In these techniques, one uses repeating patterns to identify the start and end of cases. Typically, one defines rules to “window events” based on observed patterns, i.e., causal rules that group events by indicating the start and end activities of an example run. For example, if one would observe an I/O software interface,



a rule could be that a case starts with a constructor call or an interface method named `open()` and ends with a destructor call or an interface method named `close()` or `release()`. A useful practice is to identify and number repeating patterns, storing the resulting run identifiers in a new, artificial data attribute. Alternatively, one could also track the opened and closed I/O handles in a queue, and number a new run each time the queue is empty, i.e., at so-called *regeneration points*. Afterwards, the new run identifiers can be combined with other instance identifiers such as, for example, (file) handles and product or user identifiers. In Chapter 12, we will show a more detailed real-life example on an event log from a software interface at ASML where a protocol run was identified based on observations made with a Dotted Chart analysis.

## 5.2 Business Event Logs versus Software Event Logs

In the previous section, we discussed the various ways to obtain software event logs. In this section, we will compare a set of publicly-available software event logs with well-known publicly-available business event logs on both log size (Section 5.2.1) and type of behavior (Section 5.2.2).

### 5.2.1 Log Size Comparison

In this section, we present a log size comparison between business event logs and software event logs. The size of an event log is typically measured in terms of four metrics: number of traces, number of events, number of activities (size of the alphabet), and average trace length.

#### Example Event Logs

Table 5.1 shows the event logs used in this comparison with their input sizes.

For the business event logs, we selected the following logs. The *BPIC 2012* [63] and *BPIC 2013* [174] event logs are so-called BPI Challenge logs. These large real-life event logs with complex behavior are often used in process mining evaluations. The challenge logs are made available yearly in conjunction with the BPM conference and are considered sufficiently large and complex inputs to stress test process mining techniques. The *WABO* [45] event log describes the receipt phase of an environmental permit application process ('WABO') at a Dutch municipality. The *Road fine* [60] event log was obtained from an information system managing road traffic fines and is one of the largest event logs in this collection.

For the software event logs, we used an extended version of the instrumentation tool developed for [119], yielding XES event logs with method-call level events. The JUnit 4.12 software [67] was executed once, using the example input found at [40]. For the Apache Commons Crypto 1.0.0 software [29], we executed the `CbcNoPaddingCipherStreamTest` unit test. For the NASA CEV soft-

**Table 5.1:** Event logs used in comparing business and software event logs.

Event Log	# Traces	# Events	# Acts	Trace length		
				Min	Mean	Max
[108] JUnit 4.12	1	946	182	946	946	946
[109] Crypto 1.0.0	3	241,973	74	278	80,658	241,140
[110] NASA CEV	2,566	73,638	47	12	29	50
[112] Alignments	1	17,912	90	17,912	17,912	17,912
[63] BPIC 2012	13,087	262,200	24	3	20	175
[174] BPIC 2013	7,554	65,533	13	1	9	123
[45] WABO	1,434	8,577	27	1	6	25
[174] Road Fines	150,370	561,470	11	2	4	20

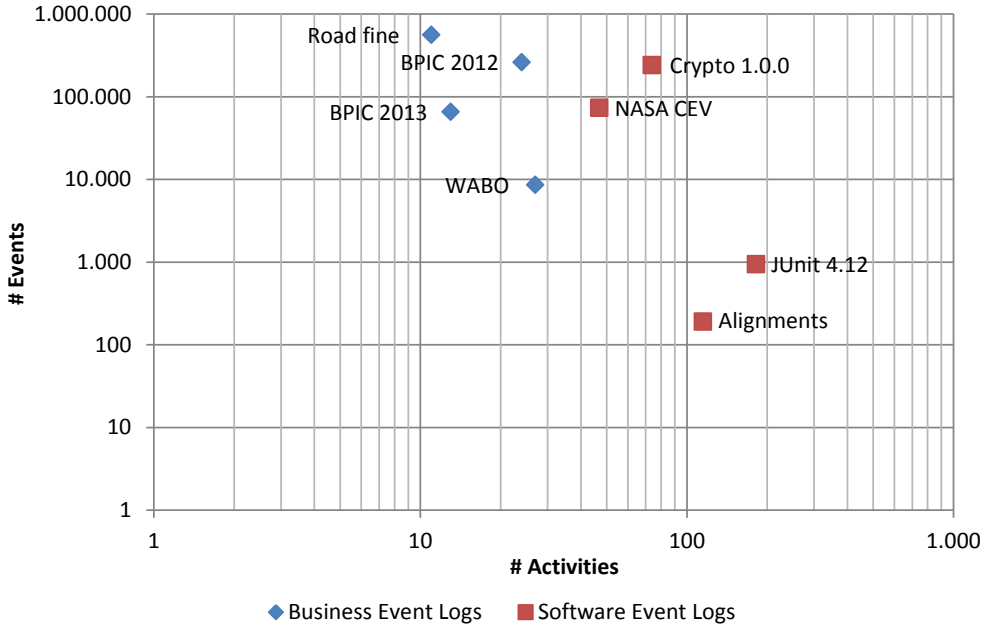
ware [148], we executed a unit test generated from the source code, covering all of the code branches. For the alignments software [21, 187], we executed an alignment computation on a typical input log and model.

### Comparison

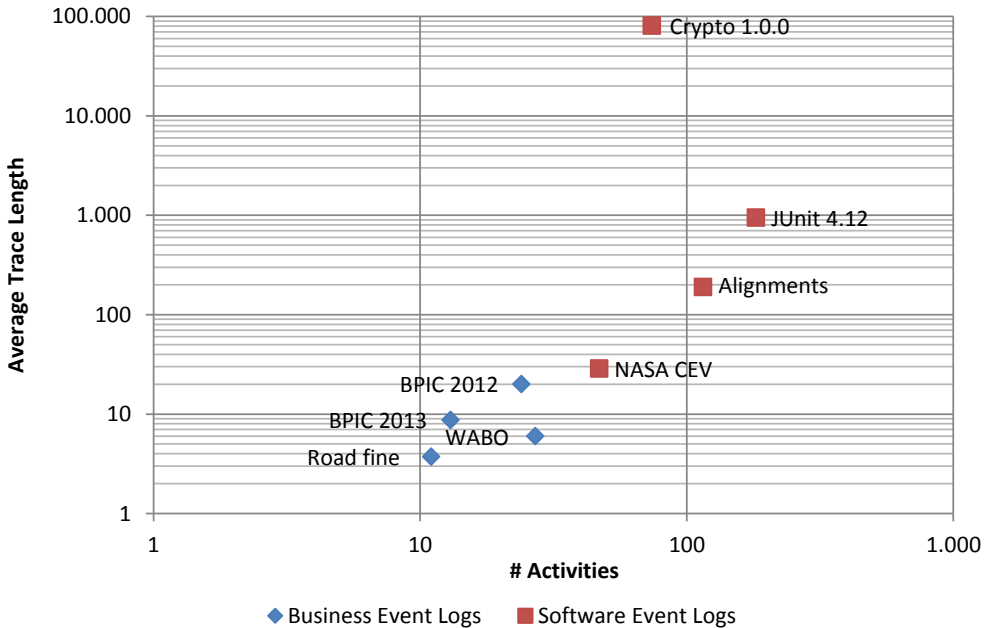
As we can already see from Table 5.1, business and software event logs differ greatly in the number of activities and average trace length. Figure 5.1 presents two graphical comparisons, illustrating the size differences between logs (note the logarithmic scales). We will discuss these comparisons in more detail below.

Based on Figure 5.1a, we observe that, although business and software event logs have a comparable number of total events, software event logs tend to have an order of magnitude more unique activities. Recall from Chapter 1, page 9, that current state-of-the-art process discovery techniques scale well with a large number of events and traces, but are mainly limited by the number of activities or length of traces. Hence, the fact that software event logs tend to have significantly more activities can be problematic. Especially process discovery algorithms can suffer from the large amount of activities. For example, the Inductive Miner discovery algorithm scales linearly in the number of traces and events, but is limited in the number of activities or alphabet size it can handle (see also Section 4.3.4 on page 86).

Based on Figure 5.1b, we observe that software event logs, compared to business event logs, tend to have several orders of magnitude more events per trace. Again, recall from Chapter 1 the scalability limitation of process mining techniques in terms of the trace length. For example, at the time of writing, the Alignments algorithm [21] for conformance and performance analysis cannot process the Apache Crypto event log simply because the implementation was never designed to handle such long traces.



(a) Number of unique activities (x-axis) versus total number of events (y-axis).



(b) Number of unique activities (x-axis) versus average trace length (y-axis).

**Figure 5.1:** Comparison of business event logs and software event logs. Note that all axes use a logarithmic scale. These charts show a clear division between business event logs and software event logs in terms of various log sizes.

Large software event logs with many activities and a large average trace length can be problematic for process mining techniques. However, a smart event log interpretation can go a long way in reducing this complexity. In the next section, we will discuss some opportunities for such smart interpretations.

### 5.2.2 Behavior Comparison and Accompanied Challenges

In this section, we will compare the different types of behavior found in business event logs and software event logs. In addition, we will discuss some of the associated behavioral properties, patterns, and accompanied challenges one can encounter in such software process settings.

#### Inter-Process Relations and Hierarchies

In business event logs, an event log usually captures the activities performed by humans within a business process. In the case such logs originated from an (enterprise) information systems or business transaction logs, each event usually represents a user completing a certain activity and recording the associated information within the system. Since most of the actual work is performed outside of the system, the recorded activities are typically “flat”. That is, activities are not composed of more detailed steps. For example: an employee registers a request and details the customers details, a doctor visits a patient and records the patient’s vitals, etc. In some cases, such as in artifact-centric [39, 151] and service-oriented [6] settings, there are multiple processes or process instances interacting, each providing a perspective on the overall process. For example, a customer may place an order to be delivered (process A), while a warehouse processes multiple orders into multiple deliveries (process B). Such processes are considered equal and interact side-by-side. That is, often there is no master or main process from which the rest of the behavior originates.

In software event logs, each event in an event log typically captures what happens (the activities) at a particular abstraction level. For example, consider a user visiting a website. Such a visit consists of requesting multiple webpages. In turn, requesting a webpage consists of querying multiple systems for one business transaction: a web server cooperates with a database server and a cache server to deliver the requested content. And so on. Often, there is a lot of structure to the recorded behavior and activities are seldom “flat”. That is, typically activities are composed of smaller steps and a hierarchical relations amongst activities and (sub)process can be inferred. In Chapter 6, we will further explore such structures by making the underlying hierarchical relations explicit for process discovery.

#### Error Handling and Cancelation

Another behavioral property to consider is the notion of error handling. In business processes, whenever something goes wrong according to the process,

the actual human users of the process can step in. In such cases, errors are usually handled outside of the information system recording the processes. In extreme cases, the predefined process can be ignored to solve issues, and, as a result, deviating behavior is recorded by the information system. For example, when a patient visiting a hospital for a routine checkup suddenly suffers from a cardiac arrest, the doctor simply ignores the predefined process and acts. Afterwards, the doctor can skip the routine checkup steps and record the unfortunate event. In less extreme cases, errors can be handled at specific go/no-go points in a process. For example, upon a fraud detection, a loan request is simply rejected at a later decision step. There is often no need to explicitly account for error handling within business processes; errors are handled by humans and the appropriate decisions are made at the right moment.

In software processes, error handling is typically much more involved. Often, errors and exceptions must be anticipated and handled automatically. Moreover, a complete abort of a process is usually not desired, and multiple, overlapping error recovery schemes are designed and deployed. Hence, software processes tend to have much more complex error catching, cancelation, and recovery schemes in place. In addition, these notions of errors, cancelations, and recovery schemes are often more explicit in software process than in human-driven processes. In Chapter 7, we will further investigate such error-handling behavior and propose a process discovery solution based on cancelation regions.

### **Concurrency, Multi-Instance, and Communication**

One of the basic behavioral patterns covered by process discovery approaches is the notion of concurrency or parallelism. In business event logs, concurrency often occurs when different people or departments can progress independently of each other or when the order of a group of activities does not matter. In some cases, resource and organizational information in an event log can hint at the concurrency. However, in practice, concurrency is simply inferred from the causal relations over multiple traces (recall the concurrency detection from Section 4.2.3 on page 76). With the relatively small number of activities in business event logs, any reasonably sized event log is likely to have seen enough traces to correctly reconstruct concurrency patterns. Communication between parallel branches is either visible via synchronization points in the process or occurring outside the information system recording the process.

When observing larger software systems, concepts like multi-threading, multi-processes, and distributed systems start to become relevant. In event logs from such systems, there is often much more concurrency than in business event logs. Moreover, there is often a precise structure or order to this type of concurrency. Often there is a lot of extra data available about this concurrency and the communication between concurrent parts of a software process. Simply looking at the causal relations alone to infer the concurrency patterns

is unlikely to be effective. Not only is the relatively large number of activities a problem, also the unlikelihood of specific interleavings is making concurrency discovery more difficult. In the software engineering community, hidden bugs and deadlocks caused by so-called race conditions are infamous examples of how unlikely certain concurrency interleavings occur in practice. In addition, in concurrent software, often the same activity or method is executed multiple times in parallel with itself. For example: a `process()` method is instantiated in parallel for each element in a large dataset to be processed by a software system. The alignments software event log contains a nice example of such a pattern, see for example the evaluation in Section 8.5.1 on page 239. In short, unlike with most business event logs, concurrency in software processes can and should be discovered based on the data perspective in a software event log. Discovery algorithms should actively support inferring causal relations from data attributes like thread and process identifiers, and add support for specifically discovering fork and join patterns as well as multi-instance patterns. We will not address the above challenges in this thesis, but we will come back to some opportunities in the future work sections.

### Services, Batches, and Orchestration

In artifact-centric [39, 151] and service-oriented [6] event logs, more than one process is observed. These processes are interacting and coordinate their progress via some orchestration mechanism. As indicated in [6, 9] one of the main challenges in such a setting is how to correlate instances across all the involved processes. That is, to use the recorded examples in an event log to derive the orchestration mechanisms. Since such coordination and orchestration mechanisms are usually very diverse, and any form of interaction can happen in practice, this is a very broad and challenging problem.

In software event logs, similar challenges occur. For example, in the software log from a software interface at ASML, we identified a product-level process and a batch-level process interacting (see Chapter 12). However, unlike the very broad and generic settings encountered in business settings, such software event logs usually have a lot of auxiliary data available. In the ASML example, the correlation between batch and product instance identifiers can be extracted from the event log itself. Moreover, we can use knowledge about how the different software services interact (based on their design and interfaces), and we can infer specific interaction patterns. For example, we know that an instance of the batch-level process creates and destroys multiple instances of the product-level process. Similar patterns can be found in numerous other software systems: cyber-physical systems such as industrial plants and baggage handling systems processing batches of products/items, databases and (web)services processing batches of requests (e.g., Dapper explicitly uses request parent-child relations [170]), etc. In such settings, it should be possible

to use the extra information from the data perspective to infer where batch workflow patterns are present, how services are connected, how processes are related (possibly hierarchically), and how communication across processes is orchestrated. It should be possible to discover and model such complex combinations of processes. We can model interaction constraints and track different process instances using, for example, colored Petri nets [95] or other high-level nets. We will not address the above challenges in this thesis, but we will come back to some opportunities in the future work sections.

### 5.3 Software Event Data

In this section, we propose common elements and structures for software event data. Some of these common elements were already mentioned in the previous sections. In this section, we assume that a software event log was recorded at the method-call or interface-call level during software execution. This fine-grained view on internal software behavior is much like the tracing, instrumentation, and profiling setups discussed in Section 5.1.1. Events generated at this level reference a specific point in the software source code. The *Software Event* extension [128] captures this event location information, together with some basic runtime information related to this location. In the remainder of this section, we will first cover some basic terminology (Section 5.3.1) and after that cover various data aspects with some examples (Sections 5.3.2 to 5.3.7). We refer to [128] for a complete example event log.

In this section, we describe the common data elements based on the structure of event logs (Section 2.3.1) and the XES extension format (Section 2.3.2). In an XES extension, attributes are defined at a particular *level* (log, trace, event, or meta), have a specific *key* and data *type*, and have an associated *description*. For example, suppose we annotate an event with a “name” of type “string”. We define this “name” attribute as follows:

Level	Key	Type	Description
	event name	string	Some semantical description. . .

We refer the reader to the official XES software extensions for more details and other elements that can be found in software event logs [127, 128, 129].

#### 5.3.1 Terminology

In this section, we briefly explain some key terminology using the example code snippet shown in Listing 5.1. When recording a software event at the method call level, we refer to the method being called or invoked as the *callee*. Optionally, we can track the context method where a method is triggered from. We call this context method the *caller*. That is, the *caller* method invokes the

*callee* method. In Table 5.2, we can see how the callee and caller role can change over the course of executing the code in Listing 5.1.

**Listing 5.1** Example code snippet showing when which event types are triggered.

```

1  class A {
2      void f(int y) { - - - - - call A.f
3          try {
4              - - - - - calling B.g
5                  int r = b.g(12, y);
6              - - - - - returning B.g
7          } catch (Exception e) { - - - - - handle in A.f
8              ...
9          }
10     } - - - - - return/throws A.f
11 }
12 class B {
13     int g(int x, int y) { - - - - - call B.g
14         return x / y;
15     } - - - - - return/throws B.g
16 }

```

**Table 5.2:** The caller-callee roles for an execution of Listing 5.1.

Line	Callee	Caller	Line	Callee	Caller	Line	Callee	Caller
2	A.f	—	4	B.g	A.f	13	B.g	—
10	A.f	—	6	B.g	A.f	15	B.g	—

### 5.3.2 Event Type and Lifecycle Transactions

Software events can be triggered at different states during a software execution. To specify at which state an event was triggered, we record an indicative event type attribute for method-level execution events, as defined below. Listing 5.1 shows these event types on the right in the context of an example code snippet.

Level	Key	Type	Description
event	type	string	Software event type, indicating at which state during execution this event was generated. Possible values are enumerated below.

The possible software event type values we recognize are:

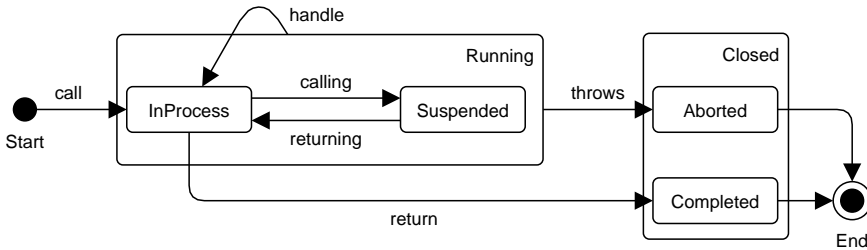
- call**            The start of a method block
- return**        The normal end of a method block
- throws**        The end of a method block in case of an uncaught exception
- handle**        The start of an exception handle catch block
- calling**        The start of calling/invoking another method
- returning**     The end of returning a called method





The software event type values describe transitions in a transactional model for the lifecycle of a method execution. In Figure 5.2, the state machine for the method execution transactional model is given. As an example, Table 5.3 shows the states for the two methods in Listing 5.1. In Chapter 6, we will (indirectly) use this event type information for extracting hierarchies based on nested intervals or nested calls.

The software event type values can be related to values in the *standard lifecycle transactional model*, defined in the *lifecycle* extension. We suggest the mapping in Table 5.4. Note that the standard lifecycle transactional model cannot correctly support the **handle** transition.



**Figure 5.2:** State machine for the method execution transactional model.

**Table 5.3:** The states for the two methods in Listing 5.1 using the transactional lifecycle model from Figure 5.2. Shown are two different runs, with the event types generated at the indicated line numbers, and the corresponding lifecycle state for both methods.

Run without exceptions				Run with exceptions			
Ln.	Event Type	Lifecycle State		Ln.	Event Type	Lifecycle State	
		A. f	B. g			A. f	B. g
2	<b>call</b> A. f	InProcess	–	2	<b>call</b> A. f	InProcess	–
4	<b>calling</b> B. g	Suspended	–	4	<b>calling</b> B. g	Suspended	–
13	<b>call</b> B. g	Suspended	InProcess	13	<b>call</b> B. g	Suspended	InProcess
15	<b>return</b> B. g	Suspended	Completed	15	<b>throws</b> B. g	Suspended	Aborted
6	<b>returning</b> B. g	InProcess	Completed	7	<b>handle</b> in A. f	InProcess	Aborted
10	<b>return</b> A. f	Completed	Closed	10	<b>throws</b> A. f	Aborted	Closed

**Table 5.4:** Mapping from the *method execution transactional model* (Figure 5.2) to the *standard lifecycle transactional model* [8].

Type	Lifecycle	Type	Lifecycle	Type	Lifecycle
<b>call</b>	start	<b>calling</b>	start	<b>handle</b>	reassign
<b>return</b>	complete	<b>returning</b>	complete	<b>throws</b>	ate_abort

### 5.3.3 Event Location

Software events are triggered at a particular, uniquely identifiable location in the software source code. With this location, each event is traceable back to the source code location where it was generated.

Each event has a *callee* location. Events with the **calling** or **returning** event type also have an additional *caller* location. Each location is described by the following attributes, with a *callee-* or *caller-* key prefix:

Level	Key	Type	Description
event	package	string	The <i>package</i> in the software code architecture to which the method belongs.
event	class	string	The <i>class</i> to which the method belongs.
event	method	string	The referenced <i>method</i> name.
event	paramSig	string	The <i>parameter signature</i> of the method.
event	returnSig	string	The <i>return signature</i> of the method.
event	isConstructor	boolean	If the method <i>is a class constructor</i> .
event	instanceId	string	The <i>instance id</i> of the corresponding class instance. The absence of an instance id is represented by the value “0”.
event	filename	string	The <i>file name</i> of the corresponding source code artifact.
event	lineNr	int	The <i>line number</i> of the executed source code statement.

The software event callee values can be related to the *concept name*, defined in the *concept* extension. We suggest to use the concatenation of *callee package*, *class*, *method* and *paramSig* as a *concept:name* (e.g., `demo.A.f(int)`). This way, the concept name is the unique, canonical name for the executed method. Note

**Table 5.5:** Example event location information for some events for an execution of the program in Listing 5.1.

	Line 2 <b>call</b> A.f	Line 4 <b>calling</b> B.g		Line 4 <b>calling</b> B.g
callee-package	demo	demo	caller-package	demo
callee-class	A	B	caller-class	A
callee-method	f	g	caller-method	f
callee-paramSig	(int)	(int,int)	caller-paramSig	(int)
callee-returnSig	void	int	caller-returnSig	void
callee-isConstructor	false	false	caller-isConstructor	false
callee-instanceId	1074593562	660017404	caller-instanceId	1074593562
callee-filename	source/A.java	source/B.java	caller-filename	source/A.java
callee-lineNr	2	13	caller-lineNr	5
concept:name	demo.A.f(int)	demo.B.g(int,int)		

that, in this way, cases like inheritance, method overloading, and dynamic binding can be resolved at the labeling level. Table 5.5 presents example event location information for some events from Listing 5.1.

### 5.3.4 Method Parameter Data

At runtime, when a method is called or returns, data can be passed along. A **return** or **returning** event can define a return data value. A **call** or **calling** event can define multiple parameter data values.

Level	Key	Type	Description
log	hasData	boolean	If method data is recorded for this log.
event	returnValue	string	The <i>return value</i> for the returning method.
event	params	list	List of <i>parameters</i> for the called method.
meta	paramValue	string	A <i>parameter value</i> in the list <i>params</i> .
meta	valueType	string	The runtime <i>value type</i> for a <i>returnValue</i> or a <i>paramValue</i> .

For simple primitive datatypes, like integer and boolean type of parameters, the actual values can be stored directly. For example, for the **call** `A.f` event in Table 5.5, we could record parameter `y = 0`. For more complex datatypes such as class objects, either a serialization, a hash code, or a pointer address can be stored, depending on the use case.

### 5.3.5 Application Metadata

At runtime, the software events are generated by a particular application or process instance. Users can annotate events with application information, indicating which application instance generated the events. Not only makes this the event log more self-contained, it also can help in tracing events in a distributed or multi-process setup.

Level	Key	Type	Description
event	appName	string	The user defined <i>application name</i> .
event	appTier	string	The user defined <i>application tier</i> .
event	appNode	string	The user defined <i>application node</i> .
event	appSession	string	The user defined <i>application session</i> .

### 5.3.6 Runtime Data

At runtime, the software events in an application process are triggered on a particular thread. Such a thread can be identified by its thread id.

The nano time attribute is used to measure elapsed time in software. It does not have to be related to any other notion of system or wall-clock time. The value represents nanoseconds since some fixed but arbitrary time (perhaps

in the future, so values may be negative). Note that, compared to the existing XES timestamp extension, which is limited to milliseconds, the nanotime attribute allows for a more fine-grained time to be stored.

Level	Key	Type	Description
event	threadId	string	The <i>thread id</i> of the thread on which the event was generated.
event	nanotime	int	The elapsed <i>nano time</i> since some fixed but arbitrary time, see the description above.

### 5.3.7 Exception Data

An *exception* in software indicates an error. When an exception is *thrown* from a particular location/method, it can be caught/*handled* at a particular location.

Level	Key	Type	Description
log	hasException	boolean	If exception data is recorded for this log.
event	exThrown	string	The <i>thrown exception type</i> for a <b>throws</b> or <b>handle</b> event.
event	exCaught	string	The <i>caught exception type</i> for a <b>handle</b> event.

For example, suppose that for the program in Listing 5.1 we record a **call** `B.g` event with parameters  $x = 12$  and  $y = 0$ . Then, due to a division by zero, we will observe a **throws** event (line 15) with *exThrown* recording *ArithmeticException*, followed by a **handle** event (line 7) with *exThrown* recording *ArithmeticException* and *exCaught* recording *Exception*. In Chapter 7, we will (indirectly) use this exception data for extracting cancellation regions based on handled exceptions.

## 5.4 Conclusion

In this chapter, we discussed software event logs, how to obtain such logs, and how software event logs compare to traditional business event logs (Contribution 1). In addition, we covered some common elements and structures for software event data. We concluded that unlike business event logs, software event logs tend to have more unique activities and longer traces, which can potentially be problematic for process mining techniques. However, we also observed the rich data which becomes available with observing and logging software executions. Using this data in smart ways can provide various interesting process mining solutions. For example, in Chapter 6, we will further investigate (implicit) hierarchical structures and propose a discovery solution using such hierarchical structures to scale better with more unique activities

and longer traces. In Chapter 7, we will further investigate error-handling behavior and propose an accompanied process discovery solution based on cancellation regions.

“There is always another secret.”

— Brandon Sanderson, *The Final Empire*

## 6 | Hierarchical and Recursion Aware Discovery

In this chapter, we introduce a modeling notation and discovery techniques for hierarchical and recursive behavior (Contribution 2). We start by motivating the need for hierarchy and recursion in Section 6.1. After that, we introduce (*atomic*) *hierarchical event logs* (Section 6.2) and the *hierarchical process tree* notation (Section 6.3). Based on these concepts, Section 6.4 introduces the *naïve hierarchical discovery* algorithm, and Section 6.5 introduces the *recursion aware discovery* algorithm. In addition, Section 6.6 elaborates on extending the ideas to existing discovery extensions and to non-atomic event logs. Finally, Section 6.7 evaluates these algorithms on rediscoverability and performance.

### 6.1 Why We Need Hierarchy and Recursion – Reality Is Not Flat

When applying process mining on event data originating from software systems, new patterns and challenges pop up. Typically, the run-time behavior of a software system is large, complex, and contains some form of hierarchical structure. Such hierarchies can arise from structural relations such as dependencies in client-server relations, communicating software components, or object relations. When focusing on runtime structures, hierarchies can arise from call-relations amongst functions, methods, or co-routine invocations. In such cases, one typically also encounters repetitive structures down the hierarchy, suggesting recursive behavior. In the context of business processes, low-level events recorded by information systems may not directly match high-level activities that make sense to process stakeholders [139]. Often, also in these cases, a hierarchy of business activities can be identified. In most cases, the data contains enough information to extract hierarchies with named submodels and recursions, which can and should be exploited for discovery and analysis purposes. However, current techniques usually produce “flat” models where activities are not composed of more detailed steps. Such models are often not expressive enough to master the observed (software) complexity and are often difficult to understand, see also Section 3.2 and Challenges 2 and 3.

---

**Listing 6.1** Running example Java code illustrating recursive behavior. Upon execution, this program is logged at the method level.

---

```

1  public class Main {
2      // Entry point
3      public static void main(int i) {
4          A inst = input(i); // inst runtime type can be either A or B
5          inst.process(i); // invokes either A.process() or B.process()
6          output();
7      }
8      private static A input(int i) { ... }
9      private static void output() { ... }
10 }
11 class A {
12     public void process(int i) { ... }
13 }
14 class B extends A {
15     public void process(int i) {
16         if (i <= 0) {
17             super.process(i); // Call to A.process()
18         } else {
19             stepPre();
20             process(i - 1); // Recursive call to B.process()
21             stepPost();
22         }
23     }
24     private void stepPre() { ... }
25     private void stepPost() { ... }
26 }

```

---

As an example of hierarchical and recursive software behavior, consider the program in Listing 6.1. The program starts with the `Main.main()` function. During this `main()` function, we start with invoking `Main.input()`, followed by `inst.process()`, and we finish with `Main.output()`. Note that the variable `inst` can at runtime be an instance of either `class A` or `class B`. Based on this runtime type, either `A.process()` or `B.process()` is invoked. For function `A.process()`, no further behavior is specified. For function `B.process()`, a recursive process is defined. Based on the input parameter `int i`, either a base case is executed if `i <= 0`, invoking `A.process()`, or a step case is executed if `i > 0`. During this step case of `B.process()`, first `B.stepPre()` is invoked, followed by a recursive call to `B.process(i - 1)`, decreasing `i`, and finally `B.stepPost()` is invoked.

Observe how, in the above example, the function-call relations induce a hierarchical structure to the process description. In addition, observe that the recursive behavior of `B.process()` is *not a tail recursion*. That is, since there is a computation after each recursive call, it cannot be reduced to a simple

loop. Another consequence of this recursive behavior is the emergent counting behavior in the step case of `B.process()`. For example, due to the recursive call to `B.process()` and the associated call stack, the number of `B.stepPost()` invocations equals the number of `B.stepPre()` invocations. In the next section, we will show how hierarchical event logs can capture this type of behavior.

## 6.2 Hierarchical Event Logs

In this section, we extend the *event log* notation (Definition 2.3.3, page 46) with support for hierarchical behavior. In our extended notation, called the *hierarchical event log*, we can assign subtraces to events, capturing what happened at different levels of granularity. Section 6.2.1 will start with a motivating example based on the program in Listing 6.1. After that, Section 6.2.2 will define *hierarchical event logs*, Section 6.2.3 presents an *atomic* variant, and Section 6.2.4 will discuss various ways to obtain such logs.

### 6.2.1 Example Log of Executing a Program

Consider the program in Listing 6.1 again. Suppose we execute this program with `i = 1`, and `Main.input()` returns an instance of `class B`. And suppose we trace and log the start and end of each called method. Table 6.1 shows a resulting example event log. Observe how the start and complete lifecycle information implicitly capture a set of overlapping intervals. Figure 6.1 explicitly depicts Case 1 as a set of intervals. Note how the intervals contain each other as can be expected from the call relations, implying a hierarchical order. For example, we can deduce that most events happen during the interval of `Main.main()`. Below, we will show how a *hierarchical event log* can make such an implied hierarchical order explicit.

In practice, an event log can record more lifecycle information beyond the start and complete lifecycle transitions assumed above, recall Figure 5.2 on page 104. However, for now, we assume starts and completes only. In most cases, additional lifecycle information, such as the abort and handle lifecycle transitions can be either encoded via starts and completes, or filtered out.

### 6.2.2 The Hierarchical Event Log

In the *hierarchical event log*, we can assign subtraces as attributes to events as defined below, capturing what happened at different levels of granularity.

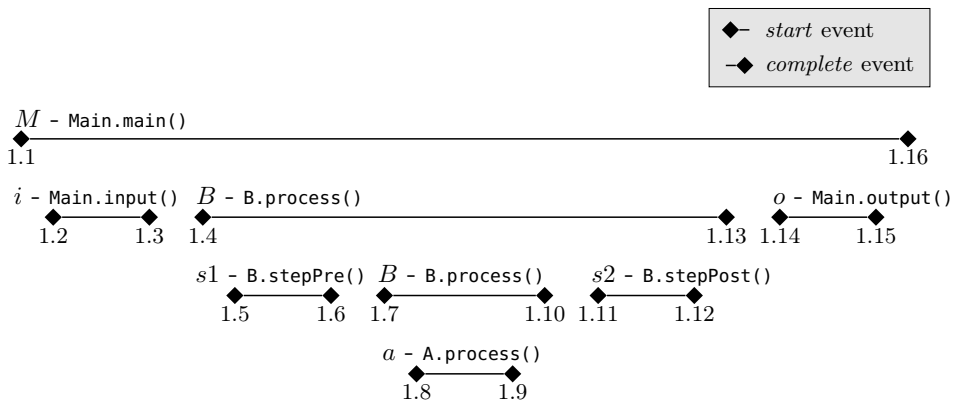
**Definition 6.2.1 — Hierarchical Event Log.** Let  $\mathbb{E}$  be the event universe. We define the following hierarchical extension attribute:

$\#_{subtrace}(e)$  the *hierarchical subtrace*  $\#_{subtrace}(e) \in \mathbb{E}^*$  associated to event  $e \in \mathbb{E}$ .



**Table 6.1:** Example snippet of an event log for the program in Listing 6.1. Each row is an event and each column corresponds to an attribute. The events relate to the start and end of each called method.

Case id	Event id	Attributes					
		Activity	Lifecycle	Timestamp	Resource	...	
1	1.1	<i>M</i> Main.main()	start	30-10-2017 11:02:45.000	main-thread	...	
	1.2	<i>i</i> Main.input()	start	30-10-2017 11:02:45.200	main-thread	...	
	1.3	<i>i</i> Main.input()	complete	30-10-2017 11:02:45.400	main-thread	...	
	1.4	<i>B</i> B.process()	start	30-10-2017 11:02:45.650	main-thread	...	
	1.5	<i>s1</i> B.stepPre()	start	30-10-2017 11:02:45.690	main-thread	...	
	1.6	<i>s1</i> B.stepPre()	complete	30-10-2017 11:02:45.730	main-thread	...	
	1.7	<i>B</i> B.process()	start	30-10-2017 11:02:45.770	main-thread	...	
	1.8	<i>a</i> A.process()	start	30-10-2017 11:02:45.840	main-thread	...	
	1.9	<i>a</i> A.process()	complete	30-10-2017 11:02:45.920	main-thread	...	
	1.10	<i>B</i> B.process()	complete	30-10-2017 11:02:45.940	main-thread	...	
	1.11	<i>s2</i> B.stepPost()	start	30-10-2017 11:02:45.980	main-thread	...	
	1.12	<i>s2</i> B.stepPost()	complete	30-10-2017 11:02:46.160	main-thread	...	
	1.13	<i>B</i> B.process()	complete	30-10-2017 11:02:46.180	main-thread	...	
	1.14	<i>o</i> Main.output()	start	30-10-2017 11:02:46.250	main-thread	...	
	1.15	<i>o</i> Main.output()	complete	30-10-2017 11:02:46.460	main-thread	...	
	1.16	<i>M</i> Main.main()	complete	30-10-2017 11:02:46.580	main-thread	...	
2	2.1	<i>M</i> Main.main()	start	30-10-2017 11:02:47.440	main-thread	...	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	



**Figure 6.1:** Case 1 from Table 6.1 depicts as intervals. Each diamond represents an event, with the event identifier below and activity name (abbreviated and full name) above the diamond. The lines connect corresponding start and complete events, creating the intervals for each activity instance. Observe how the intervals contain each other as expected based on the call relations.

In the presence of lifecycle information, we adopt the convention to assign subtraces to the *start* event. When also timing information is available, we assume that the hierarchical subtrace describes the lower level events during an activity instance. That is, given two events  $e, e'$  that describe the start and end of the same activity instance with  $\#_{life}(e) = start$  and  $\#_{life}(e') = complete$ , if the start event is assigned a subtrace (i.e.,  $\#_{subtrace}(e) \neq \perp$ , recall null/ $\perp$  from Definition 2.3.1 on page 45), then:

$$\begin{aligned} \#_{time}(e) &\leq \#_{time}(head(\#_{subtrace}(e))) \\ &\leq \#_{time}(end(\#_{subtrace}(e))) \leq \#_{time}(e') \end{aligned}$$

Given a set of event sequences  $L \subseteq \mathbb{E}^*$ , the flattened set of all (sub)traces in  $L$  is the smallest set  $Subtraces(L)$  such that:

$$\begin{aligned} Subtraces(\emptyset) &= \emptyset \\ Subtraces(L) &= L \cup Subtraces(\{ \#_{subtrace}(e) \\ &\quad | \sigma \in L \wedge e \in \sigma \wedge \#_{subtrace}(e) \neq \perp \}) \end{aligned}$$

The set  $L \subseteq \mathbb{E}^*$  is a *hierarchical event log* iff no event is duplicated in  $L$ , i.e., for any two traces  $\sigma, \sigma' \in Subtraces(L)$  and any two numbers  $i, j$  such that  $1 \leq i \leq |\sigma|$  and  $1 \leq j \leq |\sigma'|$  we have  $(\sigma(i) = \sigma'(j)) \Rightarrow (\sigma = \sigma' \wedge i = j)$ .

Consider Case 1 from the event log in Table 6.1 again. Table 6.2 shows the corresponding hierarchical event log interpretation using Definition 6.2.1 and the implied interval containment hierarchical order from Figure 6.1.

**Table 6.2:** Example snippet of the hierarchical event log derived from Table 6.1. Each row shows a subtrace, derived from Case 1 and the implied interval containment hierarchical order shown in Figure 6.1. The last column shows the corresponding activity + shortened lifecycle sequence interpretation, using the notation from Definition 2.3.7 on page 49. Note that, amongst others, event 1.8 has no subtraces defined.

Trace	Event Sequence	Activity + Lifecycle Seq.
	Case 1: $\langle \text{event 1.1, event 1.16} \rangle$	$\langle M+s, M+c \rangle$
$\#_{subtrace}(\text{event 1.1}) =$	$\langle \text{event 1.2, event 1.3, event 1.4, event 1.13, event 1.14, event 1.15} \rangle$	$\langle i+s, i+c, B+s, B+c, o+s, o+c \rangle$
$\#_{subtrace}(\text{event 1.4}) =$	$\langle \text{event 1.5, event 1.6, event 1.7, event 1.10, event 1.11, event 1.12} \rangle$	$\langle s1+s, s1+c, B+s, B+c, s2+s, s2+c \rangle$
$\#_{subtrace}(\text{event 1.7}) =$	$\langle \text{event 1.8, event 1.9} \rangle$	$\langle a+s, a+c \rangle$
$\#_{subtrace}(\text{event 1.8}) =$	$\perp$	Undefined

### 6.2.3 The Atomic Hierarchical Event Log

For the basic discovery algorithms and their explanation, we will use the following, simpler atomic view. In the *atomic hierarchical event log* we assume that an event is described by a sequence of atomic activities. That is, if we assume a set of activities  $\mathbb{A}$ , then an event  $e$  is described by a sequence of such activities, i.e.,  $e \in \mathbb{A}^*$ . Hence, we can describe an atomic hierarchical event log simply as a multiset of traces, where each trace is a sequence of events, and each event is a sequence of activities.

**Definition 6.2.2 — Atomic Hierarchical Event Log.** Let  $\mathbb{A}$  be a set of activities. Let  $L \in \mathcal{B}((\mathbb{A}^*)^*)$  be an *atomic hierarchical event log*, a multiset of traces. A trace  $\sigma \in L$  with  $\sigma \in (\mathbb{A}^*)^*$  is a sequence of events. Each event  $e \in \sigma$  with  $e \in \mathbb{A}^*$  is described by a non-empty sequence of activities, stating which activity was executed at each level in the hierarchy.

Consider, for example, the atomic hierarchical event log  $L = [\langle \langle g, a \rangle, \langle g, b \rangle, \langle c \rangle \rangle]$ . This log has one trace, where the first event is labeled  $\langle g, a \rangle$ , the second event is labeled  $\langle g, b \rangle$ , and the third event is labeled  $\langle c \rangle$ . For the sake of readability, we will use the following shorthand notation: given an event labeled  $\langle a_1, a_2, \dots, a_n \rangle$ , we write  $a_1.a_2.\dots.a_n$ . For example, the above log  $L$  can be written as  $L = [\langle g.a, g.b, c \rangle]$ .

**Definition 6.2.3 — Atomic Hierarchical Functions.** We define the following utility functions over atomic hierarchical event logs:

$\|L\|$  *Hierarchical depth:* the length of the longest event label in the event log:

$$\|L\| = \max(\{ |e| \mid \sigma \in L \wedge e \in \sigma \})$$

For example, using the above log  $L$ , we have:  $\|L\| = 2$ .

$f.L$  *Hierarchical concatenation:* appends activity  $f \in \mathbb{A}$  to the start of each event's activity sequence, where  $f \in \mathbb{A} \wedge \forall i : e_i \in \mathbb{A}^*$ :

$$f.L = [f.\sigma \mid \sigma \in L]$$

$$f.\sigma = f.\langle e_1, \dots, e_n \rangle = \langle \langle f \rangle \cdot e_1, \dots, \langle f \rangle \cdot e_n \rangle$$

For example, using the above log  $L$ , we have:

$f.L = [\langle f.g.a, f.g.b, f.c \rangle]$ . Note that  $\|f.L\| = 1 + \|L\| = 3$ .

$L|_i^*$  *Hierarchical projection:* removes a prefix of length  $i$  from every event's activity sequence:

$$L|_i^* = [\sigma|_i^* \mid \sigma \in L]$$

$$\sigma|_i^* = \langle \langle a_{i+1}, \dots, a_n \rangle \mid \langle a_1, \dots, a_n \rangle \in \sigma \wedge i < n \rangle$$

For example, using  $\sigma = \langle g.a, g.b, c \rangle$ , we have:  $\sigma|_0^* = \langle g.a, g.b, c \rangle$ ,  $\sigma|_1^* = \langle a, b \rangle$ , and  $\sigma|_2^* = \varepsilon$ . Note that  $(f.L)|_1^* = L$ .

$\|\mathbb{A}(L)\|$  *Hierarchical alphabet size*: the maximum activity alphabet size across all hierarchy layers:

$$\|\mathbb{A}(L)\| = \max(|\{ \text{head}(e) \mid \sigma \in L \wedge e \in \sigma \}|, \|\mathbb{A}(L|_1^*)\|)$$

For example:  $\|\mathbb{A}([\langle f.a, f.a \rangle])\| = 1$ ,  $\|\mathbb{A}([\langle f.a, g.a \rangle])\| = 2$ ,  $\|\mathbb{A}([\langle f.a, f.b \rangle])\| = 2$ ,  $\|\mathbb{A}([\langle f.a, g.b \rangle])\| = 2$ , and  $\|\mathbb{A}(L)\| = 2$

**Definition 6.2.4 — Deriving an Atomic Hierarchical Event Log.** Let  $L \subseteq \mathbb{E}^*$  be a hierarchical event log as defined in Definition 6.2.1. Assume a classifier  $\lambda_{\#}$  has been defined.

Given a trace  $\sigma \in \mathbb{E}^*$ , a hierarchical trace  $\sigma' = H(\sigma, \lambda_{\#})$  with  $\sigma' \in (\mathbb{A}^*)^*$  is derived as shown below. Function  $H$  inspects each event in the trace. For each event, it either returns that event's label when no substraces are present, or it returns that event's label appended to the start of each of the event's activity sequences found in the corresponding subtrace.

$$H(\varepsilon, \lambda_{\#}) = \varepsilon$$

$$H(\langle e \rangle \cdot \sigma', \lambda_{\#}) = \begin{cases} \underbrace{\lambda_{\#}(e) \cdot H(\#_{\text{subtrace}}(e), \lambda_{\#}) \cdot H(\sigma', \lambda_{\#})}_{\text{event label plus substraces}} & \text{if } \#_{\text{subtrace}}(e) \neq \perp \\ \underbrace{\langle \lambda_{\#}(e) \rangle \cdot H(\sigma', \lambda_{\#})}_{\text{no substraces}} & \text{otherwise} \end{cases}$$

The atomic hierarchical event log  $L_{H,A}$  is derived as follows:

$$L_{H,A} = [H(\sigma, \lambda_{\#}) \mid \sigma \in L]$$

For example, consider Case 1 from the hierarchical event log in Table 6.2 again. Suppose we filter out the complete events and use the abbreviated activity classifier (e.g.,  $M$  for `main.main()`,  $B$  for `B.process()`, see Table 6.1). Table 6.3 shows how we use hierarchical concatenation and the subtrace attribute to “assemble” the resulting atomic (sub)traces. Table 6.4 shows the hierarchical structure of the resulting atomic hierarchical trace.

Observe how we can use the utility functions from Definition 6.2.3 to inspect the hierarchical trace from Tables 6.3 and 6.4:

$$\begin{aligned} \|H(\text{Case 1}, \lambda_{\#})\| &= 4 \\ H(\text{Case 1}, \lambda_{\#})|_1^* &= H(\#_{\text{subtrace}}(\text{event 1.1}), \lambda_{\#}) \\ H(\text{Case 1}, \lambda_{\#})|_2^* &= H(\#_{\text{subtrace}}(\text{event 1.4}), \lambda_{\#}) \\ H(\text{Case 1}, \lambda_{\#})|_3^* &= H(\#_{\text{subtrace}}(\text{event 1.7}), \lambda_{\#}) \end{aligned}$$

**Table 6.3:** Example snippet of the atomic hierarchical event log derived from Table 6.2. Each row shows how the atomic (sub)traces are “assembled” into the atomic hierarchical trace for Case 1.

Trace	Hierarchical Activity Sequence
$H(\#_{\text{subtrace}}(\text{event 1.7}), \lambda_{\#}) =$	$\langle a \rangle$
$H(\#_{\text{subtrace}}(\text{event 1.4}), \lambda_{\#}) =$	$\langle s1, B.a, s2 \rangle$
$H(\#_{\text{subtrace}}(\text{event 1.1}), \lambda_{\#}) =$	$\langle i, B.s1, B.B.a, B.s2, o \rangle$
$H(\text{Case 1}, \lambda_{\#}) =$	$\langle M.i, M.B.s1, M.B.B.a, M.B.s2, M.o \rangle$

**Table 6.4:** The hierarchical structure for Case 1 as given in Table 6.3. Each column is one event, and each row is a level in the hierarchy. Each cell shows both the abbreviated and full activity name.

<i>M</i> Main.main()	<i>M</i> Main.main()	<i>M</i> Main.main()	<i>M</i> Main.main()	<i>M</i> Main.main()
<i>i</i> Main.input()	<i>B</i> B.process()	<i>B</i> B.process()	<i>B</i> B.process()	<i>o</i> Main.output()
	<i>s1</i> B.stepPre()	<i>B</i> B.process()	<i>s2</i> B.stepPost()	
		<i>a</i> A.process()		

## 6.2.4 Transformations – Heuristics for Hierarchy

In the above sections, we used the example of a software execution to demonstrate the concepts of hierarchical event logs. In this example, we used the implicit interval containment to infer a hierarchy. In practice, there are many sources that can be used to construct a hierarchy. In this section, we will discuss a few common heuristics to transform an *event log* (Definition 2.3.3) into a *hierarchical event log* (Definition 6.2.1).

### Nested Intervals

The *nested intervals* heuristic uses the lifecycle attribute to view an event log as a collection of intervals. By inferring an interval containment relation, a hierarchy is built, as shown in the above sections.

A typical application of this heuristic is the call stack behavior associated with a software execution logged at the method level, see the example from Table 6.1 and Figure 6.1 we used in the above sections.

Another typical application is to establish the relation between component or application interfaces. For example, like in the method/call stack example used above, whenever one interface function uses another interface function, this is manifested in the event log as a call relation or nested interval. By interpreting such events using the nested intervals heuristic, one can infer which interfaces are used (called by) a provided interface, and thus by extension also how components are related and referring to each other.

### Structured Names

The *structured names* heuristic uses a predefined pattern to infer a hierarchy from activity names. In some cases, an exact pattern can be specified as, for example, a regular expression. In other cases, one can identify a so-called split symbol to interpret path-like activity name structures.

For example, suppose the activities in an event log all have the following pattern:  $\langle \text{phase} \rangle \_ \langle \text{step} \rangle$ . Then we can create a hierarchical log where the top level captures the behavior amongst phases, and the level below that captures the steps for each phase. For example, the atomic trace  $\langle \text{Prepare\_config}, \text{Prepare\_input}, \text{Process\_compute} \rangle$  can be interpreted as the hierarchical trace with the events  $\langle \text{Prepare}, \text{config} \rangle$  followed by  $\langle \text{Prepare}, \text{input} \rangle$  and  $\langle \text{Process}, \text{compute} \rangle$ .

As an example of the split symbol approach, suppose that the activities in an event log are packages names which can be splitted on the dot (.) symbol. Then we can capture the static structure or “architecture” of a piece of software. For example, the activity `org.processmining.Main.main()` can be interpreted as the hierarchical event label sequence  $\langle \text{org}, \text{processmining}, \text{Main}, \text{main}() \rangle$ .

### Multiple Attributes

The *multiple attributes* heuristic uses a sequence of classifiers or attribute names to infer a hierarchical event label sequence. This heuristic is very effective where events have been annotated with external information such as source-code information, domain knowledge, or patterns inferred by domain experts and other (ProM) tools.

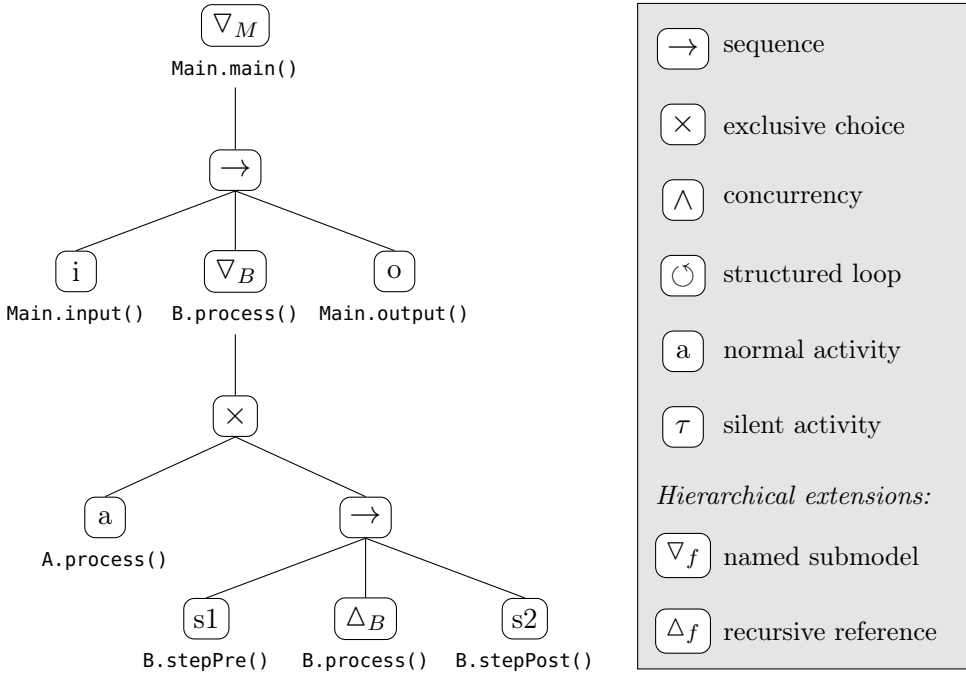
For example, suppose the events in an event log have a *step* attribute and a *phase* attribute. Then, like before, we can create a hierarchical log where the top level captures the behavior amongst phases, and the level below that captures the steps for each phase. For example, consider event  $e$  with  $\#_{\text{phase}}(e) = \text{Prepare}$  and  $\#_{\text{step}}(e) = \text{config}$ . Using the sequence  $\langle \text{phase}, \text{step} \rangle$  to infer our hierarchical log, we would interpret  $e$  as the hierarchical event label sequence  $\langle \text{Prepare}, \text{config} \rangle$ .

## 6.3 Hierarchical Process Trees

In this section, we extend the *process tree* notation (Section 2.2.6, page 38), with support for hierarchies in the form of named subprocesses and recursions. In our extended notation, called the *hierarchical process tree*, we add a new tree operator  $\nabla_f$  to represent the *named submodel* and add a new tree leaf  $\Delta_f$  to denote a *recursive reference*.

### 6.3.1 Example Model of a Program Execution

We will explain the hierarchical extensions using the example hierarchical process tree in Figure 6.2. This tree is modeled based on the event log in Table 6.1,



**Figure 6.2:** Example hierarchical process tree for the event log in Table 6.1. This tree models concrete runtime behavior for the program in Listing 6.1.

see also the program in Listing 6.1. The root node is a *named submodel* operator ( $\nabla_M$ ) modeling the `Main.main()` function. The subtree of  $\nabla_M$  models what happens during this submodel. In this case, a sequence ( $\rightarrow$ ) of `Main.input()`, `B.process()`, and `Main.output()`. Note that `B.process()` is again modeled with a *named submodel* operator ( $\nabla_B$ ). Observe that, compared to the program in Listing 6.1, this tree models runtime execution behavior where the reference `inst.process()` is resolved to `B.process()`. The submodel of `B.process()` ( $\nabla_B$ ) is defined by the tree rooted in the choice node ( $\times$ ): either `A.process()` can be performed, or the sequence of `B.stepPre()`, `B.process()`, and `B.stepPost()` can be performed. Note that this second `B.process()` is modeled as a *recursive reference* ( $\Delta_B$ ). This means that this leaf references back to the named submodel of the same name, i.e., during  $\Delta_B$  the submodel of  $\nabla_B$  is performed. This construction allows for multiple recursive executions of  $\nabla_B$ . Hence, like with structured loops, the language of this tree is infinite.

The hierarchical process tree in Figure 6.2 can also be represented textually:

$$\nabla_M(\rightarrow(i, \nabla_B(\times(a, \rightarrow(s1, \Delta_B, s2))), o))$$

The same submodel may appear multiple times in the same hierarchical process tree. For example, the tree  $\rightarrow(\nabla_A(a), \nabla_A(b))$  models a sequence of two

submodels named  $A$ , where the first time the submodel is defined in terms of  $a$ , and the second time in terms of  $b$ .

### 6.3.2 Syntax and Semantics

To formalize hierarchical process trees, we extend the notion of process trees from Definition 2.2.11 on page 40 by introducing the following syntax and semantics.

**Definition 6.3.1 — Hierarchical Process Tree.** We formally define *hierarchical process trees* recursively. We assume a finite alphabet  $\mathbb{A}$  of activities with  $\nabla, \Delta \notin \mathbb{A}$  and a set  $\otimes$  of operators to be given. The symbol  $\tau \notin \mathbb{A}$  denotes the silent activity.

We define the following base cases for hierarchical process trees:

$a$  any  $a \in (\mathbb{A} \cup \{\tau\})$  is a (silent) activity leaf;

$\Delta_f$  **Hierarchical extension:** denotes a *recursive reference* to a named subtree ancestor with name  $f \in \mathbb{A}$  (see below).

Let  $Q_1, \dots, Q_n$  with  $n > 0$  be hierarchical process trees and let  $\otimes \in \otimes$  be a hierarchical process tree operator, then  $\otimes(Q_1, \dots, Q_n)$  is a hierarchical process tree. We consider the following hierarchical process tree operators:

$\rightarrow$  denotes a *sequence* or the sequential composition of all subtrees;

$\times$  denotes an *exclusive choice* or XOR choice between one of the subtrees;

$\wedge$  denotes *concurrency* or the parallel composition of all subtrees;

$\circlearrowleft$  denotes the *structured loop* or redo loop with loop body  $Q_1$  and alternative loop back paths  $Q_2, \dots, Q_n$  (with  $n \geq 2$ );

$\nabla_f$  **Hierarchical extension:** denotes the *named subtree* with name  $f \in \mathbb{A}$  and subtree  $Q_1$  (i.e.,  $n = 1$ ).

The intuition behind the hierarchical operators is as follows. We can name any (sub)tree  $Q$  using the *named subtree*  $\nabla_f$  operator. This effectively prefixes the language of  $Q$  with  $f$  and allows us to refer back to this prefixed language. At any point in the subtree of  $\nabla_f$ , we can reference back to this prefixed language using the *recursive reference*  $\Delta_f$  leaf.

The semantics of hierarchical process trees are defined by extending the language function  $\mathcal{L}(Q)$  (see also Definition 2.2.12 on page 40).

**Definition 6.3.2 — Hierarchical Process Tree Semantics and Language.** In this definition, we use the notations from Definition 6.2.3 and the semantics previously defined in Definition 2.2.12. Let  $\mathbb{A}^\Delta = \mathbb{A} \cup \{\Delta_f \mid f \in \mathbb{A}\}$ . Below, we define a language of the type  $\mathcal{B}(((\mathbb{A}^\Delta)^*)^*)$ , i.e., a set of traces, where each symbol/event in a trace is a list of activity labels and (unresolved) recursive references, i.e., each symbol  $e$  is a list such that  $e \in (\mathbb{A}^\Delta)^*$ .



First, we define the language of the *recursive reference*  $\Delta_f$  leaf. We will simply “mark” this leaf in the language, indicating that we need to resolve the mark later. We define:

$$\mathcal{L}(\Delta_f) = \{ \langle \Delta_f \rangle \} \text{ for } f \in \mathbb{A}$$

The *named subtree*  $\nabla_f$  operator prefixes the language of its subtree with the activity label  $f \in \mathbb{A}$  and resolves all equally-named recursive references  $\Delta_f$ . We will use the function  $\Psi_{\mathcal{L}}(f, L, e)$  to resolve the recursive reference markers, as defined below. Function  $\Psi_{\mathcal{L}}$  scans each event’s label sequence  $e \in \mathbb{A}^*$  for recursive references and resolve the event’s language  $\Psi_{\mathcal{L}}(f, L, e) = L' \subseteq (\mathbb{A}^*)^*$ . For  $\nabla_f(Q)$  we define the language-join function  $\nabla_{\mathcal{L}}$ :

$$\begin{aligned} \nabla_{\mathcal{L}}(f, L) = \{ f.(\sigma' \cdot \dots \cdot \sigma'_n) \mid \langle e, \dots, e_n \rangle \in L & \text{ for } f \in \mathbb{A} \\ \wedge \forall 1 \leq i \leq n : \sigma'_i \in \Psi_{\mathcal{L}}(f, L, e_i) \} \end{aligned}$$

The function  $\Psi_{\mathcal{L}}(f, L, e)$  scans an event’s label sequence label by label. When a recursive reference is encountered, the name is checked. If it equals  $f$ , we resolve the recursion by using the language  $\nabla_{\mathcal{L}}(f, L)$ , i.e., we recursively use the language defined at the named subtree. If it does not match, we keep the recursion marker.

$$\begin{aligned} \Psi_{\mathcal{L}}(f, L, \varepsilon) &= \{ \varepsilon \} \\ \Psi_{\mathcal{L}}(f, L, a.e') &= \{ a.(\sigma') \mid \sigma' \in \Psi_{\mathcal{L}}(f, L, e') \} \text{ for } a \in \mathbb{A}, e' \in \mathbb{A}^* \\ \Psi_{\mathcal{L}}(f, L, \langle \Delta_g \rangle) &= \begin{cases} \nabla_{\mathcal{L}}(f, L) & \text{if } f = g \\ \{ \langle \Delta_g \rangle \} & \text{if } f \neq g \end{cases} \text{ for } g \in \mathbb{A} \end{aligned}$$

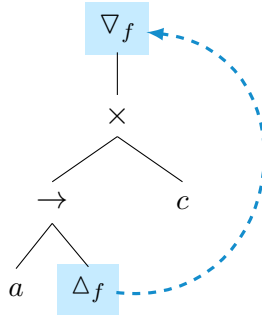
Note that when modelling software behavior, cases like inheritance, method overloading, and dynamic binding are resolved at the labelling level. See for example the overloading of the method `process()` in Listing 6.1 and Figure 6.2.

■ **Example 6.1** The following examples demonstrate the basics of the above language semantics, without recursion:

$$\begin{aligned} \mathcal{L}(\times(\rightarrow(a, b), c)) &= \{ \langle a, b \rangle, \langle c \rangle \} \\ \mathcal{L}(\nabla_f(\times(\rightarrow(a, b), c))) &= \{ \langle f.a, f.b \rangle, \langle f.c \rangle \} \\ \mathcal{L}(\nabla_f(\times(\rightarrow(a, \nabla_g(b)), c))) &= \{ \langle f.a, f.g.b \rangle, \langle f.c \rangle \} \end{aligned}$$

■

■ **Example 6.2** To demonstrate the basics of the recursion semantics, consider the tree  $\nabla_f(\times(\rightarrow(a, \Delta_f), c))$ , as depicted in Figure 6.3. If we execute the right subtree at the choice  $\times$  operator, we get the trace  $\langle f.c \rangle$ . If we execute the



**Figure 6.3:** Example recursive hierarchical process tree. The grey arrow indicates the “jumping back” in the language of the recursive reference  $\Delta_f$ .

left subtree, we encounter a recursion on  $f$ , “jumping back” to the root  $\nabla_f$ . Hence, if we execute the left subtree once, and in the recursion choose the right subtree, we get the trace  $\langle f.a, f.f.c \rangle$ . With two executions of the left subtree, we get the trace  $\langle f.a, f.f.a, f.f.f.c \rangle$ . We write this tree’s language as:

$$\mathcal{L}(\nabla_f(\times(\rightarrow(a, \Delta_f), c))) = \{ \langle f.c \rangle, \langle f.a, f.f.c \rangle, \langle f.a, f.f.a, f.f.f.c \rangle, \dots \}$$

The tree from Figure 6.2 has the language shown below. Observe that  $H(\text{Case 1}, \lambda_{\#})$  from Table 6.3, is in this language.

$$\begin{aligned} \mathcal{L}(\nabla_M(\rightarrow(i, \nabla_B(\times(a, \rightarrow(s1, \Delta_B, s2))), o))) = \\ \{ \langle M.i, M.B.a, M.o \rangle, \langle M.i, M.B.s1, M.B.B.a, M.B.s2, M.o \rangle, \dots \} \end{aligned}$$

Below are some more complex recursive examples. These examples show how recursive references can cross multiple named submodels, how mutual recursions can be specified, and how a loop instead of a choice can be used to specify a base and a recursive alternative.

$$\begin{aligned} \mathcal{L}(\nabla_f(\nabla_g(\times(\rightarrow(a, \Delta_f), c))) &= \{ \langle f.g.c \rangle, \langle f.g.a, f.g.f.g.c \rangle, \\ &\quad \langle f.g.a, f.g.f.g.a, f.g.f.g.f.g.c \rangle, \dots \} \\ \mathcal{L}(\nabla_f(\nabla_g(\times(a, \Delta_f, \Delta_g))) &= \{ \langle f.g.a \rangle, \langle f.g.g.a \rangle, \langle f.g.g.g.a \rangle, \langle f.g.f.g.a \rangle, \\ &\quad \langle f.g.f.g.g.a \rangle, \langle f.g.g.f.g.a \rangle, \dots \} \\ \mathcal{L}(\nabla_f(\odot(a, \Delta_f))) &= \{ \langle f.a \rangle, \langle f.a, f.f.a, f.a \rangle, \\ &\quad \langle f.a, f.f.a, f.a, f.f.a, f.a \rangle, \langle f.a, f.f.a, f.f.f.a, f.f.a, f.a \rangle, \dots \} \end{aligned}$$

■

Observe that hierarchical process trees are not automatically sound. Below, we define and discuss soundness for hierarchical process trees.

**Definition 6.3.3 — Sound Hierarchical Process Tree.** A hierarchical process tree  $Q$  is *sound* if and only if its language  $\mathcal{L}(Q)$ :

1. does not contain unresolved recursion markers:

$$\forall \sigma \in \mathcal{L}(Q) \wedge e \in \sigma : \neg(\exists f \in \mathbb{A} : \Delta_f \in e)$$

2. every resolved recursion has the option to terminate:

$$\begin{aligned} \forall \nabla_f(Q') \in \text{enum}(Q) : & \quad \text{for all named subtrees } \nabla_f \\ (\exists \sigma \in \mathcal{L}(Q') \wedge e \in \sigma : \Delta_f \in e) \Rightarrow & \quad \text{if there exists a matching } \Delta_f, \text{ then} \\ (\exists \sigma \in \mathcal{L}(Q') : (\forall e \in \sigma : \Delta_f \notin e)) & \quad \text{there exists a non-}\Delta_f \text{ alternative} \end{aligned}$$

All of the example trees shown above are sound. The following trees, for example, are *not sound*:

$\Delta_f$	<b>Issue:</b> language contains an unresolved recursion marker
$\nabla_f(\Delta_g)$	<b>Issue:</b> language contains an unresolved recursion marker
$\nabla_f(\Delta_f)$	<b>Issue:</b> language does not contain non-recursive alternative
$\nabla_f(\rightarrow(a, \Delta_f))$	<b>Issue:</b> language does not contain non-recursive alternative

## 6.4 Naïve Hierarchical Discovery

In this section, we introduce the *naïve hierarchical discovery* algorithm. In this algorithm, we extend the Inductive Miner (IM) framework (see Chapter 4) with support for the named subtree ( $\nabla_f$ ) operator. The recursive reference ( $\Delta_f$ ) operator will be covered in Section 6.5. In the remainder of this section, Section 6.4.1 gives an overview of the algorithm, Section 6.4.2 details our naïve hierarchical extensions, Section 6.4.3 shows additional discovery examples, and Section 6.4.4 covers the discovery guarantees maintained for our extension.

### 6.4.1 Algorithm Overview

For the *naïve hierarchical discovery* algorithm, we will follow the traditional IM divide and conquer approach (see also Section 4.2 on page 74). That is, given a log  $L$ , we search for possible splits of  $L$  into sublogs  $L_1, \dots, L_n$ , such that these sublogs combined with a process tree operator  $\otimes \in \{\rightarrow, \times, \wedge, \circ\}$  can (at least) reproduce  $L$  again. The framework then recurses on the corresponding sublogs, repeats the above process, and returns the discovered submodels as normal. However, in the naïve hierarchical discovery algorithm, we will adopt slightly different base cases. Note that we will assume atomic hierarchical event logs for the remainder of this section.

For naïve hierarchical discovery, the *empty log* and *empty traces* base case (i.e., Base Case 4.1) remains unchanged. However, the *single activity* base case (i.e., Base Case 4.2) is modified to consider the head of each event's activity

sequence and check if there is a lower level in the hierarchy. We say there is lower level in the hierarchy iff there exists an event activity sequence of length two or more, i.e., an event  $e \in \mathbb{A}^*$  with  $|e| \geq 2$ . If the head of each event refers to the same activity  $f$  and there is a lower level in the hierarchy, then we will model a named subtree  $\nabla_f$ . For the subtree, we recurse on the reduced event log obtained by removing the head activity from each event, i.e., we recurse on the event log  $L|_1^*$ . Below, we will discuss these adaptations in detail. We will be using the example run in Table 6.5 as clarification.

### 6.4.2 Framework Extensions

Formally, in the *naïve hierarchical discovery* algorithm, we use the original cut detections, log splits, and fallbacks as defined in Section 4.2 on page 74. In this extension, we will:

- use alternative base cases as detailed below, and
- adapt the directly-follows graph construction slightly such that we can use the original cut detections over a hierarchical event log.

In the following formulas and examples, we will write  $\text{NHDiscover}(L)$  to refer to the *naïve hierarchical discovery* (NHD) algorithm, i.e., the naïve hierarchical instantiation of the IM framework (see Algorithm 6.1).

---

#### Algorithm 6.1: Naïve Hierarchical Discovery (NHD) Algorithm

---

**Input:** An event log  $L$ .

**Output:** A hierarchical process tree  $Q$  such that  $L$  fits  $Q$ .

**Description:** The function  $\text{NHDiscover}()$  recursively tries to discover a hierarchical process tree capturing (at least) the behavior in  $L$ . We use  $\perp$  to model when no valid base case or valid cut was found for the given log.

```

NHDiscover( $L$ )
1   $Q_{base} = \text{BaseCase}(L)$                                 // Base Cases 4.1, 6.1, and 6.2
2  if  $Q_{base} \neq \perp$  then
3      return  $Q_{base}$                                      // Returns either  $\tau$ ,  $a$ , or  $\nabla_f(\text{NHDiscover}(L|_1^*))$ 
4  if  $\varepsilon \notin L$  then
5       $(\otimes, (\Sigma_1, \dots, \Sigma_n)) = \text{FindCut}(L)$     // Cut Detections 4.1, 4.2, 4.3, and 4.4
6      if  $(\otimes, (\Sigma_1, \dots, \Sigma_n)) \neq \perp$  then
7           $(L_1, \dots, L_n) = \text{SplitLog}(L, (\otimes, (\Sigma_1, \dots, \Sigma_n)))$ 
8          return  $\otimes(\text{NHDiscover}(L_1), \dots, \text{NHDiscover}(L_n))$ 
9  return  $\text{Fallback}(L)$ 

```

---

#### Adapted Base Cases

The *naïve hierarchical discovery* algorithm uses Base Case 4.1 as defined on page 75 to handle the empty log/empty traces base case ( $\tau$ ). Instead of the normal activity base case, the naïve hierarchical discovery algorithm uses the Base Cases 6.1 and 6.2 as defined below to detect named submodels.

**Table 6.5:** Example Naïve Hierarchical Discovery on the log  $L = [\langle f.a, f.f.b \rangle]$ . The rows illustrate how the discovery progresses step by step. The highlights indicate the parts of the log and directly-follows graph used, and relate them to the corresponding partial process tree model that is discovered. The boxed nodes in the directly-follows graph indicate lower levels in the hierarchy; the dashed lines indicate the cuts. The corresponding YAWL model (see Section 2.2.2, page 33) is shown at the bottom.

Event Log	Sublog View	Directly-Follows Graph	Discovered Model
	$[\langle f.a, f.f.b \rangle]$		
	$[\langle a, f.b \rangle]$		
	$[\langle f.b \rangle]$		
	$[\langle b \rangle]$		

■ **Base Case 6.1 — Single Activity – No Hierarchy.**

*Condition:*  $L \neq [] \wedge (\exists a \in \mathbb{A} : \forall \sigma \in L : \sigma = \langle a \rangle)$

*Return:*  $a$

*Description:* The *Single Activity – No Hierarchy* base case applies when the traces in the log contain only events with a single activity label  $a$ . That is, for every trace  $\sigma \in L$  and every event  $e \in L$  we have  $|e| = 1 \wedge e = \langle a \rangle$ . This base case returns  $a$  as a leaf node.

*Example:* Consider step 2 in Table 6.5. After the sequence cut, we have a sublog consisting of one event labeled  $a$  with no lower level in the hierarchy. Hence, this base case holds and we return  $a$  as a leaf node. Similarly, in step 4 in Table 6.5, the log consists of one event labeled  $b$ , and there is no lower level in the hierarchy, resulting in the leaf  $b$ . Observe that this base case does not apply in step 1 since there is a lower level in the hierarchy.

■ **Base Case 6.2 — Single Activity – Lower Level in the Hierarchy.**

*Condition:*  $L \neq [] \wedge (\exists f \in \mathbb{A} : (\forall \sigma \in L \wedge e \in \sigma : \text{head}(e) = f) \wedge (\exists \sigma \in L \wedge e \in \sigma : |e| \geq 2))$

*Return:*  $\nabla_f(\text{NHDiscover}(L|_1^*))$

*Description:* The *Single Activity – Lower Level in the Hierarchy* base case applies when the head of each event refers to the same activity  $f$  and there exists an event activity sequence of length two or more. This base case returns the named subtree  $\nabla_f$  and recurses on the event log reduced by removing the head activity from each event, i.e., the event log  $L|_1^*$ .

*Example:* In step 1 in Table 6.5, the log consists only of events with head label  $f$ , and there is a lower level in the hierarchy. Hence, this base case holds, we return  $\nabla_f$ , and we recurse on the reduced event log as shown in step 2. Similarly, in step 3 in Table 6.5, the log consists of one event labeled  $f.b$  and there is a lower level in the hierarchy, resulting in the named submodel  $\nabla_f$  and a recursive discovery call.

**Cut Detection and the Directly-Follows Graph in a Hierarchical Event Log**

The cut detection is still performed as normal on the directly-follows graph  $G(L)$  of the (sub)log  $L$ . This graph is built using the head of each event's activity sequence. That is, the directly-follows graph constructed for the “current” or “topmost” level in the hierarchy. In Table 6.5, the graph at step 1 is constructed from  $\langle f, f \rangle$  (the head of each element in  $\langle f.a, f.f.b \rangle$ ), steps 2 and 3 use the graph based on  $\langle a, f \rangle$  (the head of each element in  $\langle a, f.b \rangle$ ), and step 4 uses the graph constructed from  $\langle b \rangle$ .

**6.4.3 Discovery Examples**

Table 6.5 shows a step-by-step example run of the *naïve hierarchical discovery* algorithm. Note that the recursive pattern on the named submodel  $\nabla_f$  is not discovered with this discovery algorithm.

Below are some more example logs and the corresponding discovered models. Note how multiple named submodels can be discovered and how a varying hierarchical depth can lead to  $\tau$  elements.

$$\begin{aligned}
\text{NHDiscover}([\langle f.a, f.b \rangle, \langle f.c \rangle]) &= \nabla_f(\text{NHDiscover}([\langle a, b \rangle, \langle c \rangle])) \\
&= \nabla_f(\times(\text{NHDiscover}([\langle a, b \rangle]), c)) \\
&= \nabla_f(\times(\rightarrow(a, b), c)) \\
\text{NHDiscover}([\langle f.a, f.g.f.b \rangle]) &= \nabla_f(\text{NHDiscover}([\langle a, g.f.b \rangle])) \\
&= \nabla_f(\rightarrow(a, \text{NHDiscover}([\langle g.f.b \rangle]))) \\
&= \nabla_f(\rightarrow(a, \nabla_g(\text{NHDiscover}([\langle f.b \rangle]))) \\
&= \nabla_f(\rightarrow(a, \nabla_g(\nabla_f(b)))) \\
\text{NHDiscover}([\langle f.a \rangle, \langle f \rangle]) &= \nabla_f(\text{NHDiscover}([\langle a \rangle, \varepsilon])) \\
&= \nabla_f(\times(\tau, \text{NHDiscover}([\langle a \rangle]))) \\
\text{NHDiscover}([\langle f.f \rangle]) &= \nabla_f(\text{NHDiscover}([\langle f \rangle])) \\
&= \nabla_f(f)
\end{aligned}$$

#### 6.4.4 Guarantees

The *naïve hierarchical discovery* (NHD) algorithm maintains the IM discovery guarantees from Section 4.3 on page 84 and returns a sound hierarchical process tree (Definition 6.3.3). In this section, we will discuss these discovery guarantees and properties. We refer the reader to Section A.2 on page 368 for the proofs.

##### Soundness and Termination

We start with three general properties of the NHD algorithm: soundness is guaranteed (Theorem 6.4.1), termination is guaranteed (Theorem 6.4.2).

**Theorem 6.4.1 — NHD guarantees soundness.** All models  $Q$  returned by the NHD algorithm are guaranteed to be sound.

*Proof.* See page 368. □

**Theorem 6.4.2 — NHD guarantees termination.** The NHD algorithm is guaranteed to always terminate.

*Proof.* See page 369. □

### Perfect Fitness

The perfect fitness guarantee in Theorem 6.4.3 states that all the log behavior is in the model discovered by the NHD algorithm.

**Theorem 6.4.3 — NHD guarantees fitness.** The NHD algorithm returns a model that fits the log. That is, given an event log  $L$ , the NHD algorithm returns a model  $Q$  such that  $L \subseteq \mathcal{L}(Q)$ .

*Proof.* See page 369. □

### Language Rediscoverability

The language rediscoverability property tells whether and under which conditions a discovery algorithm can discover a model that is language-equivalent to the original process. That is, given a system model  $Q$  and an event log  $L$  that is complete with respect to  $Q$  (for some notion of completeness), then we rediscover a model  $Q'$  such that  $\mathcal{L}(Q') = \mathcal{L}(Q)$ .

For the NHD algorithm, we will prove the language rediscoverability property for directly-follows complete logs (Definition 4.3.1) and for all hierarchical process trees that are:

- *Sound* hierarchical process trees (Definition 6.3.3), and
- In the *class of rediscoverable process trees* (Definition 4.3.2).

Observe how, by using the atomic hierarchical event logs from Definition 6.2.2, the notion of directly-follows complete logs as defined in Definition 4.3.1 is automatically defined over all levels of a hierarchical log.

To prove language-rediscoverability, we will use the proof framework used by the base Theorem 4.3.7 as listed on page 85:

1. Show that the base cases can be rediscovered.
2. Show that any root process tree operator can be rediscovered, proving that the cut criteria are correct.
3. Show that for all process tree operators, the graph cut yields the correct activity division and the log is correctly subdivided.
4. Finally, Theorem 6.4.6 uses the above lemmas and base Theorem 4.3.7 to prove language rediscoverability using induction on the model size.

Reusing base Theorem 4.3.7, for the NHD algorithm, we only have to show that the root tree operator can still be rediscovered (Lemma 6.4.4) and that the log is still correctly subdivided (Lemma 6.4.5).

**Lemma 6.4.4 — NHD selects the right tree operator.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the above model restrictions and let  $L$  be a log such that  $L \diamond_{df} Q$ . Then  $\text{NHDDiscover}(L)$  returns a tree with root  $\otimes$ .

*Proof.* See page 370. □



**Lemma 6.4.5 — NHD splits logs correctly.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the above model restrictions and let  $L$  be a log such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ . Then for the resulting sublogs  $L_i$  produced by NHD we have  $L_I \diamond_{df} Q_i \wedge L_i \subseteq \mathcal{L}(Q_i)$ .

*Proof.* See page 370. □

Using the above lemmas and base Theorem 4.3.7, we can prove language rediscoverability using induction on the model size.

**Theorem 6.4.6 — NHD guarantees language rediscoverability.** If the model restrictions detailed above hold for a process tree  $Q$ , then `NHDiscover` language-rediscovered  $Q$ , i.e.,  $\mathcal{L}(Q) = \mathcal{L}(\text{NHDiscover}(L))$  for any log  $L$  such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ .

*Proof.* See page 370. □

### Polynomial Runtime Complexity

The basic IM framework is implemented as a polynomial algorithm and scales well with large event logs. The NHD algorithm maintains a polynomial runtime complexity.

**Theorem 6.4.7 — NHD has polynomial runtime complexity.** The runtime complexity of the NHD algorithm is bounded by  $O((\|\mathbb{A}(L)\| + \|L\|) \cdot \|\mathbb{A}(L)\|^4 + (\|\mathbb{A}(L)\| + \|L\|) \cdot |L|)$ .

*Proof.* See page 371. □

Note that, compared with basic IM algorithm, although we depend on the additional  $\|L\|$  term, the hierarchical alphabet size factor  $\|\mathbb{A}(L)\|$  is bound to be smaller than the original alphabet size factor  $|\mathbb{A}(L)|$ . This results directly from the idea that  $\|\mathbb{A}(L)\|$  is defined as the maximum activity alphabet size *across all hierarchy layers*. That is, due to the hierarchical projection, the activity set is effectively divided across the hierarchical levels.

## 6.5 Recursion Aware Discovery

In this section, we introduce the *recursion aware discovery* algorithm. In this algorithm, we extend the Inductive Miner (IM) framework (see Chapter 4) with support for both the named subtree ( $\nabla_f$ ) operator and the recursive reference ( $\Delta_f$ ) operator. Section 6.5.1 gives an overview of the algorithm, Section 6.5.2 details our recursion aware extensions, Section 6.5.3 shows additional discovery examples, and Section 6.5.4 covers the discovery guarantees maintained for our extension.

### 6.5.1 Algorithm Overview

For the *recursion aware discovery* algorithm, we need a mechanism to detect and handle recursive references. We will assume atomic hierarchical event logs for the remainder of this section and we again use the traditional IM divide and conquer approach as a basis (see also Section 4.2 on page 74). However, at any point during this top-down divide and conquer approach, we need to check whether an event's activity label refers back to an existing named subtree. If so, the event at the current level of the hierarchy represents a recursive reference. For example, take a look Table 6.5 again. In step 3, we encounter a label  $f$  with a lower level in the hierarchy. Note that at this step, we already have discovered a named subtree labeled  $f$  (see step 1). Hence, if we would have used this knowledge, we could have modeled a recursive reference  $\Delta_f$  at step 3.

To capture the knowledge of all the discovered named subtrees on the path from the root node to the node currently being discovered, we introduce the notion of a *context path* as defined below.

**Definition 6.5.1 — Context Path.** Let  $Q$  be a hierarchical process tree and let  $Q'$  be a specific (to be discovered) node in  $Q$ . A *context path*  $C \in \mathbb{A}^*$  is a sequence of activities representing the labels of the named subtrees on the path from the root  $Q$  to  $Q'$ , i.e.:

$$C = \langle f \mid Q'' \in \text{path}(Q, Q') \wedge Q'' = \nabla_f \rangle$$

Using the context path, we can verify the recursion on  $f$  in step 3 of Table 6.5 using  $f \in C = \langle f \rangle$ . If we now would have modeled the recursive reference  $\Delta_f$  at step 3, then the definition of the named subtree  $\nabla_f$  would have to be updated accordingly. After all, in the recursive call to  $f$ , the event log recorded activity  $b$  instead of activity  $a$ . Hence, we need to be able to *delay the discovery* of the named subtree's definition, and *repeat the discovery* once new information is available after modeling recursive references. To support delayed and repeated discovery, we allow our algorithm to create and update sublogs for specific context paths. We write  $L(C)$  to denote the sublog associated with the context path  $C$ . We will be using these sublogs in the base cases below.

For recursion aware discovery, the *empty log* and *empty traces* base case (i.e., Base Case 4.1) remain unchanged. However, the *single activity* base case (i.e., Base Case 4.2) is modified to check if the head of each event refers to the same activity  $f$  and if there is a lower level in the hierarchy. Recall, we say there is lower level in the hierarchy if and only if there exists an event activity sequence of length two or more. If this is the case, then we check  $f \in C$ . If  $f \in C$  holds, recursion is detected, we return the leaf  $\Delta_f$  and update  $L(C')$  for the corresponding context path  $C'$ . If  $f \notin C$  holds, a named subtree is detected, we return the partial named submodel  $\nabla_f(?_{C'})$ , where  $?_{C'}$  is a

placeholder for the undiscovered model, and create the sublog  $L(C')$  for the corresponding context path  $C'$ . Whenever a sublog  $L(C)$  is created or updated, we repeat discovery for that sublog. During this rediscovery, new information may become available and sublogs are changed accordingly. Once the sublogs are no longer changed, we finish the discovery by replacing all placeholders  $?_C$  with their actual model, gluing all the results together.

Below, we will discuss these adaptations in detail. We will be using the example run in Table 6.6 as clarification. This example run uses the same event log as in Table 6.5, but shows how a recursive reference is discovered.

### 6.5.2 Framework Extensions

Formally, in the *recursion aware discovery* algorithm, we use the original cut detections, log splits, and fallbacks as defined in Section 4.2 on page 74. In this extension, we will:

- use alternative base cases as detailed below,
- adapt the directly-follows graph construction slightly such that we can use the original cut detections over a hierarchical event log, and
- use the extended framework in the context of Algorithm 6.2 to implement the idea of delayed and repeated discovery.

In the following formulas and examples, we will write  $\text{RADstep}(L, C)$  to refer to the recursion aware instantiation of the IM framework (see Algorithm 6.3). The actual *recursion aware discovery* (RAD) algorithm is given in Algorithm 6.2, which uses  $\text{RADstep}()$  to repeatedly discover process trees for

---

#### Algorithm 6.2: Recursion Aware Discovery (RAD) Algorithm

---

**Input:** An (atomic hierarchical) event log  $L$ .

**Output:** A hierarchical process tree  $Q$  such that  $L$  fits  $Q$ .

**Description:** The function  $\text{RADiscover}()$  repeatedly tries to discover process trees for all the sublogs  $L(C)$  using the recursive function  $\text{RADstep}(L, C)$ . Note that  $\text{RADstep}()$  is the recursion aware instantiation of the IM framework derived from Algorithm 4.1 on page 75, see Algorithm 6.3.

```

RADiscover(L)
1 // Discover the root model using the complete event log L and context path C = ε
2  $Q_{root} = \text{RADstep}(L, \epsilon)$ 
3 // As long as any sublog L(C) changed, (re)discover the corresponding submodel
4 while  $\exists C \in \mathbb{A}^* : L(C)$  changed do
5      $Q_C = \text{RADstep}(L(C), C)$ 
6 // Glue the partial models together
7 foreach node Q in the process tree  $Q_{root}$  (any order, including new children) do
8     if  $\exists C \in \mathbb{A}^* : Q = ?_C$  then
9         Substitute  $?_C$  with  $Q_C$  // I.e.,  $\nabla_f(?_C)$  becomes  $\nabla_f(Q_C)$ 
10 return  $Q_{root}$ 

```

---

**Algorithm 6.3:** Recursion Aware Discovery – Step**Input:** An event log  $L$ .**Output:** A hierarchical process tree  $Q$  such that  $L$  fits  $Q$ .**Description:** The function  $\text{RADstep}()$  recursively tries to discover a (partial) hierarchical process tree capturing (at least) the behavior in  $L$ . We use  $\perp$  to model when no valid base case or valid cut was found.

```

RADstep( $L$ )
1   $Q_{base} = \text{BaseCase}(L)$  // Base Cases 4.1, 6.1, 6.3, and 6.4
2  if  $Q_{base} \neq \perp$  then
3    return  $Q_{base}$  // Returns either  $\tau$ ,  $a$ ,  $\nabla_f(?_{C'})$ , or  $\Delta_f$ 
4  if  $\varepsilon \notin L$  then
5     $(\otimes, (\Sigma_1, \dots, \Sigma_n)) = \text{FindCut}(L)$  // Cut Detections 4.1, 4.2, 4.3, and 4.4
6    if  $(\otimes, (\Sigma_1, \dots, \Sigma_n)) \neq \perp$  then
7       $(L_1, \dots, L_n) = \text{SplitLog}(L, (\otimes, (\Sigma_1, \dots, \Sigma_n)))$ 
8      return  $\otimes(\text{NHDdiscover}(L_1), \dots, \text{NHDdiscover}(L_n))$ 
9  return  $\text{Fallback}(L)$ 

```

all the sublogs  $L(C)$ . Algorithm 6.2 implements the idea of delayed and repeated discovery. Below, we will first explain Algorithm 6.2 using an example. Afterwards, we will detail the new recursion aware base cases.

**Delayed and Repeated Discovery – Algorithm 6.2 Explained**

Below, we will go over Algorithm 6.2 using the concrete example discovery run from Table 6.6. We will relate each line of the algorithm to the concrete steps in the example run and explain why the algorithm performs these steps. Note how  $\text{RADstep}(L, C)$ , i.e., the recursion aware instantiation of the IM framework, is used multiple times to (re)discover the correct model.

**Alg. 6.2 Tab. 6.6**

The algorithm starts with the initial discovery on the full input log and uses  $\text{RADstep}(L, \varepsilon)$  to discover the partial root model  $Q_{root}$ . After this step, the sublog  $L(\langle f \rangle)$  is created due to the discovery of the named subtree  $\nabla_f(?_{\langle f \rangle})$ .

Since the sublog  $L(\langle f \rangle)$  was changed, we perform a separate discovery for the named subtree  $\nabla_f(?_{\langle f \rangle})$ . Using the context path  $C = \langle f \rangle$ , we use  $\text{RADstep}(L(\langle f \rangle), \langle f \rangle)$  to discover the partial model  $Q_{\langle f \rangle}$ . Afterwards, due to the discovery of the recursive reference  $\Delta_f$ , the sublog  $L(\langle f \rangle)$  is updated with the new information in step 3.

Due to the changed sublog  $L(\langle f \rangle)$ , we again use  $\text{RADstep}(L(\langle f \rangle), \langle f \rangle)$  for the same context path  $C = \langle f \rangle$  to rediscover the partial model  $Q_{\langle f \rangle}$ . This time, since the sublog  $L(\langle f \rangle)$  already contains all the behavior, no sublogs are changed in step 6.

Line 2

Step 1

Line 5

Steps 2  
and 3

Line 5

Steps 4,  
5 and 6

**Alg. 6.2 Tab. 6.6**

We now have discovered up-to-date submodels for all the sublogs. Therefore, Algorithm 6.2 continues with gluing the submodels together. In our example, the placeholder  $?_{\langle f \rangle}$  is replaced by the submodel  $Q_{\langle f \rangle}$ .

Line 9 n/a

Afterwards, the resulting complete model is returned. The last row of Table 6.6 shows this resulting model  $Q$ .

Line 10 model  $Q$ 

So far, we only detailed the high-level workings of Algorithm 6.2. Below, we will give the details of detecting and discovering the named submodels and recursive references.

**Adapted Base Cases**

The  $\text{RADstep}(L, C)$  algorithm used by the *recursion aware discovery* algorithm uses Base Case 4.1 as defined on page 75 and Base Cases 6.1, 6.3 and 6.4 as defined below.

**■ Base Case 6.3 — Single Activity – Named Subtree.**

*Condition:*  $L \neq [] \wedge (\exists f \in \mathbb{A} \wedge f \notin C : (\forall \sigma \in L \wedge e \in \sigma : \text{head}(e) = f) \wedge (\exists \sigma \in L \wedge e \in \sigma : |e| \geq 2))$

*Derived context path:*  $C' = C \cdot \langle f \rangle$

*Created sublog:*  $L(C') = L \uparrow_1^*$

*Return:*  $\nabla_f(?_{C'})$

*Description:* The *Single Activity – Named Subtree* base case applies when the head of each event refers to the same activity  $f \notin C$  and there exists an event activity sequence of length two or more. For this base case, we consider the derived context path  $C'$ . This base case returns the named subtree  $\nabla_f(?_{C'})$ , and creates the new sublog  $L(C')$ . Due to the new sublog, Line 5 in Algorithm 6.2 will discover a model  $Q_{C'}$  to substitute  $?_{C'}$ .

*Example:* Consider step 1 in Table 6.6. The log consists only of events with head label  $f$ , and there is a lower level in the hierarchy. Since  $f \notin C = \varepsilon$ , this base case holds, we return  $\nabla_f(?_{C'})$ , and we create  $L(C') = L \uparrow_1^*$  for further discovery. Observe that this base case does not apply in steps 3 and 6 since we have  $f \in C = \langle f \rangle$ .

**■ Base Case 6.4 — Single Activity – Recursive Reference.**

*Condition:*  $L \neq [] \wedge (\exists f \in \mathbb{A} \wedge f \in C : (\forall \sigma \in L \wedge e \in \sigma : \text{head}(e) = f) \wedge (\exists \sigma \in L \wedge e \in \sigma : |e| \geq 2))$

*Derived context path:*  $C' = C_1 \cdot \langle f \rangle$  **where**  $C = (C_1 \cdot \langle f \rangle \cdot C_2)$

*Updated sublog:*  $L(C') = L(C) \cup L \uparrow_1^*$

*Return:*  $\Delta_f$

*Description:* The *Single Activity – Recursive Reference* base case applies when the head of each event refers to the same activity  $f \in C$  and there exists an event activity sequence of length two or more. For this base case, we consider the derived context path  $C'$ . This base case returns the recursive reference

**Table 6.6:** Example Recursion Aware Discovery on the log  $L = [\langle f.a, f.f.b \rangle]$ . The rows illustrate how the discovery progresses step by step. The highlights indicate the parts of the log and directly-follows graph used, and relate them to the corresponding partial process tree model that is discovered. The boxed nodes in the directly-follows graph indicate lower levels in the hierarchy; the dashed lines indicate the cuts. A double line between rows indicates where a new  $\text{RADstep}(L, C)$  run starts. The corresponding YAWL model (see Section 2.2.2, page 33) and complete discovered tree are shown at the bottom.

Event Log	Sublog View	Directly-Follows Graph	Discovered Model
	<b>Input:</b> $[\langle f.a, f.f.b \rangle]$ with context $C = \varepsilon$ $L = [\langle f.a, f.f.b \rangle]$		$Q_{root}$ :
	<b>Input:</b> $[\langle a, f.b \rangle]$ with context $C = \langle f \rangle$ $L = [\langle f.a, f.f.b \rangle]$ $L(\langle f \rangle) = [\langle a, f.b \rangle]$		$Q_{\langle f \rangle}$ :
	<b>Input:</b> $[\langle f.b \rangle]$ with context $C = \langle f \rangle$ $L = [\langle f.a, f.f.b \rangle]$ $L(\langle f \rangle) = [\langle a, f.b \rangle]$		$Q_{\langle f \rangle}$ :
	<b>Input:</b> $[\langle a, f.b \rangle, \langle b \rangle]$ with context $C = \langle f \rangle$ $L = [\langle f.a, f.f.b \rangle]$ $L(\langle f \rangle) = [\langle a, f.b \rangle, \langle b \rangle]$		$Q_{\langle f \rangle}$ :
	<b>Input:</b> $[\langle a, f.b \rangle]$ with context $C = \langle f \rangle$ $L = [\langle f.a, f.f.b \rangle]$ $L(\langle f \rangle) = [\langle a, f.b \rangle, \langle b \rangle]$		$Q_{\langle f \rangle}$ :
	<b>Input:</b> $[\langle f.b \rangle]$ with context $C = \langle f \rangle$ $L = [\langle f.a, f.f.b \rangle]$ $L(\langle f \rangle) = [\langle a, f.b \rangle, \langle b \rangle]$		$Q_{\langle f \rangle}$ :
			$Q$ :

$\Delta_f$ , and updates the sublog  $L(C')$ . Due to the changed sublog, Line 5 in Algorithm 6.2 will rediscover a model  $Q_{C'}$  for the named subtree  $\nabla_f$ . Such a named subtree exists because  $f \in C$ .

*Example:* Consider step 3 in Table 6.6. The log consists only of events with head label  $f$ , and there is a lower level in the hierarchy. Since  $f \in C = \langle f \rangle$ , this base case holds, we return  $\Delta_f$ , and we update  $L(C') = L(C') \cup L|_1^*$  for further discovery. Note that, in step 3, this changes  $L(C')$  by adding the subtrace  $\langle b \rangle$ . In step 6, this base case is also applied, but since the sublog  $L(\langle f \rangle)$  already contains all the behavior, no sublogs are changed.

### Cut Detection and the Directly-Follows Graph in a Hierarchical Event Log

As with the naïve discovery, the cut detection is still performed as normal. For recursion aware discovery, we reuse the naïve discovery solution, see page 125.

## 6.5.3 Discovery Examples

Table 6.6 shows a step-by-step example run of the *recursion aware discovery* algorithm. Note that, in contrast to the example run in Table 6.5, the recursive pattern on the named submodel  $\nabla_f$  is discovered in this discovery algorithm. Below are some more example logs, the corresponding discovery runs (enumerated), and the resulting discovered models.

■ **Example 6.3** —  $\text{RADiscover}([\langle f.a, f.g.f.b \rangle]) = \nabla_f(\times(b, \rightarrow(a, \nabla_g(\Delta_f))))$ .

1.  $Q_{root} = \text{RADstep}([\langle f.a, f.g.f.b \rangle], C = \varepsilon) = \nabla_f(?_{\langle f \rangle})$
2.  $Q_{\langle f \rangle} = \text{RADstep}([\langle a, g.f.b \rangle], C = \langle f \rangle) = \rightarrow(a, \nabla_g(?_{\langle f.g \rangle}))$
3.  $Q_{\langle f.g \rangle} = \text{RADstep}([\langle f.b \rangle], C = \langle f, g \rangle) = \Delta_f$
4.  $Q_{\langle f \rangle} = \text{RADstep}([\langle a, g.f.b \rangle, \langle b \rangle], C = \langle f \rangle) = \times(b, \rightarrow(a, \nabla_g(?_{\langle f.g \rangle})))$  ■

■ **Example 6.4** —  $\text{RADiscover}([\langle f.f \rangle]) = \nabla_f(\times(\tau, \Delta_f))$ .

1.  $Q_{root} = \text{RADstep}([\langle f.f \rangle], C = \varepsilon) = \nabla_f(?_{\langle f \rangle})$
2.  $Q_{\langle f \rangle} = \text{RADstep}([\langle f \rangle], C = \langle f \rangle) = \Delta_f$
3.  $Q_{\langle f \rangle} = \text{RADstep}([\langle f \rangle, \varepsilon], C = \langle f \rangle) = \times(\tau, \Delta_f)$  ■

■ **Example 6.5** —  $\text{RADiscover}([\langle f.g.g.a \rangle, \langle f.g.f.g.a \rangle]) = \nabla_f(\nabla_g(\times(a, \Delta_f, \Delta_g)))$ .

1.  $Q_{root} = \text{RADstep}([\langle f.g.g.a \rangle, \langle f.g.f.g.a \rangle], C = \varepsilon) = \nabla_f(?_{\langle f \rangle})$
2.  $Q_{\langle f \rangle} = \text{RADstep}([\langle g.g.a \rangle, \langle g.f.g.a \rangle], C = \langle f \rangle) = \nabla_g(?_{\langle f.g \rangle})$
3.  $Q_{\langle f.g \rangle} = \text{RADstep}([\langle g.a \rangle, \langle f.g.a \rangle], C = \langle f, g \rangle) = \times(\Delta_f, \Delta_g)$
4.  $Q_{\langle f \rangle} = \text{RADstep}([\langle g.g.a \rangle, \langle g.f.g.a \rangle, \langle g.a \rangle], C = \langle f \rangle) = \nabla_g(?_{\langle f.g \rangle})$
5.  $Q_{\langle f.g \rangle} = \text{RADstep}([\langle g.a \rangle, \langle f.g.a \rangle, \langle a \rangle], C = \langle f, g \rangle) = \times(a, \Delta_f, \Delta_g)$  ■

- **Example 6.6** —  $\text{RADiscover}([\langle f.a, f.f.a, f.a \rangle]) = \nabla_f(\odot(a, \Delta_f))$ .
1.  $Q_{\text{root}} = \text{RADstep}([\langle f.a, f.f.a, f.a \rangle], \mathbf{C} = \varepsilon) = \nabla_f(?_{\langle f \rangle})$
  2.  $Q_{\langle f \rangle} = \text{RADstep}([\langle a, f.a, a \rangle], \mathbf{C} = \langle f \rangle) = \odot(a, \Delta_f)$
  3.  $Q_{\langle f \rangle} = \text{RADstep}([\langle a, f.a, a \rangle, \langle a \rangle], \mathbf{C} = \langle f \rangle) = \odot(a, \Delta_f)$
- 

### 6.5.4 Guarantees

The *recursion aware discovery* (RAD) algorithm maintains the IM discovery guarantees from Section 4.3 on page 84 and returns a sound hierarchical process tree (Definition 6.3.3). In this section, we will discuss these discovery guarantees and properties. We refer the reader to Section A.3 on page 371 for the proofs.

#### Soundness and Termination

We start with three general properties of the RAD algorithm: soundness is guaranteed (Theorem 6.5.1), termination is guaranteed (Theorem 6.5.2).

**Theorem 6.5.1** — **RAD guarantees soundness.** All models  $Q$  returned by the RAD algorithm are guaranteed to be sound.

*Proof.* See page 372. □

**Theorem 6.5.2** — **RAD guarantees termination.** The RAD algorithm is guaranteed to always terminate.

*Proof.* See page 372. □

#### Perfect Fitness

The perfect fitness guarantee in Theorem 6.5.3 states that all the log behavior is in the model discovered by the RAD algorithm.

**Theorem 6.5.3** — **RAD guarantees fitness.** The RAD algorithm returns a model that fits the log. That is, given an event log  $L$ , the RAD algorithm returns a model  $Q$  such that  $L \subseteq \mathcal{L}(Q)$ .

*Proof.* See page 373. □

#### Language Rediscoverability

The language rediscoverability property tells whether and under which conditions a discovery algorithm can discover a model that is language-equivalent to the original process. That is, given a system model  $Q$  and an event log  $L$  that is complete with respect to  $Q$  (for some notion of completeness), then we rediscover a model  $Q'$  such that  $\mathcal{L}(Q') = \mathcal{L}(Q)$ .



For the RAD algorithm, like with the NHD algorithm in Section 6.4.4 on page 127, we will prove the language rediscoverability property for directly-follows complete logs (Definition 4.3.1) and for all hierarchical process trees that are:

- *Sound* hierarchical process trees (Definition 6.3.3), and
- In the *class of rediscoverable process trees* (Definition 4.3.2).

To prove language-rediscoverability, we will again use the proof framework used by the base Theorem 4.3.7 as listed on page 85:

1. Show that the base cases can be rediscovered.
2. Show that any root process tree operator can be rediscovered, proving that the cut criteria are correct.
3. Show that for all process tree operators, the graph cut yields the correct activity division and the log is correctly subdivided.
4. Finally, Theorem 6.5.7 uses the above lemmas and base Theorem 4.3.7 to prove language rediscoverability using induction on the model size.

Reusing base Theorem 4.3.7, for the RAD algorithm, we only have to show that the root tree operator can still be rediscovered (Lemma 6.5.4), the recursive reference can be rediscovered (Lemma 6.5.5), and that the log is still correctly subdivided (Lemma 6.5.6).

**Lemma 6.5.4 — RAD selects the right tree operator.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the above model restrictions and let  $L$  be a log such that  $L \diamond_{df} Q$ . Then  $\text{RADiscover}(L)$  returns a tree with root  $\otimes$ .

*Proof.* See page 373. □

**Lemma 6.5.5 — RAD rediscovered the recursive reference leaf.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the above model restrictions with a leaf  $\Delta_f$  somewhere in  $Q$  and let  $L$  be a log such that  $L \diamond_{df} Q$ . Then  $\text{RADiscover}(L)$  returns a tree with a leaf  $\Delta_f$ .

*Proof.* See page 374. □

**Lemma 6.5.6 — RAD splits logs correctly.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the above model restrictions and let  $L$  be a log such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ . Then for the resulting sublog  $L(C')$  produced by RAD we have  $L(C') \diamond_{df} Q_{C'} \wedge L(C') \subseteq \mathcal{L}(Q_{C'})$ .

*Proof.* See page 374. □

Using the above lemmas and base Theorem 4.3.7, we can prove language rediscoverability using induction on the model size.

**Theorem 6.5.7 — RAD guarantees language rediscoverability.** If the model restrictions detailed above hold for a process tree  $Q$ , then **RADiscover** language-rediscovered  $Q$ , i.e.,  $\mathcal{L}(Q) = \mathcal{L}(\text{RADiscover}(L))$  for any log  $L$  such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ .

*Proof.* See page 374. □

### Polynomial Runtime Complexity

The basic IM framework is implemented as a polynomial algorithm and scales well with large event logs. The RAD algorithm maintains a polynomial runtime complexity.

**Theorem 6.5.8 — RAD has polynomial runtime complexity.** The runtime complexity of the RAD algorithm is bounded by  $O(\|L\| \cdot \|\mathbb{A}(L)\|^5 + \|L\| \cdot \|\mathbb{A}(L)\| \cdot |L|)$

*Proof.* See page 375. □

Note that, compared with basic IM algorithm, although we depend on the additional  $\|L\|$  term, the hierarchical alphabet size factor  $\|\mathbb{A}(L)\|$  is bound to be smaller than the original alphabet size factor  $|\mathbb{A}(L)|$ . This results directly from the idea that  $\|\mathbb{A}(L)\|$  is defined as the maximum activity alphabet size *across all hierarchy layers*. That is, due to the hierarchical projection, the activity set is effectively divided across the hierarchical levels.

## 6.6 Compatibility with Other Extensions

Both the naïve and recursion aware discovery, as presented in the above sections, provide a good basis for hierarchical process discovery. In this section, we will revisit the existing inductive miner extensions listed in Section 4.4 on page 86 and discuss how they can be used in combination with the hierarchical discovery extensions.

The *Inductive Miner – infrequent (IMf)* extension handles deviating and infrequent behavior by filtering the directly-follows graph according to some user-chosen frequency threshold when no cut could be found. The user can set this frequency threshold by using a so-called “path” slider in the user interface implementations. In addition, the single activity base case and empty traces fallback are slightly altered to take into account infrequent behavior. In our hierarchical setting, we can easily integrate with this extension by altering the base cases in a similar way to take into account infrequent behavior.

The *Inductive Miner – incompleteness (IMc)* extension handles incomplete behavior by using probabilistic activity relations in the cut detection. Since

the hierarchical extensions only adjust base cases, we can readily combine it with the incompleteness extension.

By using the rich hierarchical event logs from Definition 6.2.1, we can integrate with the *lifecycle aware* extension. Our hierarchical base cases can replace the  $L|_1^*$  projection by using the  $\#_{\text{subtrace}}(e)$  attributes directly.

Integration with the *directly-follows abstraction* variant for very large event logs is possible but tricky. This variant recurses on sub directly-follows graphs instead of sublogs. In our hierarchical setting, we can integrate this variant by annotating directly-follows nodes with sub directly-follows graphs whenever there is a lower level in the hierarchy. See also the boxed nodes in the graphs in Tables 6.5 and 6.6. In the recursion aware discovery algorithm, special care should be taken to track and update the subgraphs for the various context paths. Since only behavior is added whenever a recursive reference is detected, we can update subgraphs using a graph union operation.

## 6.7 Evaluation

In this section, we compare the *naïve hierarchical discovery (NHD)* and *recursion aware discovery (RAD)* algorithms against related, implemented techniques. The NHD and RAD algorithms are implemented in the *Statechart* plugin for the ProM framework, see also Chapter 10. Section 6.7.1 investigates the discovery results on a controlled example. Section 6.7.2 provides a comparison on running time and model quality. For large, real-life case studies and tool UI using the hierarchy techniques, see Chapter 12.

### 6.7.1 Evaluation using Synthetic Logs

In this evaluation, we focus on model understandability. We use a small synthetic example software log, mine a model with various discovery algorithms, and compare the resulting models on structure and visual appearance. The goal of this evaluation is to investigate how the NHD and RAD algorithms compare to existing algorithms in discovering accurate and understandable models when hierarchical behavior is present.

#### Methodology

For this evaluation, we revisit the program in Listing 6.1. We executed the program twice, once for  $i = 3$  and once for  $i = 4$ , and let `Main.input()` return an instance of `class B`. During execution, we traced and logged the start and end of each called method, resulting in an event log similar to the one shown in Table 6.1. This event log has 2 traces, 62 events, 14 activities (including lifecycles), and 7 hierarchical levels.

We use the program in Listing 6.1 and the model from Figure 6.2 as a baseline comparison. For the various discovered models, we use the default

visualization provided by the various tool implementations, and annotate the activities with their one-letter acronyms, see Table 6.7 for a summary. For the Inductive Miner and our hierarchical discovery techniques, we used the infrequent (IMf) extensions and indicate the “path” threshold. For “path” threshold, 1.0 means all behavior, 0.8 means 80% of the behavior (i.e., an 80/20 model, describing 80% of the behavior using a simpler 20% model).

For this evaluation, we provide the fitness and precision scores calculated on the Petri net translation, using the alignments-based technique from [21]. Recall from Section 1.1.2 on page 7: *Fitness* expresses the part of behavior in the event log that is also captured in the model. *Precision* expresses the part of the behavior in the model that is also present in the event log. A fitness of 1.0 indicates all behavior in the event log can be reproduced by the model; a lower fitness (minimum 0.0) means the modeled behavior represents the behavior in the event log less. A precision of 1.0 indicates all behavior in the model has been observed in the event log; a lower precision (minimum 0.0) means the modeled behavior is less supported by observations in the event log.

**Table 6.7:** One-letter acronyms for the activities used in the hierarchical synthetic evaluation. These acronyms correspond with the baseline model in Figure 6.2 for the program in Listing 6.1.

M	Main.main()	B	B.process()	Lifecycle suffixes: +s start +c complete
i	Main.input()	s1	B.stepPre()	
o	Main.output()	s2	B.stepPost()	
A	A.process()			

## Results

Figure 6.4 to 6.16 show the discovered models. Below, we will discuss each model in turn.

**Alpha miner [18]** – The Alpha miner result in Figure 6.7 shows a very disconnected model. Since no alpha-relations were inferred between most activities, most transitions are not connected to places and can fire at any time. This model gives us no information about how the different activities are causally related. The very low fitness score reflects this lack of information. Hence, this model does not aid in understanding the behavior.

**Fuzzy miner [75]** – The Fuzzy miner result in Figure 6.8 shows a reasonably structured model. Although the submodel structures are not explicit in this model, using *activity+lifecycle* labels, we get a decent idea about the interval containment relations. The causal relations of activities *M*, *i*, *o* and *B* can be correctly inferred. In addition, we can infer that during activity *B*, we can only finish by performing activity *A*, i.e., the termination or recursion

base case can be spotted. However, the recursive and nested nature of the  $B$  submodel cannot be deduced from this model. In addition, the loops and self-loops on  $B$ ,  $s1$ , and  $s2$  are misleading. There are no constraints shown on the number of starts and completes, and no constraint on the relation between the number of  $s1$  and  $s2$  executions. Using the *significance cutoff* in the Fuzzy miner to cluster behavior does not help in finding more structures in this model.

**Heuristics miner [192]** – The Heuristics miner result in Figure 6.4 again shows a reasonably structured model. This model has the same overall structure and issues as the *Fuzzy miner* model discussed above. In addition, the split-join semantics around the  $B$  activities are unclear, leading to soundness issues.

**ILP miner [204]** – The ILP miner result in Figure 6.5 again shows a reasonably structured model. This model has the same overall structure and issues as the *Fuzzy miner* model discussed above. Although this model has perfect fitness, the very low precision score reflects the lack of constraints and information in the  $B$  submodel.

**Genetic miner [10]** – The Genetic miner result in Figure 6.6 again shows a reasonably structured model. This model has the same overall structure and issues as the *Fuzzy miner* model discussed above. In addition, the split-join semantics around the  $B$  activities are unclear and the  $B+complete$  activity is disconnected from the main flow, leading to soundness issues.

**ETMd miner [46]** – The ETMd miner result in Figure 6.6 shows a trace model fitted to the longest trace (for input  $i = 4$ ). This model has a high fitness and precision, but it is clearly too overfitting and does not show us anything about the underlying recursive and repetitive structures. Hence, if we would rerun the software with a different input, and thus a different number of recursive calls, this model would not be able to support the resulting behavior. In short, this model does not aid in understanding the behavior.

**TS Regions [16]** – The Transition System miner using Regions result in Figure 6.10 again shows a reasonably structured model. This model has the same overall structure and issues as the *Fuzzy miner* model discussed above. Observe how, due to the lack of lifecycle information, the fitness score dropped.

**MINT ktails [190]** – The MINT ktails result in Figure 6.11 shows a trace model fitted to both traces. Like with the *ETMd miner* model discussed above, this model is clearly too overfitting, and does not show us anything about the underlying recursive and repetitive structures.

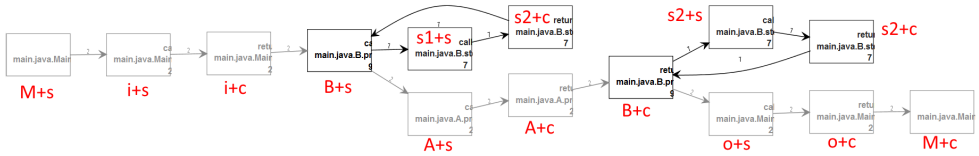
**MINT redblue [190]** – The MINT redblue result in Figure 6.13 shows a rather convoluted and inaccurate model. It is difficult to understand what the relations are between the activities  $A$ ,  $B$   $s1$ , and  $s2$ . In addition, the model incorrectly shows the optionality of activity  $i$ , and all information about the interval containment relations is lost. Even when ignoring the lack of

lifecycle information, completely incorrect traces are admitted by this model, e.g.:  $\langle M, B, o, o \rangle$  or  $\langle M, i, i, i, s2, A, o, o \rangle$ . The low fitness score confirms this.

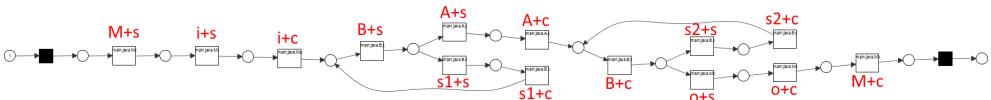
**Synoptic miner [37]** – The Synoptic miner result in Figure 6.12 again shows a reasonably structured model. Compared to the *Fuzzy miner* type of models discussed above, similar structures are present in this result. In addition, the activities  $s1$  and  $s2$  are better placed in this model than in the previous results. In this model  $s1$  and  $s2$  happen between successive executions of  $B$  and cannot be skipped. However, again the constraints between  $s1$  and  $s2$  as well as the concrete recursive nature of the  $B$  submodel cannot be deduced from this model.

**Inductive miner [130]** – The Inductive miner results in Figure 6.14 show structured but imprecise models. The *path* thresholds indicate the amount of behavior included: 1.0 is all behavior, 0.8 yields an 80/20 model. Activity  $M$  is placed in a strange loop with the remaining activities due to the interval containment relation. The causal relations of activities  $i$ ,  $o$  and  $B$  can be correctly inferred, but the behavior during activity  $B$  is again placed in a strange loop due to the interval containment relation. By interpreting these loops as “happens during” relations, we coincidentally get a decent idea about the interval containment relations. However, no constraints or relations can be deduced for the activities  $A$ ,  $s1$ , or  $s2$ . The relatively low fitness score reflects this lack of information.

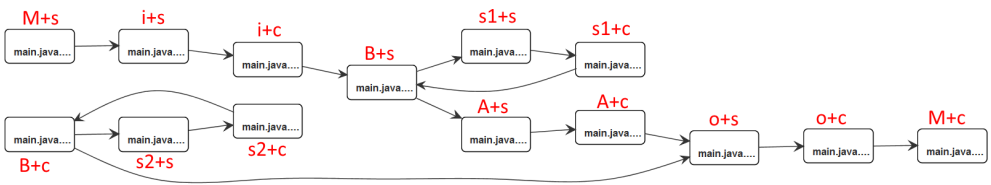
**NHD and RAD (this chapter)** – The Naïve Hierarchical Discovery (NHD) and Recursion Aware Discovery (RAD) algorithms from this chapter yield the models in Figure 6.15, 6.16, and 6.17. The NHD model clearly shows the correctly identified submodels. Although this model is in essence a trace model, it does clearly show the inferred containment relations via the submodels. Note that this model has perfect fitness and precision. In the RAD model the recursion was correctly detected and generalized, thereby reducing the visual complexity. Note that the discovered hierarchical process tree in Figure 6.15b exactly matches the baseline model shown in Figure 6.2. In addition, observe how the small statechart representation in Figure 6.16b clearly shows all the submodels, recursion relation and constraints (e.g., relation between the number of  $s1$  and  $s2$  executions), while at the same time being smaller and easier to read than the larger model in Figure 6.16a. By dividing the activity names into class and method names we obtain the message sequence diagrams in Figure 6.17. Via the use of lifelines and activation boxes, we made the nested calls and relation amongst classes explicit. Note that Figure 6.17b now explicitly shows how the recursion is terminated in class  $A$ .



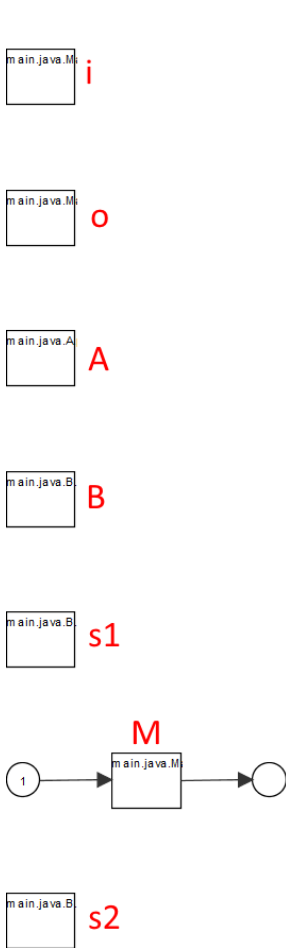
**Figure 6.4:** The Heuristics miner [192] result, model is unsound. Although this model is reasonably structured, we cannot deduce the recursive and nested nature of the  $B$  submodel and we cannot infer the constraint on the relation between the number of  $s1$  and  $s2$  executions.



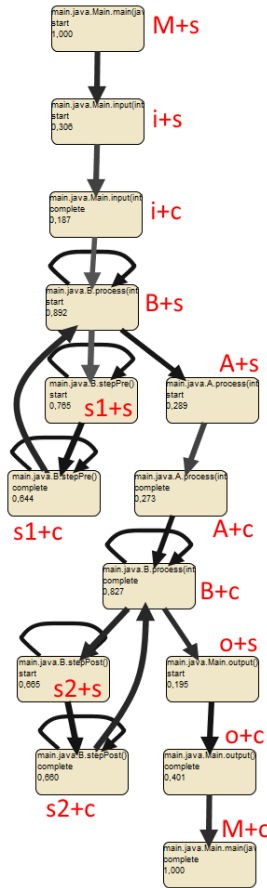
**Figure 6.5:** The ILP miner [204] result with fitness 1,00 and precision 0,19. Although this model is reasonably structured, we cannot deduce the recursive and nested nature of the  $B$  submodel and we cannot infer the constraint on the relation between the number of  $s1$  and  $s2$  executions.



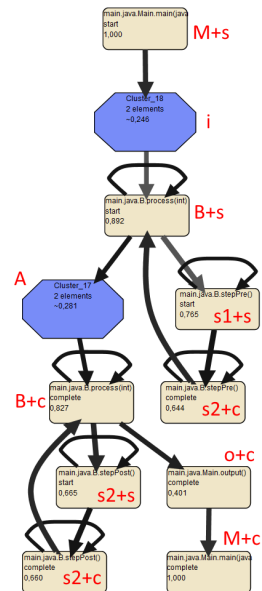
**Figure 6.6:** The Genetic miner [10] result, model is unsound. Although this model is reasonably structured, the  $B+complete$  activity is disconnected from the main flow, we cannot deduce the recursive, and nested nature of the  $B$  submodel and we cannot infer the constraint on the relation between the number of  $s1$  and  $s2$  executions.



**Figure 6.7:** The Alpha miner [18] result with fitness 0,97 and precision 0,15. The model is very disconnected and does not aid in understanding the behavior.



(a) Significance: 0.0



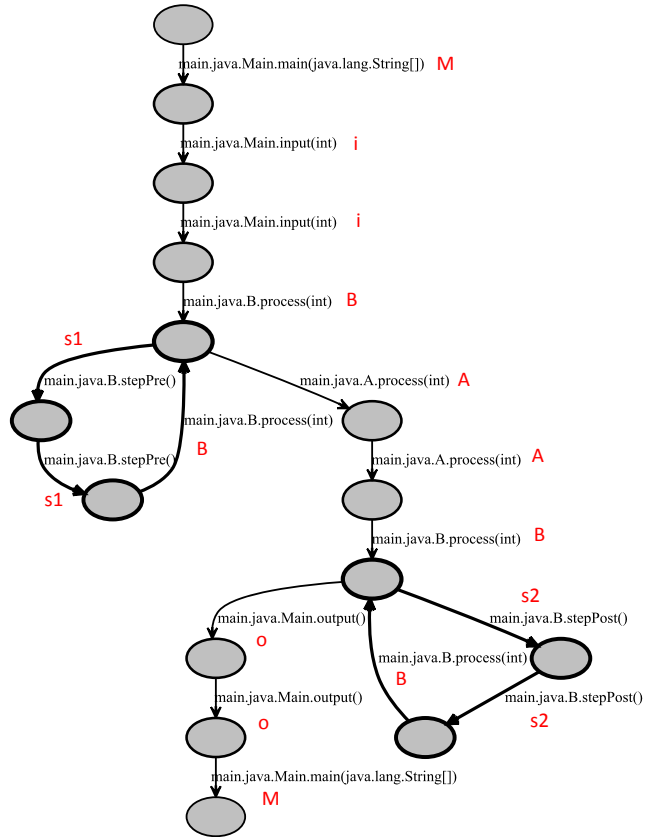
(b) Significance: 0.4

**Figure 6.8:** The Fuzzy miner [75] result. The *significance cutoff* threshold controls how much behavior is clustered (see the octagonal nodes *i* and *A*). Although this model is reasonably structured, we cannot deduce the recursive and nested nature of the *B* submodel and we cannot infer the constraint on the relation between the number of *s1* and *s2* executions.

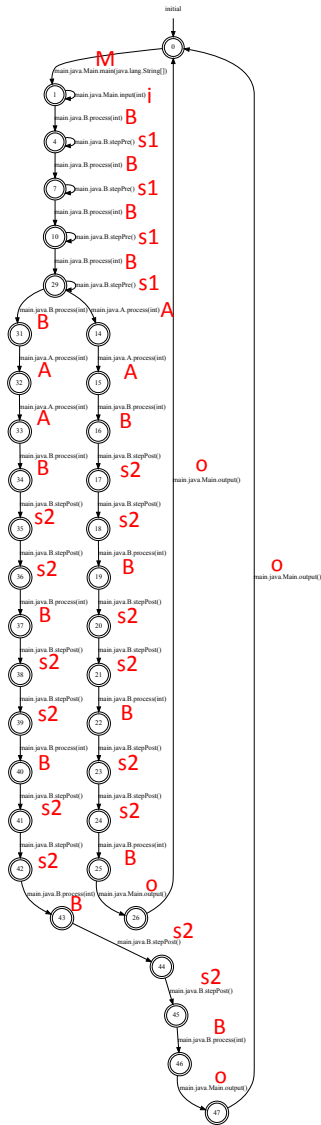




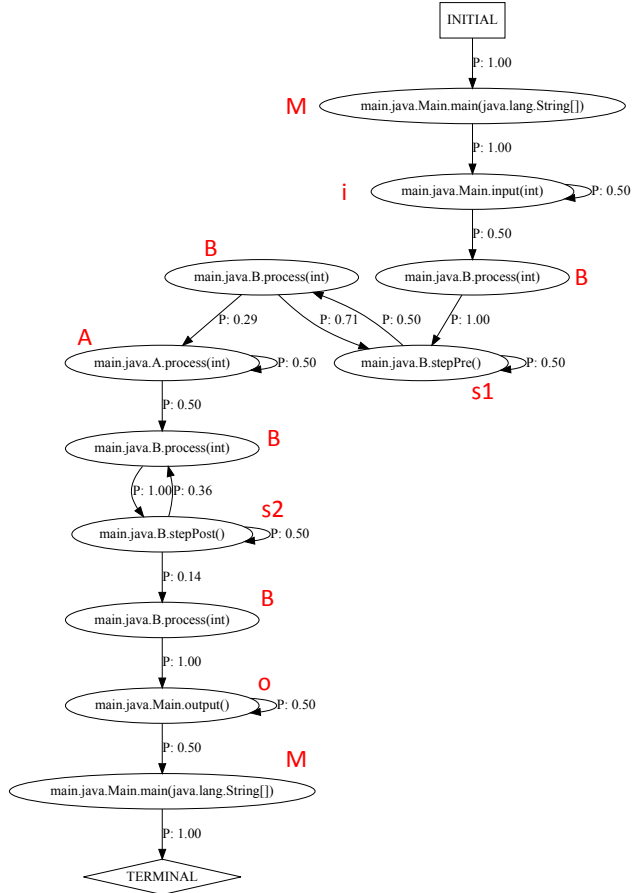
**Figure 6.9:** The ETMd miner [46] result with fitness 0,86 and precision 0,97. This model is clearly too overfitting and does not show us anything about the underlying recursive and repetitive structures.



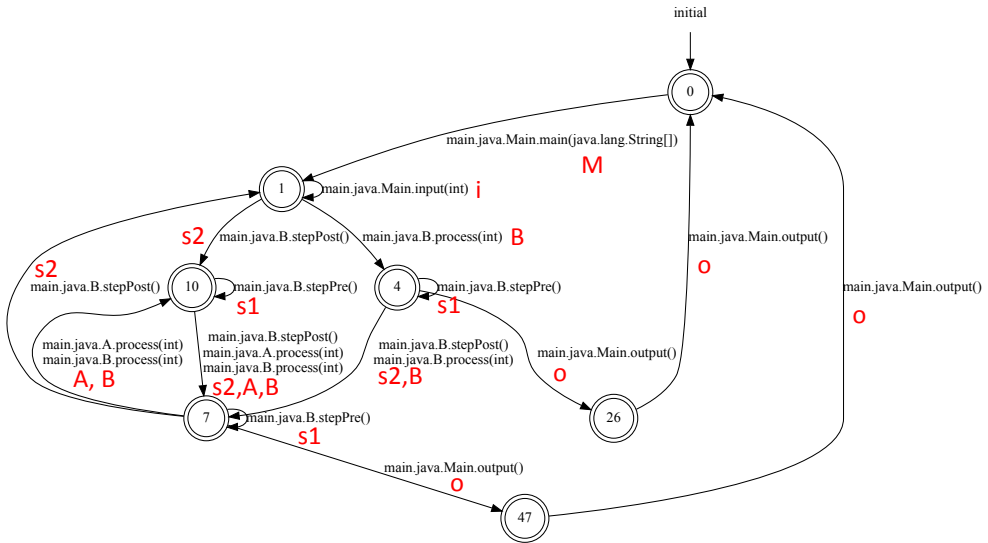
**Figure 6.10:** The Transition System miner using Regions [16] result with fitness 0,67 and precision 0,47. Although this model is reasonably structured, we cannot deduce the recursive and nested nature of the *B* submodel and we cannot infer the constraint on the relation between the number of *s1* and *s2* executions.



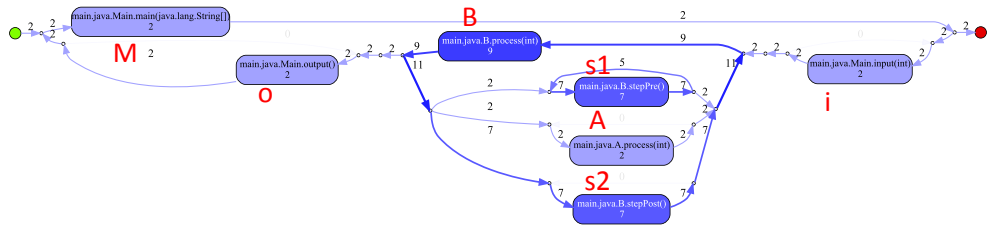
**Figure 6.11:** The MINT ktails [190] result with  $k=1$ , fitness 0,68 and precision 0,67. This model, although better than the ETMd model, is clearly too overfitting and does not show us anything about the underlying recursive and repetitive structures.



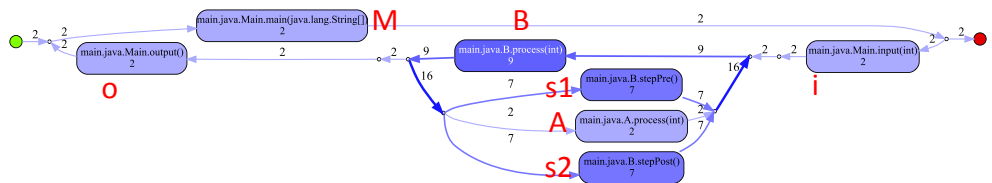
**Figure 6.12:** The Synoptic miner [37] result with fitness 1,00 and precision 0,66. This model is again reasonably structured and better shows the relations between  $B$ ,  $s1$ ,  $s2$ , and  $A$ . However, again we cannot deduce the recursive and nested nature of the  $B$  submodel and we cannot infer the constraint on the relation between the number of  $s1$  and  $s2$  executions.



**Figure 6.13:** The MINT redblue [190] result with  $k=1$ , fitness 0,68, and precision 0,67. This model is rather convoluted and inaccurate, and admits completely incorrect traces. It is difficult to understand what the relations are between the activities *A*, *B* *s1*, and *s2*.

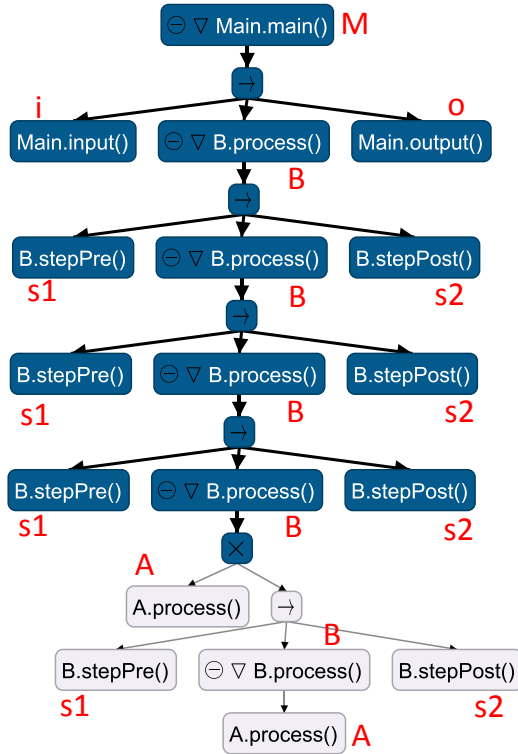


(a) Path 1.0, fitness 1,00, and precision 0,57.

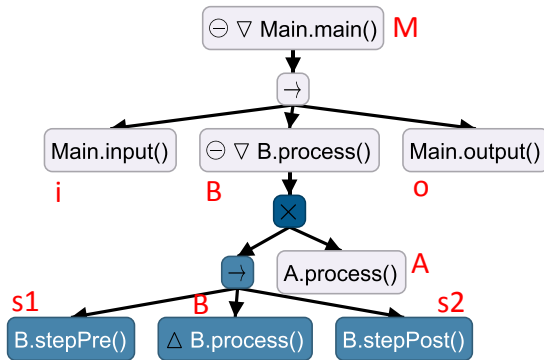


(b) Path 0.8, fitness 0,69, and precision 0,46.

**Figure 6.14:** The Inductive miner [130] result. The *path* thresholds indicate the amount of behavior included: 1.0 is all behavior, 0.8 yields an 80/20 model. These models are structured but imprecise: activities are placed in strange loops and various causal relations cannot be correctly inferred.

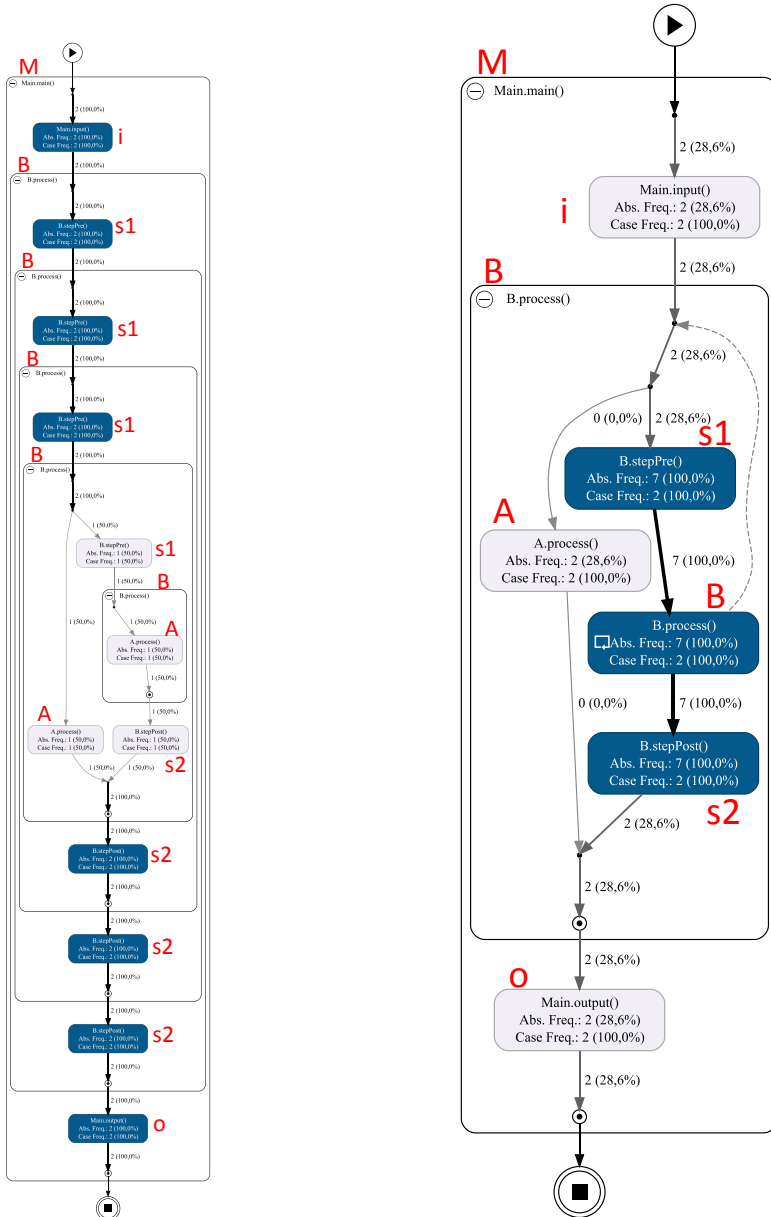


(a) Naïve Hierarchical Discovery (NHD), path 0.8, fitness 1,00, and precision 1,00.



(b) Recursion Aware Discovery (RAD), path 0.8, fitness 1,00, and precision 0,82.

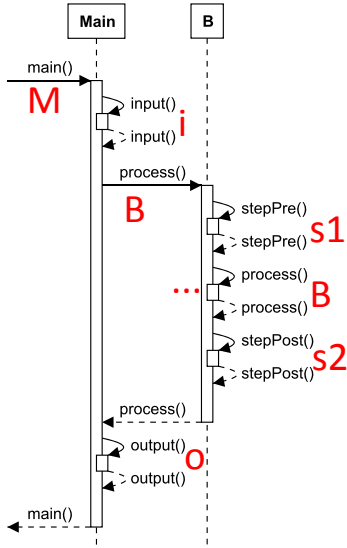
**Figure 6.15:** The NHD and RAD results visualized as process trees. The *path* thresholds indicate the amount of behavior included: 1.0 is all behavior, 0.8 yields an 80/20 model. These models clearly show the inferred containment relations via the submodels. In the RAD model the recursion was correctly detected and generalized, thereby reducing the visual complexity.



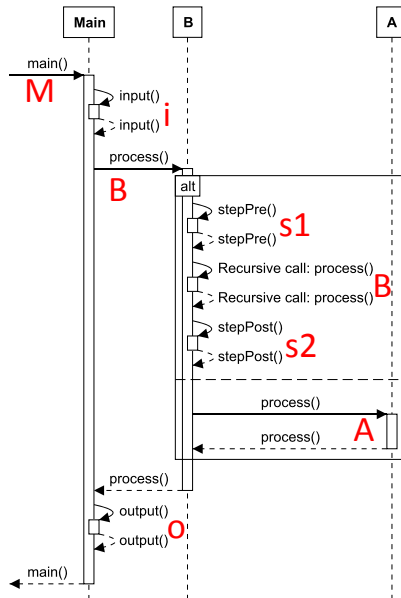
(a) NHD, path 0.8, fitness 1,00, and precision 1,00.

(b) RAD, path 0.8, fitness 1,00, and precision 0,82.

**Figure 6.16:** The NHD and RAD results from Figure 6.15 visualized as statecharts. In contrast to the process trees, these models explicitly show the control-flow arrows. Note how the RAD model explicitly shows the recursive relations and is more compact and easier to read than the NHD model.



(a) NHD, path 0.8, fitness 1,00, and precision 1,00. Only the first 3 hierarchical levels are visualized, the rest of the nested behavior is collapsed/hidden, see the dots (...).



(b) RAD, path 0.8, fitness 1,00, and precision 0,82.

**Figure 6.17:** The NHD and RAD results from Figure 6.15 visualized as message sequence diagrams. The activity names were divided into class names (lifelines) and method names (message arrows). Note how Figure 6.17b now explicitly shows how the recursion is terminated in class A.



### 6.7.2 Performance and Scalability Evaluation

In this section, we perform a comparative evaluation focusing on the performance and scalability of the various discovery algorithms. The discovery algorithms will be compared on running time and the resulting models will be compared on model quality in terms of fitness and precision.

#### Methodology

All of the algorithms in this comparison are invoked from a Java benchmark setup under the same operating conditions. For these experiments we used a laptop with an i7-4700MQ CPU @ 2.40 GHz, Windows 8.1 and Java SE 1.7.0 67 (64 bit) with 12 GB of allocated RAM.

For the running time, we measured the average running time and associated 95% confidence interval over 30 micro-benchmark executions, after 10 warmup rounds for the Java JVM. Each algorithm is allowed at most 30 seconds for completing a single model discovery. The time for loading event logs or Java classes is excluded from the measurements.

For the model quality, we use fitness and precision calculated on the Petri net translations using the alignments technique from [21, 134] and set a time limit of at most 15 minutes.

#### Event Logs

We selected seven event logs for this evaluation, covering a range of input problem sizes. The input problem size is measured in terms of four metrics: number of traces, number of events, number of distinct activities (size of the alphabet), and the (average) trace length. The event logs and their input sizes are shown in Table 6.8 and are divided into software and non-software logs.

For the software event logs we used an extended version of the instrumentation tool developed for [119], yielding XES event logs with method-call level

**Table 6.8:** The event logs used in the performance and scalability evaluation. Shown are input size statistics, indicating the problem sizes of the event logs.

Event Log	# Traces	# Events	# Acts	Trace length		
				Min	Mean	Max
[108] JUnit 4.12	1	946	182	946	946	946
[109] Crypto 1.0.0	3	241,973	74	278	80,658	241,140
[110] NASA CEV	2,566	73,638	47	12	29	50
[112] Alignments	1	17,912	90	17,912	17,912	17,912
[63] BPIC 2012	13,087	262,200	24	3	20	175
[174] BPIC 2013	7,554	65,533	13	1	9	123
[45] WABO	1,434	8,577	27	1	6	25

events. The JUnit 4.12 software [67] was executed once, using the example input found at [40]. For the Apache Commons Crypto 1.0.0 software [29], we executed the `CbcNoPaddingCipherStreamTest` unit test. For the NASA CEV software [148], we executed a unit test generated from the source code, covering all of the code branches. For the alignments software [21, 187], we executed an alignment computation on a typical input log and model.

The *BPIC 2012* [63] and *BPIC 2013* [174] event logs are so-called BPI Challenge logs. These large real-life event logs with complex behavior are often used in process mining evaluations. The challenge logs are made available yearly in conjunction with the BPM conference and are considered sufficiently large and complex inputs to stress test process mining techniques. The *WABO* [45] event log describes the receipt phase of an environmental permit application process (“WABO”) at a Dutch municipality.

For each of the above logs, we use various *heuristics for hierarchy*. For the software logs, we use the nested intervals or nested calls heuristic (Intervals) as well as a structured names heuristic based on the `package.class.method()` pattern (Names). For the *BPIC 2012* log, we use the `<A>_<name>` pattern, splitting activities based on their prefix. For the *BPIC 2013* log, we use a combination of activity name and lifecycle label to form a hierarchy. For the *WABO* log, we use a combination of resource group and activity name to form a hierarchy.

## Results – Running Time

In Tables 6.10 and 6.11, the results for the runtime benchmark are given for the software and non-software logs respectively.

We immediately observe that the ILP, MINT, and Synoptic algorithms could not finish in time on most logs. We observed that the MINT and Synoptic algorithms have difficulty handling a large number of traces, see for example the BPIC and NASA logs. In addition, we also notice that most algorithms require a long processing time and a lot of memory for the Apache Crypto log. We conclude that large trace lengths, such as in the Crypto log, are problematic for most approaches. In contrast, our NHD and RAD techniques overcome this problem by using the hierarchy to divide large traces into multiple smaller traces.

The running time of our techniques depend on an implicit input problem size metric: the depth of the discovered hierarchy. In Table 6.9, the discovered depths are given for comparison. For example, for the JUnit and Crypto logs, we see that our techniques have a much lower running time than the baseline Inductive Miner. This can be explained by the large depth in hierarchy: 25 levels and 8 levels respectively. This implies that the event log is effectively decomposed into many smaller sublogs with reduced activity alphabets. Recall from Theorems 6.4.7 and 6.5.8 that the NHD and RAD algorithms are mainly influenced by the activity alphabet size. Hence, the imposed hierarchy



**Table 6.9:** The depth of the discovered hierarchy for the event logs used in the performance and scalability evaluation.

Event Log	Intervals		Names	
	Naïve	RAD	Naïve	RAD
[108] JUnit 4.12	25	18	9	9
[109] Crypto 1.0.0	8	8	8	8
[110] NASA CEV	3	3	3	3
[112] Alignments	14	7	7	7
[63] BPIC 2012			2	2
[174] BPIC 2013			2	2
[45] WABO			2	2

indirectly yields a good decomposition of the problem, aiding the divide and conquer tactics of the underlying algorithms.

Looking at the actual running times again, in all cases, using the right heuristic for hierarchy before discovery improves the running time as a side-effect of discovering more hierarchical structures. In extreme cases, like the Apache Crypto log, it even makes the difference between getting a result and having no result at all. Note that, with a poorly chosen heuristic, we might not discover any meaningful hierarchical structures. As a consequence, the running time is also negatively affected, e.g., note the absence of models for the Apache Crypto when using the *Names* heuristic.

### Results – Model Quality

In Tables 6.12 and 6.13, the results of the model quality measurements are given. The Fuzzy miner is absent due to the lack of semantics for fuzzy models.

Note that in all software cases, the NHD and RAD algorithms using the *Intervals* hierarchy heuristics yield a big improvement in precision, with no significant impact on fitness. Clearly, using the right heuristic for hierarchy not only impacts the running time but also the model quality. Intuitively, this makes sense: if hierarchical information is (implicitly) present in an event log, using such information yields more accurate models. Note that there is a tradeoff between including recursive behavior and precision. As we already saw in Section 6.7.1, using the RAD approach can generalize and simplify a model at the expense of some precision. Depending on the use case of the model, such a small sacrifice in precision can be a good thing.

Observe that these results verify that our technique with paths at 1.0 maintains the model quality guarantees (perfect fitness). Overall, we can conclude that the added expressiveness of modeling the hierarchy when present has a positive impact on the model quality.

In the non-software cases, when using a poorly chosen heuristic for hierarchy, the NHD and RAD algorithms can yield less fitting and precise models. In these cases, there is a tradeoff between including more information via the artificial hierarchy levels and fitness and precision. That is, there is a tradeoff between the amount of information included in the model, understandability, and fitness and precision.

### Results – Hierarchy in the JUnit case

As an example of the usefulness of hierarchical models on large real-life event logs, we consider the JUnit 4.12 case again. Figure 6.18 shows the discovered model at various abstraction levels. By collapsing and expanding named subtrees, the discovered hierarchy can be used to manage the visual complexity and aiding the user in exploring the discovered behavior. The model in Figure 6.18a can still be examined with a glance. But the model in Figure 6.18b is already a bit more complex, even though only one level of named subtrees have been collapsed.

When more detail is shown, like in Figure 6.18c, it becomes clear that we need to enable the user to use the hierarchy to navigate between the different levels of complexity. In other words, one would not start looking at Figure 6.18c. Instead, one starts with Figure 6.18a, and by interactively expanding and collapsing parts of the model, knows where to zoom in on the model in Figure 6.18c to investigate the details of the shown behavior. Chapter 10 shows how we realized such user interactions.

Overall, we can conclude that the added expressiveness of modeling the hierarchical behavior explicitly has, in most cases, a positive impact on the model. In addition, the interaction with hierarchical notions proves essential for understanding large, complex (software) behavior.

**Table 6.10:** Comparison of algorithm running times on software event logs with hierarchies. Given are the average running times in milliseconds over 30 runs, with a 95% confidence interval shown in the bar plots. Note that the plots use a logarithmic scale. For our hierarchical extensions, the used hierarchy heuristic are shown in parenthesis. The paths column indicates the value for the IMf infrequent threshold: 1.0 means all behavior, 0.8 means 80% of the behavior (i.e., an 80/20 model), 0.5 means 50% of the behavior.

Algorithm	Paths	JUnit 4.12	Crypto 1.0.0	NASA CEV	Alignments	
[18] Alpha miner		9.2	183.1	37.8	15.6	
[192] Heuristics		1349.7	— <sup>T</sup>	359.6	444.4	
[75] Fuzzy miner		166.8	— <sup>T</sup>	4148.2	— <sup>T</sup>	
[193] ILP miner		— <sup>T</sup>	— <sup>T</sup>	— <sup>T</sup>	— <sup>T</sup>	
[204] ILP, filtering		— <sup>T</sup>	— <sup>T</sup>	— <sup>T</sup>	— <sup>T</sup>	
[10] Genetic miner		— <sup>T</sup>	— <sup>T</sup>	— <sup>T</sup>	21945.3	
[46] ETMd miner		— <sup>T</sup>	— <sup>T</sup>	— <sup>T</sup>	— <sup>T</sup>	
[16] TS Regions		— <sup>T</sup>	— <sup>T</sup>	— <sup>T</sup>	— <sup>T</sup>	
[190] MINT, redblue	k=1	243.9	— <sup>T</sup>	13426.0	— <sup>S</sup>	
[190] MINT, redblue	k=2	582.0	— <sup>T</sup>	22213.4	— <sup>S</sup>	
[190] MINT, redblue	k=3	751.8	— <sup>T</sup>	— <sup>T</sup>	— <sup>S</sup>	
[190] MINT, ktails	k=1	108.9	— <sup>T</sup>	— <sup>T</sup>	— <sup>S</sup>	
[190] MINT, ktails	k=2	371.6	— <sup>T</sup>	— <sup>T</sup>	— <sup>S</sup>	
[190] MINT, ktails	k=3	512.3	— <sup>T</sup>	— <sup>T</sup>	— <sup>S</sup>	
[37] Synoptic		— <sup>T</sup>	— <sup>T</sup>	— <sup>T</sup>	— <sup>T</sup>	
[130] IM (baseline)	1.0	232.1	12055.5	911.3	4560.6	
[130] IM (baseline)	0.8	291.9	9998.9	912.6	3995.8	
[130] IM (baseline)	0.5	276.3	10237.2	676.2	2602.7	
Section 6.4	NHD (Intervals)	1.0	11.6	1519.8	382.7	453.7
	NHD (Intervals)	0.8	8.8	1537.0	310.7	420.1
	NHD (Intervals)	0.5	8.4	1525.2	368.4	177.6
	NHD (Names)	1.0	27.0	11876.3	1352.3	207.9
	NHD (Names)	0.8	25.6	6315.9	1375.6	191.0
	NHD (Names)	0.5	24.9	6308.4	1377.6	192.2
Section 6.5	RAD (Intervals)	1.0	11.1	1757.3	418.1	2452.0
	RAD (Intervals)	0.8	10.4	2202.8	458.1	2642.8
	RAD (Intervals)	0.5	9.4	1968.6	418.4	2359.1
	RAD (Names)	1.0	27.3	12414.6	1548.2	190.2
	RAD (Names)	0.8	26.5	6340.2	1488.6	186.9
	RAD (Names)	0.5	24.6	6350.2	1588.6	214.7

Avg. runtime (in milliseconds) with 95% conf. int.

<sup>S</sup> Stack overflow

<sup>T</sup> Time limit exceeded (30 sec.)

**Table 6.11:** Comparison of algorithm running times on non-software event logs with hierarchies. Given are the average running times in milliseconds over 30 runs, with a 95% confidence interval shown in the bar plots. Note that the plots use a logarithmic scale. For our hierarchical extensions, the used hierarchy heuristic are shown in parenthesis. The paths column indicates the value for the IMf infrequent threshold: 1.0 means all behavior, 0.8 means 80% of the behavior (i.e., an 80/20 model), 0.5 means 50% of the behavior.

Algorithm	Paths	BPIC 2012	BPIC 2013	WABO	
[18] Alpha miner		150.1	73.5	8.8	
[192] Heuristics		840.2	278.0	49.2	
[75] Fuzzy miner		2858.5	827.4	159.9	
[193] ILP miner		– <sup>T</sup>	3259.9	233.9	
[204] ILP, filtering		7234.3	1154.7	236.3	
[10] Genetic miner		– <sup>T</sup>	22145.4	26029.7	
[46] ETMd miner		– <sup>T</sup>	– <sup>T</sup>	– <sup>T</sup>	
[16] TS Regions		– <sup>T</sup>	1572.9	– <sup>T</sup>	
[190] MINT, redblue	k=1	– <sup>T</sup>	– <sup>T</sup>	302.1	
[190] MINT, redblue	k=2	– <sup>T</sup>	– <sup>T</sup>	276.8	
[190] MINT, redblue	k=3	– <sup>T</sup>	– <sup>T</sup>	325.0	
[190] MINT, ktails	k=1	– <sup>T</sup>	– <sup>T</sup>	76.4	
[190] MINT, ktails	k=2	– <sup>T</sup>	– <sup>T</sup>	138.2	
[190] MINT, ktails	k=3	– <sup>T</sup>	– <sup>T</sup>	160.1	
[37] Synoptic		– <sup>T</sup>	– <sup>T</sup>	– <sup>T</sup>	
[130] IM (baseline)	1.0	4083.8	1269.5	113.9	
[130] IM (baseline)	0.8	3436.7	477.8	62.1	
[130] IM (baseline)	0.5	3969.9	999.7	55.2	
Sec. 6.4	NHD (Names)	1.0	4172.3	1098.5	166.1
	NHD (Names)	0.8	2128.6	797.1	143.5
	NHD (Names)	0.5	1912.5	841.3	111.4
Sec. 6.5	RAD (Names)	1.0	4996.0	1460.8	206.8
	RAD (Names)	0.8	2528.7	880.6	148.2
	RAD (Names)	0.5	2178.5	930.1	112.1

Avg. runtime (in milliseconds) with 95% conf. int.

<sup>T</sup> Time limit exceeded (30 sec.)



**Table 6.12:** Comparison of model quality scores on software event logs with hierarchies. Given are the fitness and precision values for the discovered models. These values range from 0.0 to 1.0, higher is better.

Algorithm	Paths	JUnit 4.12		Crypto 1.0.0		NASA CEV		Alignments		
		Fitness	Precision	Fitness	Precision	Fitness	Precision	Fitness	Precision	
[18] Alpha miner		$\_U$	$\_U$	$\_U$	$\_U$	0.91	0.06	1.00	0.01	
[192] Heuristics		$\_U$	$\_U$	n/a	n/a	$\_U$	$\_U$	$\_U$	$\_U$	
[193] ILP miner		n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	
[204] ILP, filtering		n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	
[10] Genetic miner		n/a	n/a	n/a	n/a	n/a	n/a	$\_U$	$\_U$	
[46] ETMd miner		n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	
[16] TS Regions		n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	
[190] MINT, redblue	k=1	0.00	$\_R$	n/a	n/a	0.79	0.44	n/a	n/a	
[190] MINT, redblue	k=2	0.48	0.17	n/a	n/a	0.81	0.45	n/a	n/a	
[190] MINT, redblue	k=3	0.13	0.06	n/a	n/a	n/a	n/a	n/a	n/a	
[190] MINT, ktails	k=1	0.00	$\_R$	n/a	n/a	n/a	n/a	n/a	n/a	
[190] MINT, ktails	k=2	0.43	0.16	n/a	n/a	n/a	n/a	n/a	n/a	
[190] MINT, ktails	k=3	0.12	0.06	n/a	n/a	n/a	n/a	n/a	n/a	
[37] Synoptic		n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	
[130] IM (baseline)	1.0	1.00	0.33	1.00	0.35	1.00	0.55	$\_T$	$\_T$	
[130] IM (baseline)	0.8	0.96	0.32	0.88	0.41	0.91	0.53	$\_T$	$\_T$	
[130] IM (baseline)	0.5	0.98	0.33	0.95	0.38	0.93	0.62	$\_T$	$\_T$	
Section 6.4	NHD (Intervals)	1.0	1.00	0.84	1.00	0.45	1.00	0.80	$\_T$	$\_T$
	NHD (Intervals)	0.8	0.90	0.87	0.99	0.45	1.00	0.81	$\_T$	$\_T$
	NHD (Intervals)	0.5	0.88	0.87	0.95	0.40	0.94	0.82	$\_T$	$\_T$
	NHD (Names)	1.0	$\_M$	$\_M$	$\_M$	$\_M$	$\_T$	$\_T$	$\_T$	$\_T$
	NHD (Names)	0.8	$\_M$	$\_M$	$\_M$	$\_M$	$\_T$	$\_T$	$\_T$	$\_T$
	NHD (Names)	0.5	$\_M$	$\_M$	$\_M$	$\_M$	$\_T$	$\_T$	$\_T$	$\_T$
Section 6.5	RAD (Intervals)	1.0	1.00	0.83	1.00	0.45	1.00	0.80	$\_T$	$\_T$
	RAD (Intervals)	0.8	0.89	0.84	0.99	0.45	1.00	0.81	$\_T$	$\_T$
	RAD (Intervals)	0.5	0.86	0.85	0.95	0.40	0.94	0.82	$\_T$	$\_T$
	RAD (Names)	1.0	$\_M$	$\_M$	$\_M$	$\_M$	$\_T$	$\_T$	$\_T$	$\_T$
	RAD (Names)	0.8	$\_M$	$\_M$	$\_M$	$\_M$	$\_T$	$\_T$	$\_T$	$\_T$
	RAD (Names)	0.5	$\_M$	$\_M$	$\_M$	$\_M$	$\_T$	$\_T$	$\_T$	$\_T$

<sup>T</sup> Time limit exceeded (15 min.)

<sup>M</sup> Out of memory (12 GB)

<sup>R</sup> Not reliable (fitness = 0)

<sup>U</sup> Unsound model

n/a No model (see Table 6.12)

**Table 6.13:** Comparison of algorithm running times on non-software event logs with hierarchies. Given are the fitness and precision values for the discovered models. These values range from 0.0 to 1.0, higher is better.

Algorithm	Paths	BPIC 2012		BPIC 2013		WABO	
		Fitness	Precision	Fitness	Precision	Fitness	Precision
[18] Alpha miner		–U	–U	0.36 █	0.88 █	–U	–U
[192] Heuristics		0.72 █	0.95 █	–U	–U	0.61 █	0.98 █
[193] ILP miner		n/a	n/a	1.00 █	0.36 █	1.00 █	0.12 █
[204] ILP, filtering		0.74 █	0.28 █	0.95 █	0.45 █	0.97 █	0.35 █
[10] Genetic miner		n/a	n/a	–U	–U	–U	–U
[46] ETMd miner		n/a	n/a	n/a	n/a	n/a	n/a
[16] TS Regions		n/a	n/a	0.56 █	0.96 █	n/a	n/a
[190] MINT, redblue	k=1	n/a	n/a	n/a	n/a	0.73 █	0.55 █
[190] MINT, redblue	k=2	n/a	n/a	n/a	n/a	0.67 █	0.55 █
[190] MINT, redblue	k=3	n/a	n/a	n/a	n/a	0.67 █	0.56 █
[190] MINT, ktails	k=1	n/a	n/a	n/a	n/a	0.00	–R
[190] MINT, ktails	k=2	n/a	n/a	n/a	n/a	0.00	–R
[190] MINT, ktails	k=3	n/a	n/a	n/a	n/a	0.00	–R
[37] Synoptic		n/a	n/a	n/a	n/a	n/a	n/a
[130] IM (baseline)	1.0	1.00 █	0.37 █	1.00 █	0.62 █	1.00 █	0.62 █
[130] IM (baseline)	0.8	0.98 █	0.49 █	0.95 █	0.64 █	0.96 █	0.73 █
[130] IM (baseline)	0.5	0.84 █	0.54 █	0.35 █	0.82 █	0.96 █	0.77 █
Sec. 6.4	NHD (Names)	1.0	–T	–T	–T	–T	–T
	NHD (Names)	0.8	–T	–T	–T	–T	–T
	NHD (Names)	0.5	–T	–T	–T	–T	–T
Sec. 6.5	RAD (Names)	1.0	–T	–T	–T	–T	–T
	RAD (Names)	0.8	–T	–T	–T	–T	–T
	RAD (Names)	0.5	–T	–T	–T	–T	–T

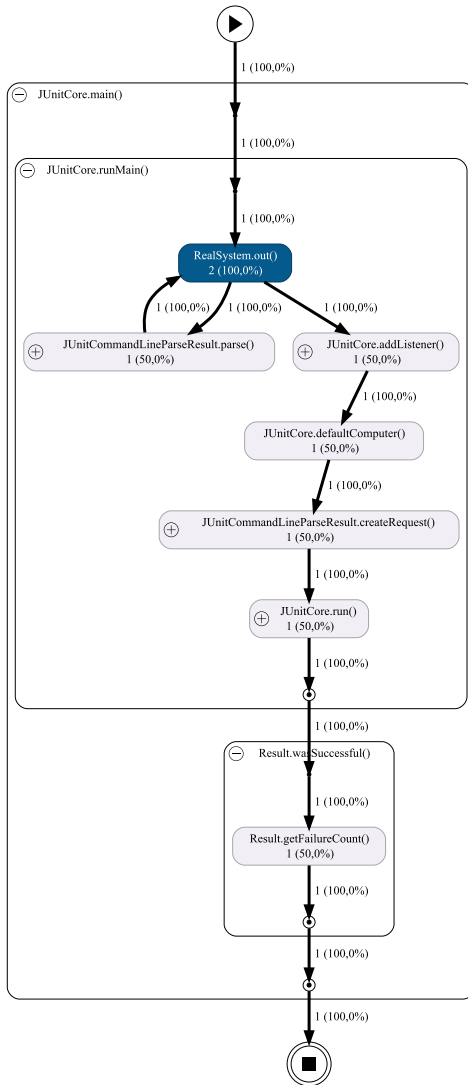
<sup>T</sup> Time limit exceeded (15 min.)

<sup>R</sup> Not reliable (fitness = 0)

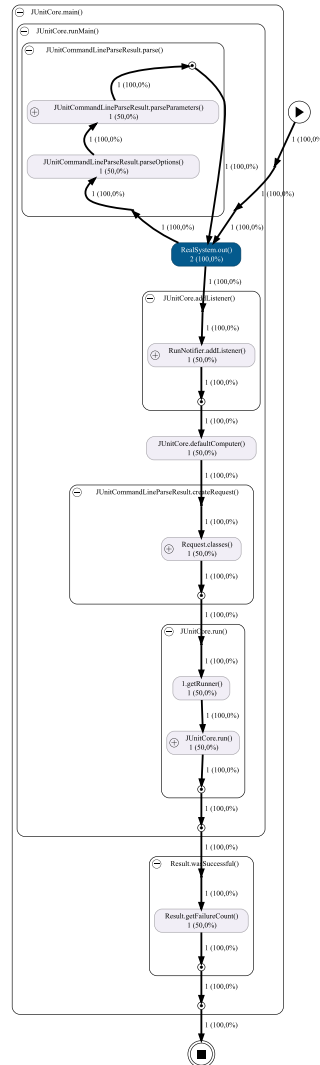
<sup>U</sup> Unsound model

n/a No model (see Table 6.12)

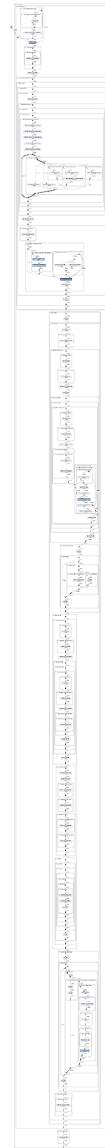




(a) 3 levels



(b) 4 levels



(c) 13 levels

**Figure 6.18:** The NHD result for the JUnit 4.12 software event log, visualized as statecharts with different number of levels in the hierarchy visualized by collapsing lower-level named subtrees into activities (note the + and - signs in Figure 6.18a).

## 6.8 Conclusion and Open Challenges

In this chapter, we introduced a modeling notation and two discovery techniques for hierarchical and recursive behavior (Contribution 2). With *hierarchical event logs*, we can explicitly capture hierarchical behavior found in, for example, software call stack behavior, relations between software components or application interfaces, behavior described by high-level and low-level activities, etc. We presented several heuristics for transforming an ordinary event log into a hierarchical event log. With the *hierarchical process tree* we provided extensions to capture named submodels and recursive behavior. The *naïve hierarchical discovery* and *recursion aware discovery* algorithms discover the named submodels and recursive references from hierarchical event logs and capture this behavior in hierarchical process trees. These discovery algorithms allow us to analyze software processes and other processes at multiple levels of granularity while offering good discovery guarantees. Moreover, these algorithms scale well and show a huge potential to speed up discovery by leveraging hierarchical information.

With the hierarchical solutions presented in this chapter, there are several interesting research directions for future work.

■ **Future Work 6.1 — Communication between Submodels.** The definition of named subtrees presented in this chapter captures block-structured and local named submodels. Everything that happens inside or during a named submodel stays inside that named submodel, and there is no way to influence the state of an external named submodel. However, in practice, one may wish to influence the state of another non-nested submodel. Such communication between submodels is already supported in notations like BPMN and Statecharts, and may be used for various types of behavior. For example, in multi-threaded or multi-process software, concurrent threads or processes (named submodels) may communicate via data structures or (network) channels. Typical examples include produce-consumer setups and network communication. Also in business processes such patterns occur. Consider for example different organization units or artifacts, represented by named submodels, synchronizing on particular states or phases.

■ **Future Work 6.2 — Instance-Aware Recursion Discovery.** In the current Recursion Aware Discovery, the recursion detection algorithm only looks at the list of symbols in the Context Path. However, consider the case where we have two instances of a `class A` and in `A.f()`, instance 1 invokes `A.f()` for instance 2 of `class A`. One can debate if, in such cases, a recursion should be detected. In other words, should the object instances be taken into account when detecting recursion patterns? The current algorithm does not take any notion of instances into account and future work might explore this concept further.



■ **Future Work 6.3 — Multi-Threaded/Multi-Process Fork and Join Identification.** The algorithms presented in this chapter work well for single-threaded software, but have difficulties correctly modeling multi-threaded software. Although concurrent behavior is supported via the concurrency operator ( $\wedge$ ), in practice the algorithms have difficulty deducing where concurrency operators should be used in the modeled behavior. More concretely, the concurrency operator indirectly points out fork and join points where behavior can happen and stop happening concurrently. The current algorithms identify these points by looking at strongly connected components in the directly-follows graph. Since software behavior tends to consist of many small steps, observing enough possible interleavings to correctly form these concurrency footprints in the directly-follows graph is highly unlikely. However, often there is enough non-causal information in running software, like thread and process identification, to suggest where behavior happens concurrently. Thus, using additional information for detecting and modeling concurrency should allow us to correctly identify fork and join points for multi-threaded and multi-process software.

■ **Future Work 6.4 — Multi-Instance Activities and Named Submodels.** A frequently used pattern in multi-threaded software is to invoke a function or subroutine in parallel for each element in a dataset. Likewise, in business processes, a subprocess is dispatched for each element in an order or delivery. Such multi-instantiation of activities or named submodels cannot be adequately modeled using the tradition loop or concurrency constructs. Multiple instances are executed concurrently and any number of instances may be executed at the same time. Like with the nested intervals heuristic, it should be possible to detect and discover multi-instance patterns based on how intervals of activities overlap. Special care should be taken when constructing subtrace hierarchies such that subtraces are assigned to the correct events/submodels.

■ **Future Work 6.5 — Mixed, Multi-Dimensional Hierarchies.** The above approach to modeling and discovering hierarchies is one dimensional. In practice, one can adopt multiple different hierarchical views. For example, software behavior can contain a call hierarchy, a thread hierarchy, and a class or component hierarchy. When analyzing such behavior, it could be useful and insightful to look at a dissection of multiple hierarchies. For example, how are method calls related (call hierarchy) across the different classes or components. One possible approach for discovering such multi-dimensional hierarchical models is to choose one main dimension (e.g., a call hierarchy), and to discover annotations or “colorings” for the other hierarchies (e.g., associated classes or components). These annotations can be used after discovery to visually indicate how the hierarchies mix, and in which component which part of a submodel belongs. Possible visualization styles include color mappings and spatial layout regions like those used by message sequence diagrams.

■ **Future Work 6.6 — Hierarchy-Aware Alignments.** The algorithms presented in this chapter scale well for hierarchical event logs and are able to produce larger models. These models are highly structured, with clear boundaries for submodels. However, when performing alignments for conformance and performance analysis, one has to flatten these models to regular Petri nets, losing this structured information. As shown in Section 6.7.2, these flat Petri nets are easily too large to be handled by the traditional alignments algorithm. Future research should look into leveraging the structured information available in hierarchical models to improve the scalability of alignments. For example, the submodels in a hierarchical model can be used to apply a divide-and-conquer strategy to alignments similar to the discovery approaches in this chapter. Since the main bottleneck in alignments is the size of the model state space and trace length, applying a divide and conquer on these dimensions should significantly reduce the problem sizes. Hence, we conjecture that a hierarchy-aware alignments approach has a huge potential to speed up computations by leveraging hierarchical information.

■ **Future Work 6.7 — Hierarchical Visualization Layout.** The layered nature of the hierarchical models used in this chapter allow the user to interactively explore behavior across different abstraction layers. Named submodels can be used to abstract from detailed behavior, and users should be able to interactively inspect, collapse and expand named submodels. Chapter 10 shows how we realized such user interactions. Due to the structured, hierarchical nature of hierarchical process trees, there is a lot of information available on how to layout such models. Hence, a visualization layout algorithm can use this information to aid the user, ensuring that the same model elements are displayed in the same fashion after a named submodel is expanded or collapsed. Such a layout algorithm should use the tree structure to clearly and deterministically position all model elements in a robust way.

■ **Future Work 6.8 — On-demand Discovery and Software Reruns.** The layered nature of the hierarchical models used in this chapter also lends itself well to explore the dataset in multiple phases. At first, a user probably wants to analyze a high-level picture of the behavior, excluding any obfuscating detailed behavior. Afterwards, a user would probably be interested in a certain path involving certain activities (e.g., software classes or methods) or properties (e.g., certain performance characteristics). At the same time, a user would ignore all the other paths. Hence, we can utilize the *delayed discovery* principle discussed in Section 6.5 to avoid unnecessary discovery computations. In addition, with some integration with the underlying software being analyzed, we could capture new behavioral data on demand by rerunning the software whenever a user wishes to dig deeper in certain classes/methods, i.e., activities and named submodels.



*“It’s easy to believe in something when you win all the time. The losses are what define a man’s faith.”*

— Brandon Sanderson, *The Well of Ascension*

## 7 | Cancellation Discovery

In this chapter, we introduce a modeling notation and process discovery technique for cancellation behavior (Contribution 3). We start by motivating the need for cancellation in Section 7.1. After that, we introduce the *cancellation process tree* notation (Section 7.2) and the idea of a *trigger oracle* (Section 7.3). Based on these concepts, Section 7.4 introduces the *cancellation discovery* algorithm. In addition, Section 7.5 will elaborate on extending the ideas to existing discovery extensions and to non-atomic event logs. Finally, Section 7.6 will evaluate the introduced algorithms on rediscoverability and performance. Section 7.7 wraps up this chapter and discusses open challenges.

### 7.1 Why We Need Cancellation – The Exceptional Case

When applying process mining on event data originating from software systems, new patterns and challenges pop up. Typically, the run-time behavior of a software system is large, complex, and also contains some form of cancellation or error-handling behavior. For example, a web server needs to handle a connection error, an X-ray machine may detect a sensor problem, a library may encounter configuration, input, or parse errors, etc. Also in the context of business processes, various errors and cancellation patterns can occur. For example, a bank loan request may be canceled or declined, a pending road fine reminder may be canceled upon payment, etc. This type of cancellation behavior can easily be expressed in existing modeling formalisms, but few existing process discovery techniques actually take these cancellation features into account. Without cancellation support, discovery algorithms can produce needlessly complex and imprecise models, as shown in Section 3.2 and Challenge 4.

As an example of cancellation behavior, consider the program in Listing 7.1. The program starts its `Main.main()` function with invoking `input()`. Based on the outcome of `input()`, either `processA()` or `processB()` is invoked. These process functions can either succeed and return or fail by throwing an exception. If the process method succeeds, the `prepareResult()` function is invoked. If the process method throws an exception, we break out of the normal control flow (i.e., a cancellation is performed), and continue with the `recover()` function in

---

**Listing 7.1** Running example Java code illustrating cancellation behavior. Upon execution, this program is logged at the method level and the catch block.

---

```
1 public class Main {
2     // Entry point
3     public static void main(int i) {
4         boolean check = input(i);
5         // the try-catch block hints at a cancellation region
6         try {
7             if (check) {
8                 processA(i); // can throw an exception
9             } else {
10                processB(i); // can throw an exception
11            }
12            prepareResult(); // only executed when no exception was thrown
13        } catch (Exception e) {
14            recover(); // only executed when an exception was thrown
15        }
16        output(); // always executed
17    }
18    private static boolean input(int i) { ... }
19    private static void processA(int i) throws Exception { ... }
20    private static void processB(int i) throws Exception { ... }
21    private static void prepareResult() { ... }
22    private static void recover() { ... }
23    private static void output() { ... }
24 }
```

---

the catch block. In both the success and fail (exception) case, afterwards we finish with the `output()` function.

Observe how, in the above example, the jump to catch upon an exception yields cancellation behavior. In this case, the try-catch block already hinted at this pattern, but this is not always the case. For example, in (embedded) C programs, a pattern like in Listing 7.2 can yield the same behavior. In other cases, there may be no direct and obvious hints in the source code. Consider, for example, null-pointers and divide-by-zero problems (i.e., runtime exceptions) or failure events and callback mechanisms used in APIs and component interfaces.

Consider the program in Listing 7.1 again. Suppose we execute this program a few times, where the first execution yields `check = true` and no exception is thrown, but the second execution yields `check = false` and throws an exception. And suppose we trace and log the start and end of each called method as well as the start of the catch block. Table 7.1 shows the resulting event log describing the first two cases. In case 1, no exception is thrown, showing “good weather behavior”. Due to the exception thrown in case 2, the function

**Listing 7.2** C version of the example program in Listing 7.1.

---

```

1  int input(int argc, const char* argv[]);
2  int processA(int argc, const char* argv[]);
3  int processB(int argc, const char* argv[]);
4  int prepareResult();
5  void recover();
6  void output();
7  // Entry point
8  int main(int argc, const char* argv[]) {
9      int check, result; // the result variable is used for error-handling
10     check = input(argc, argv);
11     if (check != 0) {
12         result = processA(argc, argv); // returns -1 upon error
13     } else {
14         result = processB(argc, argv); // returns -1 upon error
15     }
16     if (result != -1) {
17         result = prepareResult(); // only executed when no error was encountered
18     }
19     if (result == -1) {
20         recover(); // only executed when an error was encountered
21     }
22     output(); // always executed
23 }

```

---

`processB()` does not return but cancels. As a result, we see that event 2.4 is recorded as a *throws*, and event 2.5 signals an exception is being *caught*. Note that, in this way, we can deduce where a cancellation pattern is being triggered and handled, but it is not immediately obvious where the corresponding cancellation region starts.

In the next section, we will show how cancellation behavior can be modeled and made explicit in event logs using an oracle. After that, we will discuss the cancellation discovery algorithm and the challenge of finding the right cancellation region.

## 7.2 Cancellation Process Trees

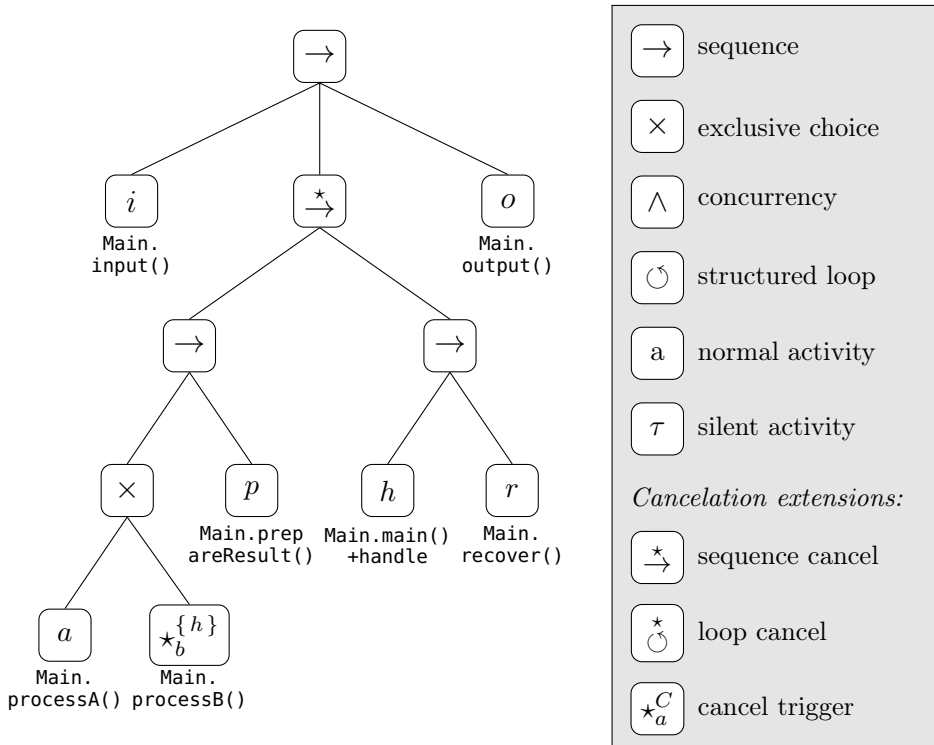
In this section, we extend the *process tree* notation (Section 2.2.6, page 38), with support for cancellation in the form of cancellation regions and cancellation triggers. In our extended notation, called the *cancellation process tree*, we add two new tree operators  $\overset{\star}{\rightarrow}$  and  $\overset{\star}{\circlearrowleft}$  to represent the *sequence* and *loop cancellation regions* respectively, and we add a new tree leaf  $\star_a^C$  to denote a *cancellation trigger*. We will first explain the concept using an example model, and afterwards we formally present the new syntax and semantics.

**Table 7.1:** Example snippet of an event log for the program in Listing 7.1. Each row is an event, and each column an attribute. Most events relate to the start and end of each called method. The event labelled  $h$  relates to the start of the catch block.

Case id	Event id	Attributes					
		Activity	Lifecycle	Timestamp	Resource	...	
1	1.1	$i$	Main.input()	start	30-10-2017 11:02:45.000	main-thread	...
	1.2	$i$	Main.input()	complete	30-10-2017 11:02:45.200	main-thread	...
	1.3	$a$	Main.processA()	start	30-10-2017 11:02:45.400	main-thread	...
	1.4	$a$	Main.processA()	complete	30-10-2017 11:02:45.650	main-thread	...
	1.5	$p$	Main.prepareResult()	start	30-10-2017 11:02:45.690	main-thread	...
	1.6	$p$	Main.prepareResult()	complete	30-10-2017 11:02:45.730	main-thread	...
	1.7	$o$	Main.output()	start	30-10-2017 11:02:45.770	main-thread	...
	1.8	$o$	Main.output()	complete	30-10-2017 11:02:45.840	main-thread	...
2	2.1	$i$	Main.input()	start	30-10-2017 11:02:48.150	main-thread	...
	2.2	$i$	Main.input()	complete	30-10-2017 11:02:48.470	main-thread	...
	2.3	$b$	Main.processB()	start	30-10-2017 11:02:48.600	main-thread	...
	2.4	$b$	Main.processB()	throws	30-10-2017 11:02:48.650	main-thread	...
	2.5	$h$	Main.main()+handle	handle	30-10-2017 11:02:48.690	main-thread	...
	2.6	$r$	Main.recover()	start	30-10-2017 11:02:48.720	main-thread	...
	2.7	$r$	Main.recover()	complete	30-10-2017 11:02:48.890	main-thread	...
	2.8	$o$	Main.output()	start	30-10-2017 11:02:48.960	main-thread	...
	2.9	$o$	Main.output()	complete	30-10-2017 11:02:49.210	main-thread	...
3	3.1	$i$	Main.input()	start	30-10-2017 11:02:56.700	main-thread	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

### 7.2.1 Example Model of a Program Execution

We will explain the cancellation extensions using the example cancellation tree in Figure 7.1. This tree is modeled based on the event log in Table 7.1, see also the program in Listing 7.1. The root node is a sequence ( $\rightarrow$ ) of `input()`, a *sequence cancellation operator* ( $\overset{\star}{\rightarrow}$ ), and finally `output()`. The sequence cancellation node has two subtrees. The left subtree models the normal behavior, i.e., either `processA()` or `processB()` is performed, followed by `prepareResults()`. At the same time this left subtree also denotes what is in the cancellation region. Observe that `processB()` is modeled as a cancellation trigger leaf ( $\star_b^{\{h\}}$ ), meaning that either `processB()` is either executed successfully, or the corresponding cancellation region is triggered. We will explain the meaning of the superscript  $\{h\}$  later in the formal semantics. The right subtree models what happens once the cancellation region is triggered. That is, when the cancellation region is triggered, we catch the exception (`main()+handle`), and execute `recover()`. Afterwards, we continue with `output()` as normal. Observe that, as a result, we do not always execute the entire left subtree. That is, in the presence of cancellation operators, we consider the prefix language of the left subtree.



**Figure 7.1:** Example cancellation process tree for the event log in Table 7.1. This tree models concrete runtime behavior for the program in Listing 7.1.

In the tree in Figure 7.1, we used the sequence cancellation operator  $\overset{*}{\rightarrow}$ . As a result, after the right subtree is executed due to a cancellation, we continue with `output()`. If instead we would replace the operator  $\overset{*}{\rightarrow}$  with the loop cancellation operator  $\overset{*}{\circlearrowleft}$ , we get looping behavior. In such a tree, after the right subtree is executed, we loop back and try executing the left subtree again. Hence, with loop cancellation, we repeatedly retry the left subtree until no cancellation is triggered.

The cancellation process tree in Figure 7.1 can also be represented textually:

$$\rightarrow(i, \overset{*}{\rightarrow}(\rightarrow(\times(a, \overset{*}{\star}_b^{\{h\}}), p), \rightarrow(h, r)), o)$$

### 7.2.2 Syntax and Semantics

To formalize cancellation process trees, we introduce the following syntax and semantics.

**Definition 7.2.1 — Cancellation Process Tree.** We formally define *cancellation process trees* recursively. We assume a finite alphabet  $\mathbb{A}$  of activities with



$\star \notin \mathbb{A}$  and a set  $\otimes$  of operators to be given.

We define the following base cases for cancellation process trees:

$a$  any  $a \in (\mathbb{A} \cup \{\tau\})$  is a (silent) activity leaf

$\star_a^C$  **Cancellation extension:** denotes a *cancellation trigger* for activity  $a \in \mathbb{A}$  and the set of corresponding triggers  $C \subseteq \mathbb{A}$

Let  $Q_1, \dots, Q_n$  with  $n > 0$  be cancellation process trees and let  $\otimes \in \otimes$  be a cancellation process tree operator, then  $\otimes(Q_1, \dots, Q_n)$  is a cancellation process tree. We consider the following cancellation process tree operators:

$\rightarrow$  denotes a *sequence* or the sequential composition of all subtrees

$\times$  denotes an *exclusive choice* or XOR choice between one of the subtrees

$\wedge$  denotes *concurrency* or the parallel composition of all subtrees

$\circlearrowleft$  denotes the *structured loop* or redo loop with loop body  $Q_1$  and alternative loop back paths  $Q_2, \dots, Q_n$  (with  $n \geq 2$ )

$\star \rightarrow$  **Cancellation extension:** denotes the *sequence cancellation* with cancellation body  $Q_1$  and mutually exclusive cancellation alternative paths  $Q_2, \dots, Q_n$

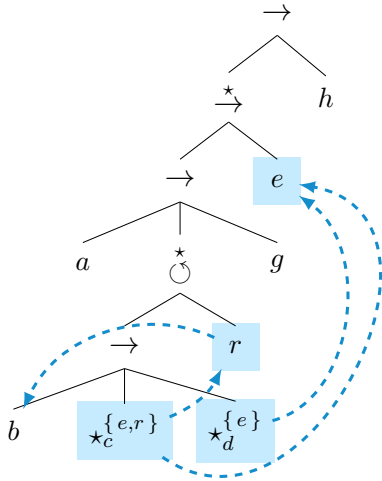
$\star \circlearrowleft$  **Cancellation extension:** denotes the *loop cancellation* with cancellation body  $Q_1$  and mutually exclusive cancellation loop-back paths  $Q_2, \dots, Q_n$

The intuition behind the cancellation operators is as follows. We can put any subtree in a cancellation region by assigning it as the first child of a sequence  $\star \rightarrow$  or loop  $\star \circlearrowleft$  cancellation operator. The non-first subtrees of a cancellation operator denote the possible cancellation paths after the cancellation region was triggered. The start activities of these non-first subtrees represent the *triggers* for the cancellation region. At any place in the first child of a cancellation operator, a cancellation trigger leaf  $\star_a^C$  can be used to link to specific cancellation region triggers  $c \in C$ .

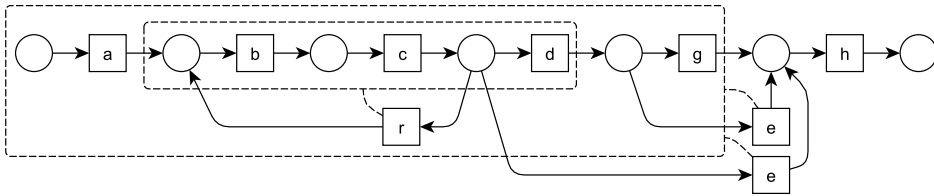
For example, in the tree  $Q_2 = \times(\rightarrow(b, c), \rightarrow(d, e))$ , the set of start activities is  $Start(Q_2) = \{b, d\}$ . In the tree  $\star \rightarrow(\star_a^{\{b\}}, Q_2)$ , we can trigger this cancellation region at activity  $a$  since  $\{b\} \cap Start(Q_2) \neq \emptyset$ . However, in the tree  $\star \rightarrow(\star_a^{\{g\}}, Q_2)$ , we *cannot* trigger this cancellation region at activity  $a$  since  $\{g\} \cap Start(Q_2) = \emptyset$ , i.e., there are no matching, corresponding triggers.

As another example, consider Table 7.2. The leaf  $\star_c^{\{e, r\}}$  can either execute  $c$  as normal or trigger either the cancellation region starting with  $e$  or  $r$ . When a cancellation region is triggered, activities  $d$  and  $g$  are skipped. If  $e$  was triggered, we continue with activity  $h$  after  $e$  was performed. If  $r$  was triggered, we return to the start activity  $b$  after  $r$  was performed. The leaf  $\star_d^{\{e\}}$  can either execute  $d$  as normal or trigger the cancellation region starting with  $e$ . Observe that, at this leaf, we cannot trigger the cancellation region starting with  $r$ .

**Table 7.2:** Example *cancellation process tree* on the left with its language on the right. Shown are the traces in the language and the corresponding triggers that are used to generate the trace. The grey arrows in the cancellation process tree indicate the possible cancellation trigger “jumps”. At the bottom, the corresponding Reset WF net (see Section 2.2.1, page 31) is shown.



Triggers	Trace
–	$\langle a, b, c, d, g, h \rangle$
$e$	$\langle a, b, c, e, h \rangle$
$e$	$\langle a, b, c, d, e, h \rangle$
$r$	$\langle a, b, c, r, b, c, d, g, h \rangle$
$r, e$	$\langle a, b, c, r, b, c, e, h \rangle$
$r, e$	$\langle a, b, c, r, b, c, d, e, h \rangle$
$r, r$	$\langle a, b, c, r, b, c, r, b, c, d, g, h \rangle$
$r, r, e$	$\langle a, b, c, r, b, c, r, b, c, e, h \rangle$
$r, r, e$	$\langle a, b, c, r, b, c, r, b, c, d, e, h \rangle$
⋮	



The semantics of cancellation process trees are formally defined by extending the language function  $\mathcal{L}(Q)$  (see also Definition 2.2.12 on page 40).

**Definition 7.2.2 — Cancellation Process Tree Semantics and Language.** In this definition, we will be using the notations from Definition 2.3.4 on page 47 and the semantics previously defined in Definition 2.2.12. Let  $\mathbb{A}\star = \mathbb{A} \cup \{\star_a^C \mid a \in \mathbb{A} \wedge C \subseteq \mathbb{A}\}$ . Below, we define a language of the type  $\mathcal{B}((\mathbb{A}\star)^*)$ , i.e., a set of traces, where each event is an activity or error trigger.

First, we define the language of the *cancellation trigger*  $\star_a^C$  leaf. At this leaf, we can either execute activity  $a \in \mathbb{A}$  as normal, or execute  $a$  and trigger an cancellation region starting with  $c \in C$ . In case of the cancellation trigger, we simply “mark” this leaf in the language and resolve this *trigger*

*marker* at the first matching cancellation region  $\xrightarrow{\star}$  or  $\circlearrowleft^{\star}$ . We define:

$$\mathcal{L}(\star_a^C) = \{ \langle a \rangle, \langle \star_a^C \rangle \} \text{ for } a \in \mathbb{A} \wedge C \subseteq \mathbb{A}$$

Next, consider the cancellation region operators  $\xrightarrow{\star}$  and  $\circlearrowleft^{\star}$ . For both operators there are two common cases we can handle in the same way:

**Case 1** No cancellation region is triggered

**Case 2** A cancellation region is triggered, but not matched by this operator. The trigger marker is propagated up the tree.

We define the language  $\otimes_{\mathcal{L}}^{\star}$  to represent these common cases below. We will use the function  $\Phi_{\mathcal{L}}(L) = L'$  to obtain the prefix of a language such that any activities after a trigger marker  $\star_a^C$  is removed.

$$\begin{aligned} \otimes_{\mathcal{L}}^{\star}(L_1, \dots, L_n) &= \{ \sigma_1 \mid \sigma_1 \in L_1 \wedge \star_a^C \notin \sigma_1 \} && \text{Case 1} \\ &\cup \{ \sigma_1 \cdot \langle \star_a^{C \setminus S} \rangle \mid \sigma_1 \cdot \langle \star_a^C \rangle \in \Phi_{\mathcal{L}}(L_1) \} && \text{Case 2} \\ &\wedge S = \{ \text{head}(t) \mid t \in \bigcup_{2 \leq j \leq n} L_j \} \\ &\wedge C \setminus S \neq \emptyset \end{aligned}$$

The function  $\Phi_{\mathcal{L}}(L)$  scans all traces, and returns the prefix upon encountering a trigger marker  $\star_a^C$ . For example,  $\Phi_{\mathcal{L}}(\{ \langle a, b, c \rangle \}) = \{ \langle a, b, c \rangle \}$  but  $\Phi_{\mathcal{L}}(\{ \langle a, \star_b^C, c \rangle \}) = \{ \langle a, \star_b^C \rangle \}$ . We define:

$$\begin{aligned} \Phi_{\mathcal{L}}(L) &= \{ \Phi_{\mathcal{L}}(\sigma) \mid \sigma \in L \} && \text{for } L \subseteq \mathbb{A}^* \\ \Phi_{\mathcal{L}}(\langle a \rangle \cdot \sigma') &= \langle a \rangle \cdot \Phi_{\mathcal{L}}(\sigma') && \text{for } a \in \mathbb{A} \\ \Phi_{\mathcal{L}}(\langle \star_a^C \rangle \cdot \sigma') &= \langle \star_a^C \rangle && \text{for } a \in \mathbb{A} \wedge C \subseteq \mathbb{A} \\ \Phi_{\mathcal{L}}(\varepsilon) &= \varepsilon \end{aligned}$$

For the *sequence cancellation* operator  $\xrightarrow{\star}$  we extend upon the generic language  $\otimes_{\mathcal{L}}^{\star}$  by allowing a matching cancellation path to be executed, after which we continue with the rest of the process tree as normal.

$$\begin{aligned} \xrightarrow{\star}_{\mathcal{L}}(L_1, \dots, L_n) &= \{ \sigma_1 \cdot \langle a \rangle \cdot \sigma_c \mid \sigma_1 \cdot \langle \star_a^C \rangle \in \Phi_{\mathcal{L}}(L_1) \\ &\quad \wedge \text{head}(\sigma_c) \in C \wedge \sigma_c \in \bigcup_{2 \leq j \leq n} L_j \} \\ &\cup \otimes_{\mathcal{L}}^{\star}(L_1, \dots, L_n) \end{aligned}$$

For the *loop cancellation* operator  $\circlearrowleft^{\star}$  we extend upon the generic language  $\otimes_{\mathcal{L}}^{\star}$  by allowing a matching cancellation loop-back path to be executed, after

which we loop back and try executing  $Q_1$  again.

$$\begin{aligned} \overset{\star}{\mathcal{O}}_{\mathcal{L}}(L_1, \dots, L_n) &= \{ \sigma_1 \cdot \langle a_1 \rangle \cdot \sigma'_1 \cdot \sigma_2 \cdot \langle a_2 \rangle \cdot \sigma'_2 \cdot \dots \\ &\quad \cdot \sigma_{m-1} \cdot \langle a_{m-1} \rangle \cdot \sigma'_{m-1} \cdot \sigma_m \\ &\quad | \sigma_m \in \overset{\star}{\mathcal{O}}_{\mathcal{L}}(L_1, \dots, L_n) \\ &\quad \wedge \forall i < m : \sigma_i \cdot \langle \star_{a_i}^{C_i} \rangle \in \Phi_{\mathcal{L}}(L_1) \\ &\quad \quad \wedge \text{head}(\sigma'_i) \in C_i \wedge \sigma'_i \in \bigcup_{2 \leq j \leq n} L_j \} \\ &\cup \overset{\star}{\mathcal{O}}_{\mathcal{L}}(L_1, \dots, L_n) \end{aligned}$$

■ **Example 7.1** To demonstrate the basics of the cancellation semantics, consider the tree depicted in Table 7.2. The sequential subtree at the bottom yields the following language:

$$\begin{aligned} \mathcal{L}(\rightarrow(b, \star_c^{\{e,r\}}, \star_d^{\{e\}})) &= \{ \langle b, c, d \rangle, \langle b, c, \star_d^{\{e\}} \rangle, \\ &\quad \langle b, \star_c^{\{e,r\}}, d \rangle, \langle b, \star_c^{\{e,r\}}, \star_d^{\{e\}} \rangle \} \end{aligned}$$

Consider the loop cancellation subtree. The right subtree starts with activity  $r$ . Hence, we can match the cancellation triggers on  $r$ , but have to propagate the triggers on  $e$  up the tree. We will first take a look at the common cases.

$$\begin{aligned} \Phi_{\mathcal{L}}(\mathcal{L}(\rightarrow(b, \star_c^{\{e,r\}}, \star_d^{\{e\}}))) &= \{ \langle b, c, d \rangle, \langle b, c, \star_d^{\{e\}} \rangle, \langle b, \star_c^{\{e,r\}} \rangle \} \\ \mathcal{L}(\overset{\star}{\mathcal{O}}(\rightarrow(b, \star_c^{\{e,r\}}, \star_d^{\{e\}}), r)) &= \{ \langle b, c, d \rangle, \langle b, c, \star_d^{\{e\}} \rangle, \langle b, \star_c^{\{e\}} \rangle \} \end{aligned}$$

Note how Case 2 removes the  $r$  in the last trace's trigger set. Using this basis, we can now deduce the complete loop cancellation language. Note that  $\star_c^{\{e,r\}}$  in the left subtree's language is replaced by  $c$  followed by the loop-back over  $r$ .

$$\begin{aligned} \mathcal{L}(\overset{\star}{\mathcal{O}}(\rightarrow(b, \star_c^{\{e,r\}}, \star_d^{\{e\}}), r)) &= \{ \langle b, c, d \rangle, \langle b, c, \star_d^{\{e\}} \rangle, \langle b, \star_c^{\{e\}} \rangle \\ &\quad \langle b, c, r, b, c, d \rangle, \langle b, c, r, b, c, \star_d^{\{e\}} \rangle, \\ &\quad \langle b, c, r, b, \star_c^{\{e\}} \rangle, \langle b, c, r, b, c, r, b, c, d \rangle, \dots \} \end{aligned}$$

Observe how the trigger sets with activity  $e$  are exposed in the above language. The sequence cancellation further up the tree can now match on these unresolved trigger markers, yielding the language shown in Table 7.2.

As another example, the tree from Figure 7.1 has the following language:

$$\begin{aligned} \mathcal{L}(\rightarrow(i, \overset{\star}{\rightarrow}(\rightarrow(\times(a, \star_b^{\{h\}}), p), \rightarrow(h, r)), o)) &= \{ \langle i, a, p, o \rangle, \langle i, b, p, o \rangle, \\ &\quad \langle i, a, h, r, o \rangle, \langle i, b, h, r, o \rangle \} \end{aligned}$$

■

■ **Example 7.2** Cancellation operators can be nested in various ways. When a cancellation trigger occurs in a right subtree, it is propagated past the first cancellation operator. This is because the semantics of the cancellation operator only searches for trigger markers in the first or left subtree. For example:

$$\mathcal{L}(\overset{\star}{\rightarrow}(\overset{\star}{\rightarrow}(\overset{\star}{\star}_a^{\{b\}}, \overset{\star}{\rightarrow}(\overset{\star}{\star}_b^{\{b\}}, c)), \rightarrow(b, d))) = \{ \langle a \rangle, \langle a, b, c \rangle, \langle a, b, b, d \rangle \}$$

Also cancellation operators can occur in the right subtree, modeling additional local cancellation regions. Consider for example:

$$\mathcal{L}(\overset{\star}{\rightarrow}(\overset{\star}{\rightarrow}(\overset{\star}{\star}_a^{\{b\}}, \overset{\star}{\star}_b^{\{c\}}), \overset{\star}{\rightarrow}(\overset{\star}{\star}_c^{\{d\}}, d))) = \{ \langle a \rangle, \langle a, b \rangle, \langle a, b, c \rangle, \langle a, b, c, d \rangle \}$$

■

Observe that cancellation process trees are not automatically sound. Below, we define and discuss soundness for cancellation process trees.

**Definition 7.2.3 — Sound Cancellation Process Tree.** A cancellation process tree  $Q$  is *sound* if and only if:

1. its language  $\mathcal{L}(Q)$  does not contain unresolved trigger markers:

$$\forall \sigma \in \mathcal{L}(Q) : \neg(\exists a \in \mathbb{A} \wedge C \subseteq \mathbb{A} : \star_a^C \in \sigma)$$

2. every cancellation path in every cancellation subtree has a corresponding cancellation trigger (i.e.,  $Q$  does not contain dead subtrees):

$$\begin{aligned} \forall \otimes(Q_1, \dots, Q_n) \in \text{enum}(Q) \wedge \otimes \in \{ \overset{\star}{\rightarrow}, \overset{\star}{\circ} \} : & \quad \text{for all cancel subtrees} \\ \forall i \geq 2 \wedge \sigma_i \in \mathcal{L}(Q_i) : & \quad \text{for each cancel path} \\ \exists a \in \mathbb{A} \wedge C \subseteq \mathbb{A} \wedge \sigma_1 \in \mathcal{L}(Q_1) \wedge \star_a^C \in \sigma_1 : & \quad \text{exists a trigger marker} \\ \text{head}(\sigma_i) \in C & \quad \text{matching the cancel path} \end{aligned}$$

All of the example trees shown above are sound. The following trees, for example, are *not sound*:

$$\begin{aligned} \star_a^{\{c\}} & \quad \textbf{Issue:} \text{ language contains an unresolved trigger marker} \\ \overset{\star}{\rightarrow}(\star_a^{\{c\}}, b) & \quad \textbf{Issue:} \text{ language contains an unresolved trigger marker and} \\ & \quad \text{the cancellation path } b \text{ has no corresponding trigger} \\ \overset{\star}{\rightarrow}(a, b) & \quad \textbf{Issue:} \text{ the cancellation path } b \text{ has no corresponding trigger} \\ \overset{\star}{\rightarrow}(\star_a^{\{c\}}, \times(b, c)) & \quad \textbf{Issue:} \text{ the cancellation path } b \text{ has no corresponding trigger} \end{aligned}$$

Like with the original process tree operators, multiple cancellation process trees can describe the same language. We extend the original reduction rules from Table 2.1 on page 42 with the cancellation reduction rules in Table 7.3.

**Table 7.3:** Reduction rules for cancelation process trees.

$$\begin{aligned} \otimes(\dots_1, \times(\dots_2), \dots_3) &= \otimes(\dots_1, \dots_2, \dots_3) && \text{for } \otimes \in \left\{ \overset{\star}{\rightarrow}, \overset{\star}{\circ} \right\} \\ \otimes(\otimes(Q_1, \dots_2), \dots_3) &= \otimes(Q_1, \dots_3) && \text{for } \otimes \in \left\{ \overset{\star}{\rightarrow}, \overset{\star}{\circ} \right\} \\ &&& \text{if } \neg \exists c \in \text{Start}(\dots_2) \wedge \star_a^C \in \text{enum}(Q_1) : c \in C^1 \end{aligned}$$

<sup>1</sup> For the last rewrite rule, we have to make sure that the inner cancelation region is not triggered (i.e., is a dead subtree) to preserve semantics.

## 7.3 The Trigger Oracle

For the cancelation process tree presented in Section 7.2, we relied on explicitly modeling cancelation triggers and trigger activities. For the cancelation discovery algorithm we present in Section 7.4, we also assume that the trigger activities are explicit in the input. However, this is usually not the case for an event log.

To capture the knowledge of cancelation trigger activities, we make trigger activities explicit using the so-called trigger oracle as defined below.

**Definition 7.3.1 — Trigger Oracle.** Let  $\mathbb{A}$  be a set of activities. A *trigger oracle* is a function  $\text{isTrigger} : \mathbb{A} \mapsto \{ \text{true}, \text{false} \}$  yielding *true* if and only if an activity  $a \in \mathbb{A}$  is a trigger activity.

In practice, there are many sources that can be used to instantiate a trigger oracle. In this section, we will discuss a few common heuristics.

### 7.3.1 Trigger Oracles in Software

In software, there are various ways to identify where exceptions, errors, and other type of cancelation or error behavior occur. Below, we discuss some of the possible heuristics that can be used in a software setting.

#### Exception and Error Data

Often, in software, extra data can be recorded whenever an exception, error, or other type of cancelation behavior occurs. For example, consider the event log in Table 7.1. In this event log, we can clearly point out in case 2 where the exception was thrown, and where it was caught, by only looking at the lifecycle information and activity names.

In other cases, extra exception and error data attributes may be logged. For example, for the program in Listing 7.2, we could track the value of the `result` variable. In such a case, the trigger oracle would track what happens once the variable is updated to `result = -1`. Recall also the *exception data* software event data from page 107.

### Domain Knowledge and Naming Schemes

When no extra data is recorded, one can use external domain knowledge. For example, an interface specification could identify certain activities as error events, timed triggers, asynchronous events, etc. In such cases, one could use a dictionary-based trigger oracle instantiation, listing the activities that are known trigger activities.

In other cases, a specific naming scheme or domain-based patterns could be used to parse activity names and identify trigger activities. For example, in a particular domain, project, or company, one could adopt the convention to label error callbacks using the following naming pattern: `callback_<function>.nok` (*nok* stands for *not ok*). Any activity name that follows this pattern can hence be identified as a trigger activity.

### 7.3.2 Non-software-specific Trigger Oracles

In non-software-specific settings, it is often also possible to identify the trigger activities. Below, we discuss some possible heuristics that can be generally applicable, be it software event logs, business event logs, or any other event logs.

#### Naming Semantics

A simple heuristic that can often be applied is to have a domain expert or semantic-based oracle identify trigger activities. In most cases, activities or specific keywords in activity labels can be recognized as trigger activities. For example, a negative activity name like “Canceled” or “Declined” is often a good candidate for a trigger activity.

#### Activity Footprints

Another heuristic is to consider the footprints of specific activities in the event log. For example, if an activity is independent of the state of a (sub)process and can happen at rather arbitrary points, it could hint at cancellation behavior. Techniques based on, for example, the entropy of activities [178], can be used to identify (candidate) trigger activities.

Another approach is to use the fallback extension point from the Inductive Miner, see Section 4.2.5. Observe that process trees only capture block-structured behavior, and cancellation behavior often has a non-block-structured footprint in the directly-follows graph. Therefore, as a fallback case, one could identify the offending activity and mark it as a trigger activity.

#### Optimization Strategy

Alternatively, we can implement an optimization strategy based on the principle that process trees only capture block-structured behavior, and cancellation behavior breaks this block-structuredness. The intuition is that, if cancellation behavior is present, modeling this behavior with cancellation operators

will yield a more fitting and possibly more precise process tree. Such an optimization strategy would feed several candidate trigger oracle functions to the discovery algorithm, compute the fitness and precision of the resulting model, and use the best scoring candidate.

## 7.4 Cancellation Discovery

In this section, we introduce the *cancellation discovery* algorithm. In this algorithm, we extend the Inductive miner (IM) framework (see Chapter 4) with support for the cancellation operators introduced in Definition 7.2.1. Section 7.4.1 gives an overview of the algorithm, Section 7.4.2 details our cancellation extensions, Section 7.4.4 shows additional discovery examples, and Section 7.4.5 covers the discovery guarantees maintained for our extension.

### 7.4.1 Algorithm Overview

For the *cancellation discovery* algorithm, we will mostly follow the traditional IM divide and conquer approach (see also Section 4.2 on page 74). Given a log  $L$  and trigger oracle  $isTrigger$ , we search for possible splits of  $L$  into sublogs  $L_1, \dots, L_n$ , such that these sublogs combined with a process tree operator  $\otimes \in \{\rightarrow, \times, \wedge, \circ, \overset{\star}{\rightarrow}, \overset{\star}{\circ}\}$  can (at least) reproduce  $L$  again. The framework then recurses on the corresponding sublogs, repeats the above process, and returns the discovered submodels as normal. In the cancellation discovery algorithm, we assume a trigger oracle  $isTrigger$  as additional input, we include the cancellation operator  $\overset{\star}{\rightarrow}, \overset{\star}{\circ}$  in the log split search, and we will adopt slightly different base cases. Note that we will assume atomic event logs for the remainder of this section.

In the cancellation discovery, we will enrich the directly-follows graph with information from the trigger oracle. Using this enriched graph, we can detect the new cancellation operators and adapt the cut detection of existing operators were needed to support the prefix-based semantics introduced by  $\Phi_{\mathcal{L}}(L)$  in Definition 7.2.2. We will reuse the existing empty log and empty traces base case (i.e., Base Case 4.1). However, the single activity base case (i.e., Base Case 4.2) is modified to consider the cancellation trigger leaf.

Below, we will discuss these adaptations in detail. We will be using the example run in Table 7.4 as clarification.

### 7.4.2 Framework Extensions

Formally, in the *cancellation discovery* algorithm, we use most of the original cut detections and log splits, and reuse all of the fallbacks as defined in Section 4.2 on page 74. In this addition, we will:



- adapt the directly-follows graph definition slightly, such that we can integrate the trigger oracle knowledge,
- use alternative base cases as detailed below, and
- introduce new cut detections and log splits for the cancellation cases.

In the following formulas and examples, we will write  $\text{CDiscover}(L, isTrigger)$  to refer to the *cancellation discovery* (CD) algorithm, i.e., the cancellation instantiation of the IM framework (see Algorithm 7.1).

---

### Algorithm 7.1: Cancellation Discovery (CD) Algorithm

---

**Input:** An event log  $L$ .

**Output:** A cancellation process tree  $Q$  such that  $L$  fits  $Q$ .

**Description:** The function  $\text{CDiscover}()$  recursively tries to discover a hierarchical process tree capturing (at least) the behavior in  $L$ . We use  $\perp$  to model when no valid base case or valid cut was found for the given log.

```

CDiscover(L)
1   $Q_{base} = \text{BaseCase}(L)$  // Base Cases 4.1, 7.1, and 7.2
2  if  $Q_{base} \neq \perp$  then
3    return  $Q_{base}$  // Returns either  $\tau$ ,  $a$ , or  $\star_a^{triggers(a)}$ 
4  if  $\varepsilon \notin L$  then
5    // Cut Detections 7.1 and 7.2 plus adapted versions of non-cancellation Cut
    // Detections 4.1, 4.2, 4.3, and 4.4
6     $(\otimes, (\Sigma_1, \dots, \Sigma_n)) = \text{FindCut}(L)$ 
7    if  $(\otimes, (\Sigma_1, \dots, \Sigma_n)) \neq \perp$  then
8       $(L_1, \dots, L_n) = \text{SplitLog}(L, (\otimes, (\Sigma_1, \dots, \Sigma_n)))$ 
9      return  $\otimes(\text{CDiscover}(L_1), \dots, \text{CDiscover}(L_n))$ 
10 return  $\text{Fallback}(L)$ 

```

---

### Directly-Follows Graph for Cancellation Discovery

In cancellation discovery, the directly-follows graph is still constructed as normal. However, using the knowledge from the trigger oracle  $isTrigger$ , we slightly adapt the associated semantics as defined below.

**Definition 7.4.1 — Trigger Edges and Cancellation Triggers.** Let  $G(L)$  be the directly-follows graph over  $L$  and let  $isTrigger : \mathbb{A} \mapsto \{true, false\}$  be a trigger oracle.

An edge  $(a, b) \in G$  is a *trigger edge*, notation  $isTrigger(a, b)$ , if and only if node  $b$  is a trigger activity  $isTrigger(b)$ .

A path  $a \rightsquigarrow b \in G$  is a *trigger path* notation  $isTrigger(a, b)$ , if and only if at least one trigger edge is part of the path.

The set of *cancellation triggers* at a node  $a$ , notation  $triggers(a) = C$ , is defined as the set of nodes that have an trigger edge from  $a$ , i.e.,  $triggers(a) = \{b \mid isTrigger(a, b)\}$ .

In any subgraph  $G' \subseteq G$ , a node  $a$  can only be an end node of  $G'$  if and only if it would be an end node without trigger edges, i.e.,  $a \in \text{End}(G') \Leftrightarrow \exists (a, b) \in G : \neg isTrigger(a, b)$ .

**Table 7.4:** Example Cancellation Discovery on the log  $L = [\langle i, a, p, o \rangle, \langle i, b, p, o \rangle, \langle i, b, h, r, o \rangle]$  with trigger oracle  $isTrigger(h) = true$  and for any  $x \neq h$  we have  $isTrigger(x) = false$ . The rows illustrate how the discovery progresses step by step. The highlights indicate the parts of the log and directly-follows graph used, and relate them to the corresponding partial process tree model that is discovered. The dashed arrow in directly-follows graph indicates a trigger edge; the dashed lines indicate the cuts. The corresponding Reset net is shown at the bottom. See also Figure 7.1.

Event Log	Sublog View	Directly-Follows Graph	Disc. Model
<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border: 1px solid black; padding: 2px; display: inline-block;">i a p o</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">i b p o</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">i b h r o</div> </div>	$[\langle i, a, p, o \rangle,$ $\langle i, b, p, o \rangle,$ $\langle i, b, h, r, o \rangle]$		
<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border: 1px solid black; padding: 2px; display: inline-block;">i a p o</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">i b p o</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">i b h r o</div> </div>	$[\langle a, p \rangle,$ $\langle b, p \rangle,$ $\langle b, h, r \rangle]$		
<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border: 1px solid black; padding: 2px; display: inline-block;">i a p o</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">i b p o</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">i b h r o</div> </div>	$[\langle h, r \rangle]$		
<div style="display: flex; flex-direction: column; gap: 5px;"> <div style="border: 1px solid black; padding: 2px; display: inline-block;">i a p o</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">i b p o</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">i b h r o</div> </div>	$[\langle a, p \rangle,$ $\langle b, p \rangle,$ $\langle b \rangle]$		

For example, consider the directly-follows graph in Table 7.4. Since we have  $isTrigger(h)$ , the edge  $(b, h)$  is a trigger edge  $isTrigger(b, h)$ , as indicated by the dashed arrow. In the subgraph  $\{a, b, p\}$  in step 4 and 5, node  $p$  is an end node, but node  $b$  is not (there is no non-trigger exit edge for node  $b$ ). Node  $b$  has a set of cancellation triggers ( $triggers(b) = \{h\}$ ), but node  $p$  has no associated cancellation triggers, i.e., the empty set ( $triggers(p) = \emptyset$ ).

### Base Cases

The *cancellation discovery* algorithm uses Base Case 4.1 as defined on page 75 and Base Cases 7.1 and 7.2 as defined below.

#### ■ Base Case 7.1 — Single Activity – No Cancellation Trigger.

*Condition:*  $L \neq [] \wedge (\exists a \in \mathbb{A} : triggers(a) = \emptyset \wedge \forall \sigma \in L : \sigma = \langle a \rangle)$

*Return:*  $a$

*Description:* The *Single Activity – No Cancellation Trigger* base case applies when the traces in the log contains only events with a single activity label  $a$  and node  $a$  has no cancellation triggers in the original directly-follows graph. This base case returns activity  $a$  as a leaf node.

*Example:* Consider step 1 in Table 7.4. After the sequence cut, we have a sublog consisting of one event labelled  $i$  with no associated cancellation triggers. Hence, this base case holds and we return  $i$  as a leaf node. Observe that this base case does not apply in step 5 for the sublog with  $b$  since we have  $triggers(b) = \{h\}$  (see the dashed arrow).

#### ■ Base Case 7.2 — Single Activity – Cancellation Trigger.

*Condition:*  $L \neq [] \wedge (\exists a \in \mathbb{A} : triggers(a) \neq \emptyset \wedge \forall \sigma \in L : \sigma = \langle a \rangle)$

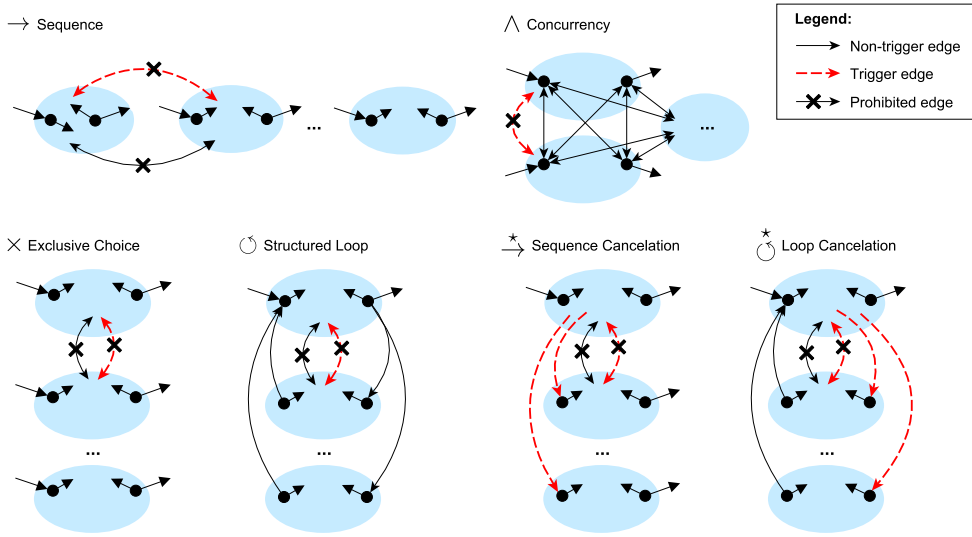
*Return:*  $\star_a^{triggers(a)}$

*Description:* The *Single Activity – Cancellation Trigger* base case applies when the traces in the log contains only events with a single activity label  $a$ . and node  $a$  has no cancellation triggers in the original directly-follows graph. This base case returns the cancellation trigger  $\star_a^{triggers(a)}$  as a leaf node, allowing to trigger any of the cancellation regions associated with a trigger activity  $b \in triggers(a)$ .

*Example:* Consider step 5 in Table 7.4. After the sequence and choice cut, we the sublog  $[\langle b \rangle^2]$ , and from the directly-follows graph we conclude  $triggers(b) = \{h\}$  (see the dashed arrow). Hence, this base case holds and we return the cancellation trigger  $\star_b^{\{h\}}$  as a leaf node.

### Finding Cuts

When no base case applies, the *cancellation discovery* algorithm searches for possible log splits and corresponding process tree operators. We include support for our cancellation operators by adding new cut detections and slightly adapting the existing cut definitions from Section 4.2.3 on page 76. For each process tree operator, a different cut or footprint is characterized based on



**Figure 7.2:** Cuts of the directly-follows graph for the cancellation discovery algorithm. The grey areas indicate partitions; the arrows indicate required and prohibited edges characterizing the cut. Dashed arrows indicate trigger edges, solid arrows indicate non-trigger edges.

the edges between nodes in  $G(L)$ . Figure 7.2 informally depicts cuts of the directly-follows graph as used in the *cancellation discovery* algorithm. We formally define these directly-follows graph cuts below. Recall that graph cuts, partitions  $(\Sigma_i)$  and path in a graph ( $\rightsquigarrow$ ) were explained in Section 2.1.2 on page 23.

In our cancellation discovery, any non-cancellation cut (i.e., for the operators  $\rightarrow$ ,  $\wedge$ ,  $\times$ , and  $\odot$ ) *cannot* have a trigger edge between two partitions. In contrast, a cancellation cut (i.e., for the operators  $\overset{*}{\rightarrow}$  and  $\overset{*}{\odot}$ ) is characterized by having trigger edges from its first partition (i.e., its cancellation region body) to all non-first partitions (i.e., the cancellation paths).

**Cut Detection Adaptation for Non-Cancellation Cuts.** We assume for all the non-cancellation cuts defined in Section 4.2.3 on page 76 that all the rules are defined over non-trigger edges. In addition, we assume that there are no trigger edges between any of partition  $\Sigma_i$  and  $\Sigma_j$  with  $i \neq j$ :

$$\forall i \neq j \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : a_i \rightsquigarrow a_j \in G \Rightarrow \neg isTrigger(a_i, a_j)$$

■ **Cut Detection 7.1 — Sequence Cancellation ( $\overset{*}{\rightarrow}$ ).**

*Description:* Directly follows graph  $G$  can be partitioned with a partially ordered cut such that the first partition represents the cancellation body and the non-first partitions represent the cancellation alternative paths.

*Definition:* A sequence cancel ( $\overset{\star}{\rightarrow}$ ) cut is a partially ordered cut  $\Sigma_1, \dots, \Sigma_n$  of a directly follows graph  $G$  such that:

1. All start activities are in the body  $\Sigma_1$ :

$$Start(G) \subseteq \Sigma_1$$

2. Every partition  $\Sigma_i$  has some end activities:

$$\forall i \geq 1 : End(G) \cap \Sigma_i \neq \emptyset$$

3. All edges from  $\Sigma_1$  to  $\Sigma_i$ , with  $i \geq 2$ , are trigger edges:

$$\forall i \geq 2 \wedge a_i \in \Sigma_i \wedge a_1 \in \Sigma_1 : (a_1, a_i) \in G \Rightarrow isTrigger(a_1, a_i)$$

4. There are no edges from  $\Sigma_i$  to  $\Sigma_j$ , where  $i \geq 2 \wedge j \geq 1 \wedge i \neq j$ :

$$\forall i \geq 2 \wedge j \geq 1 \wedge i \neq j \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : (a_i, a_j) \notin G$$

*Example:* In step 2 in Table 7.4, a sequence cancel cut is detected in the directly-follows graph, yielding two partitions: body  $\Sigma_1 = \{a, b, p\}$  and cancellation alternative  $\Sigma_2 = \{h, r\}$ . Partition  $\Sigma_1$  is handled in recursive steps 4 and 5, partition  $\Sigma_2$  is handled in recursive step 3.

### ■ Cut Detection 7.2 — Loop Cancellation ( $\overset{\star}{\circ}$ ).

*Description:* Directly follows graph  $G$  can be partitioned with a partially ordered cut such that the first partition represents the cancellation body and the non-first partitions represent the cancellation loop-back paths.

*Definition:* A loop cancel ( $\overset{\star}{\circ}$ ) cut is a partially ordered cut  $\Sigma_1, \dots, \Sigma_n$  of a directly follows graph  $G$  such that:

1. All start and end activities are in the body  $\Sigma_1$ :

$$Start(G) \cup End(G) \subseteq \Sigma_1$$

2. All edges from  $\Sigma_1$  to  $\Sigma_i$ , with  $i \geq 2$ , are trigger edges:

$$\forall i \geq 2 \wedge a_i \in \Sigma_i \wedge a_1 \in \Sigma_1 : (a_1, a_i) \in G \Rightarrow isTrigger(a_1, a_i)$$

3. There are only edges from  $\Sigma_i$ , with  $i \geq 2$ , to start nodes in  $\Sigma_1$ :

$$\forall i \geq 2 \wedge a_i \in \Sigma_i \wedge a_1 \in \Sigma_1 : (a_i, a_1) \in G \Rightarrow a_1 \in Start(G)$$

4. There are no edges from  $\Sigma_i$  to  $\Sigma_j$ , where  $i \geq 2 \wedge j \geq 2 \wedge i \neq j$ :

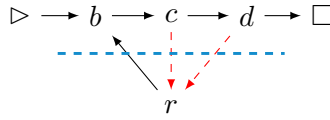
$$\forall i \geq 2 \wedge j \geq 2 \wedge i \neq j \wedge a_i \in \Sigma_i \wedge a_j \in \Sigma_j : (a_i, a_j) \notin G$$

5. If  $\Sigma_i$  with  $i \geq 2$  has an edge to  $\Sigma_1$ , it connects to all start activities:

$$\forall i \geq 2 \wedge a_i \in \Sigma_i \wedge a_1 \in Start(G) : (a_i, a_1) \in G$$

$$\Leftrightarrow (\exists a'_1 \in \Sigma_1 : (a_i, a'_1) \in G)$$

*Example:* Consider the log  $L = [\langle b, c, d \rangle, \langle b, c, r, b, c, d \rangle, \langle b, c, d, r, b, c, d \rangle]$  and trigger oracle  $isTrigger = \{r \mapsto true\}$ . Figure 7.3 shows the loop cancel cut in the corresponding directly-follows graph, yielding two partitions: body  $\Sigma_1 = \{b, c, d\}$  and cancellation loop-back  $\Sigma_2 = \{r\}$ . Partition  $\Sigma_1$  can be further partitioned by a normal sequence cut, partition  $\Sigma_2$  can be handled by Base Case 7.1.



**Figure 7.3:** Example directly-follows graph with loop cancel cut for the log  $L = [\langle b, c, d \rangle, \langle b, c, r, b, c, d \rangle, \langle b, c, d, r, b, c, d \rangle]$  and trigger oracle  $isTrigger = \{r \mapsto true\}$ , derived from Example 7.1. The dashed arrows indicate trigger edges; the dashed lines indicate the cuts. Note that the cancellation trigger  $r$  can be triggered from both activity  $c$  and  $d$ .

### 7.4.3 Splitting Logs

Once a valid cut  $\Sigma_1, \dots, \Sigma_n$  has been found for a given process tree operator  $\otimes$ , we split the log  $L$  according to the cut into sublogs  $L_1, \dots, L_n$  such that these logs combined with the operator  $\otimes$  can (at least) reproduce  $L$  again. We include support for our cancellation operators by adding new log splits and slightly adapting the existing log splits from Section 4.2.4 on page 79. We formally define these log splits below. Recall that activity log-projection ( $\mathbb{A}(\sigma)$ ) was explained in Definition 2.3.6 on page 48.

**Log Split Adaptation for Non-Cancellation Cuts.** We assume for all the non-cancellation cuts defined in Section 4.2.4 on page 79 that no empty traces are added as a result from trigger edges, i.e., we take into account the prefix-based semantics introduced by  $\Phi_{\mathcal{L}}(L)$ . This can be achieved by marking events upon log splitting if they are directly followed by trigger activities. Whenever a trace only contains events marked in this way, and after a log-split projection contains no events, then we can discard the resulting empty trace.

For example, consider the example from Figure 7.3. Suppose we used a loop cancellation cut, yielding the body partition  $\Sigma_1 = \{b, c, d\}$  and sublog  $L_1 = [\langle b, c, d \rangle^4, \langle b, c \rangle]$  where the last  $c$  is marked because it directly precedes  $r$ . (see Log Split 7.2 below). Applying a sequence cut on sublog  $L_1$  would yield the partitions  $\Sigma_{1,1} = \{b\}$ ,  $\Sigma_{1,2} = \{c\}$ , and  $\Sigma_{1,3} = \{d\}$ . For the partition  $\Sigma_{1,3}$ , the sublog would not include the empty trace as a result of splitting  $\langle b, c \rangle$  because in this trace the event labeled  $c$  is marked.

#### ■ Log Split 7.1 — Sequence Cancellation ( $\overset{*}{\rightarrow}$ ).

*Description:* Each trace in the log is split into subtraces of the cancellation body and the cancellation alternative partitions.

*Definition:* Given a sequence cancel cut  $\Sigma_1, \dots, \Sigma_n$  and event log  $L$ :

- Sublog  $L_1$  consists of all maximal prefix subtraces with activities in  $\Sigma_1$ :

$$L_1 = \{ \sigma_1 \mid \sigma_1 \cdot \sigma_2 \in L \wedge \mathbb{A}(\sigma_1) \subseteq \Sigma_1 \\ \wedge (\sigma_2 = \varepsilon \vee (\sigma_2 = \langle c, \dots \rangle \wedge c \notin \Sigma_1)) \}$$

- Sublog  $L_{i \geq 2}$  consists of all maximal postfix substraces with activities in  $\Sigma_i$ :

$$L_{i > 1} = \{ \sigma_2 \mid \sigma_1 \cdot \sigma_2 \in L \wedge \mathbb{A}(\sigma_2) \subseteq \Sigma_i \\ \wedge (\sigma_1 = \varepsilon \vee (\sigma_1 = \langle \dots, a_1 \rangle \wedge a_1 \in \Sigma_1)) \}$$

*Example:* In step 2 in Table 7.4, we have the input log  $L = [\langle a, p \rangle, \langle b, p \rangle, \langle b, h, r \rangle]$  and sequence cancel cut  $\Sigma_1 = \{a, b, p\}$  and  $\Sigma_2 = \{h, r\}$ . As a result,  $L$  is split into the sublogs  $L_1 = [\langle a, p \rangle, \langle b, p \rangle, \langle b \rangle]$  and  $L_2 = [\langle h, r \rangle]$ .

### ■ Log Split 7.2 — Loop Cancellation ( $\odot$ ).

*Description:* Each trace in the log is split into substraces of the cancellation body and the cancellation loop-back partitions.

*Definition:* Given a loop cancel cut  $\Sigma_1, \dots, \Sigma_n$  and event log  $L$ :

- Sublog  $L_i$  consists of all maximal substraces with activities in  $\Sigma_i$ :

$$L_i = \{ \sigma_2 \mid \sigma_1 \cdot \sigma_2 \cdot \sigma_3 \in L \wedge \mathbb{A}(\sigma_2) \subseteq \Sigma_i \\ \wedge (\sigma_1 = \varepsilon \vee (\sigma_1 = \langle \dots, a_1 \rangle \wedge a_1 \notin \Sigma_i)) \\ \wedge (\sigma_3 = \varepsilon \vee (\sigma_3 = \langle a_3, \dots \rangle \wedge a_3 \notin \Sigma_i)) \}$$

*Example:* In the example from Figure 7.3, we have the input log  $L = [\langle b, c, d \rangle, \langle b, c, r, b, c, d \rangle, \langle b, c, d, r, b, c, d \rangle]$  and loop cancel cut  $\Sigma_1 = \{b, c, d\}$  and  $\Sigma_2 = \{r\}$ . As a result,  $L$  is split into the sublogs  $L_1 = [\langle b, c, d \rangle^4, \langle b, c \rangle]$  and  $L_2 = [\langle r \rangle^2]$ .

## 7.4.4 Discovery Examples

Table 7.4 shows a step-by-step example run of the *cancellation discovery* algorithm. Consider the sequence cancel cut detected in step 2 and the cancellation trigger discovered in step 5. Note the subtlety of the non-cancellation log splits applying the prefix semantics on the subtrace  $\langle b \rangle$  in step 4 and 5. If we would use the original Inductive Miner without the cancellation operators, Fallback 4.6 would be used in step 2, yielding  $\text{IMdiscover}(L) = \rightarrow(i, \odot(\times(a, b, p, h, r), \tau), o)$ .

Below are some more example logs  $L$  and trigger oracles  $isTrigger$  together with the discovered models. In addition, the  $\text{IMdiscover}$  result is given to compare with the original Inductive Miner without the cancellation operators.

$$\begin{aligned} & \text{CDiscover}(\overbrace{[\langle a, b, c, d, f \rangle, \langle a, b, c, d, e, b, r_1, f \rangle]}^L, \overbrace{\{r_1 \mapsto true\}}^{isTrigger}) \\ &= \rightarrow(a, \text{CDiscover}([\langle b, c, d \rangle, \langle b, c, d, e, b, r_1 \rangle], \{r_1 \mapsto true\}), f) \\ &= \rightarrow(a, \overset{*}{\rightarrow}(\text{CDiscover}([\langle b, c, d \rangle, \langle b, c, d, e, b \rangle], \{r_1 \mapsto true\}), r_1), f) \\ &= \rightarrow(a, \overset{*}{\rightarrow}(\odot(\text{CDiscover}([\langle b, c, d \rangle, \langle b, c, d \rangle, \langle b \rangle], \{r_1 \mapsto true\}), e), r_1), f) \\ &= \rightarrow(a, \overset{*}{\rightarrow}(\odot(\rightarrow(\overset{\{r_1\}}{\star_b}, c, d), e), r_1), f) \end{aligned}$$

$$\begin{aligned}
& \text{IMdiscover}([\langle a, b, c, d, f \rangle, \langle a, b, c, d, e, b, r_1, f \rangle]) \\
&= \rightarrow(a, \wedge(\odot(b, e), c, d), \times(r_1, \tau), f) \\
\text{CDdiscover}([\langle a, b, c, d, e, f \rangle, \langle a, b, c, r_1, g, b, c, d, e, f \rangle, \\
&\quad \langle a, b, c, d, e, r_1, g, b, c, d, e, f \rangle], \underbrace{\{r_1 \mapsto true\}}_{isTrigger}) \\
&= \rightarrow(a, \text{CDdiscover}([\langle b, c, d, e \rangle, \langle b, c, r_1, g, b, c, d, e \rangle, \\
&\quad \langle b, c, d, e, r_1, g, b, c, d, e \rangle], \{r_1 \mapsto true\}), f) \\
&= \rightarrow(\odot^*(\text{CDdiscover}([\langle b, c, d, e \rangle^4, \langle b, c \rangle], \{r_1 \mapsto true\}), \\
&\quad \text{CDdiscover}([\langle r_1, g \rangle^2], \{r_1 \mapsto true\})), f) \\
&= \rightarrow(\odot^*(\rightarrow(b, \star_c^{\{r_1\}}, d, \star_e^{\{r_1\}}), \rightarrow(r_1, g)), f) \\
\text{IMdiscover}([\langle a, b, c, d, e, f \rangle, \langle a, b, c, r_1, g, b, c, d, e, f \rangle, \\
&\quad \langle a, b, c, d, r_1, g, b, c, d, e, f \rangle]) \\
&= \rightarrow(a, \odot(\text{IMdiscover}([\langle b, c, d, e \rangle, \langle b, c, r_1, g \rangle, \langle b, c, d, e \rangle, \\
&\quad \langle b, c, d, e, r_1, g \rangle, \langle b, c, d, e \rangle]), \tau), f) \\
&= \rightarrow(a, \odot(\rightarrow(b, c, \times(\rightarrow(d, e), \tau), \times(\rightarrow(r_1, g), \tau)), \tau), f)
\end{aligned}$$

Observe how in the above examples skips (i.e.,  $\times(\dots, \tau)$ ) are discovered in the **IMdiscover** models to approximate the prefix-based patterns associated with a cancellation body, as captured by  $\Phi_{\mathcal{L}}(L)$  in Definition 7.2.2.

### 7.4.5 Guarantees

The *cancellation discovery* (CD) algorithm maintains the IM discovery guarantees from Section 4.3 on page 84 and returns a sound cancellation process tree (Definition 7.2.3). In this section, we will discuss these discovery guarantees and properties. We refer the reader to Section A.4 on page 375 for the proofs.

#### Soundness and Termination

We start with two general properties of the CD algorithm: soundness is guaranteed (Theorem 7.4.1), termination is guaranteed (Theorem 7.4.2).

**Theorem 7.4.1** — **CD guarantees soundness.** All models  $Q$  returned by the CD algorithm are guaranteed to be sound.

*Proof.* See page 376. □



**Theorem 7.4.2 — CD guarantees termination.** The CD algorithm is guaranteed to always terminate.

*Proof.* See page 376. □

### Perfect Fitness

The perfect fitness guarantee in Theorem 7.4.3 states that all the log behavior is in the model discovered by the CD algorithm.

**Theorem 7.4.3 — CD guarantees fitness.** The CD algorithm returns a model that fits the log. That is, given an event log  $L$ , the CD algorithm returns a model  $Q$  such that  $L \subseteq \mathcal{L}(Q)$ .

*Proof.* See page 376. □

### Language Rediscoverability

The language rediscoverability property tells whether and under which conditions a discovery algorithm can discover a model that is language-equivalent to the original process. That is, given a ‘system model’  $Q$  and an event log  $L$  that is complete with respect to  $Q$  (for some notion of completeness), then we rediscover a model  $Q'$  such that  $\mathcal{L}(Q') = \mathcal{L}(Q)$ .

For the CD algorithm, we will prove the language rediscoverability property for directly-follows complete (Definition 4.3.1) and trigger complete (see Definition 7.4.2 below) logs and for all cancellation process trees that are:

- *Sound* cancellation process trees (Definition 7.2.3), and
- In the *class of rediscoverable process trees* (Definition 4.3.2).

**Definition 7.4.2 — Trigger Completeness.** A trigger oracle  $isTrigger$  is trigger complete to a model  $Q$ , denoted as  $isTrigger \diamond_{\star} Q$ , if and only if  $isTrigger(a) \Leftrightarrow a \in triggerSet(Q)$ , where  $triggerSet(Q)$  is defined as:

$$triggerSet(Q) = \begin{cases} \left\{ \begin{array}{l} \{ head(\sigma) \mid \sigma \in \bigcup_{i \geq 2} \mathcal{L}(Q_i) \} \\ \cup \bigcup_i triggerSet(Q_i) \end{array} \right. & \text{if } Q = \overset{\star}{\otimes}(Q_1, \dots, Q_n) \\ & \wedge \overset{\star}{\otimes} \in \{ \overset{\star}{\rightarrow}, \overset{\star}{\circ} \} \\ \bigcup_i triggerSet(Q_i) & \text{if } Q = \otimes(Q_1, \dots, Q_n) \\ & \wedge \otimes \notin \{ \overset{\star}{\rightarrow}, \overset{\star}{\circ} \} \\ \emptyset & \text{otherwise} \end{cases}$$

To prove language-rediscoverability, we will use the proof framework used by the base Theorem 4.3.7 as listed on page 85:

1. Show that the base cases can be rediscovered.
2. Show that any root process tree operator can be rediscovered, proving that the cut criteria are correct.

3. Show that for all process tree operators, the graph cut yields the correct activity division and the log is correctly subdivided.
4. Finally, Theorem 7.4.7 uses the above lemmas and base Theorem 4.3.7 to prove language rediscoverability using induction on the model size.

Reusing base Theorem 4.3.7, for the CD algorithm, we have to show that the cancellation trigger can be rediscovered (Lemma 7.4.4), the root tree operator can still be rediscovered (Lemma 7.4.5), and that the log is still correctly subdivided (Lemma 7.4.6).

**Lemma 7.4.4 — CD rediscovered the cancellation trigger leaf.** Let  $Q$  be a reduced model that adheres to the above model restrictions with a leaf  $Q' \in \mathbb{A} \cup \{\tau\} \cup \{\star_a^C \mid a \in \mathbb{A}, C \subseteq \mathbb{A}\}$ , let  $L$  be a log such that  $L \diamond_{df} Q$ , and let  $isTrigger$  be a trigger oracle such that  $isTrigger \diamond_{\star} Q$ . Then  $CDiscover(L)$  returns a tree with a leaf  $Q'$ .

*Proof.* See page 377. □

**Lemma 7.4.5 — CD selects the right tree operator.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the above model restrictions, let  $L$  be a log such that  $L \diamond_{df} Q$ , and let  $isTrigger$  be a trigger oracle such that  $isTrigger \diamond_{\star} Q$ . Then  $CDiscover(L)$  returns a tree with root  $\otimes$ .

*Proof.* See page 378. □

**Lemma 7.4.6 — CD splits logs correctly.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the above model restrictions, let  $L$  be a log such that  $L \diamond_{df} Q$ , and let  $isTrigger$  be a trigger oracle such that  $isTrigger \diamond_{\star} Q$ . Then for the resulting sublogs  $L_i$  produced by CD we have  $L_i \diamond_{df} Q_i \wedge L_i \subseteq \mathcal{L}(Q_i)$ .

*Proof.* See page 378. □

Using the above lemmas and base Theorem 4.3.7, we can prove language rediscoverability using induction on the model size.

**Theorem 7.4.7 — CD guarantees language rediscoverability.** If the model restrictions detailed above hold for a process tree  $Q$ , then  $CDiscover$  language-rediscovered  $Q$ , i.e.,  $\mathcal{L}(Q) = \mathcal{L}(CDiscover(L, isTrigger))$  for any log  $L$  such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$  and any trigger oracle  $isTrigger$  such that  $isTrigger \diamond_{\star} Q$ .

*Proof.* See page 380. □

### Polynomial Runtime Complexity

The basic IM framework is implemented as a polynomial algorithm and scales well with large event logs. The CD algorithm maintains a polynomial runtime complexity.

**Theorem 7.4.8** — **CD has polynomial runtime complexity.** The runtime complexity of the CD algorithm is bounded by  $O(|\mathbb{A}(L)|^5 + |\mathbb{A}(L)| \cdot |L|)$ .

*Proof.* See page 380. □

## 7.5 Compatibility with Other Extensions

The cancellation discovery, as presented in the above sections, allows for the discovery of the new cancellation process tree operators. In this section, we will revisit the existing inductive miner extensions listed in Section 4.4 on page 86 as well as the hierarchical extensions from Chapter 6 and discuss how they can be used in combination with the cancellation discovery extensions.

The *Inductive Miner – infrequent (IMf)* extension handles deviating and infrequent behavior by filtering the directly-follows graph according to some user-chosen frequency threshold when no cut could be found. In addition, the single activity base case and empty traces fallback are slightly altered to take into account infrequent behavior. In our cancellation setting, we can easily integrate with this extension by altering the base cases in a similar way to take into account infrequent behavior. Since our cancellation cut detections are just based on graph patterns as usual, we automatically benefit from the frequency-based filtering on the directly-follows graph.

The *Inductive Miner – incompleteness (IMc)* extension handles incomplete behavior by using probabilistic activity relations in the cut detection. For each cut pattern, an accumulated probability is defined. During cut detection, probabilities are estimated and the cut with the highest probability is selected. For our cancellation operators, we can model an accumulated probability using the definition for the loop accumulated probability as a basis.

By using the rich event logs from Definition 2.3.3, we can integrate with the *lifecycle aware* extension. Special care should be taken in preprocessing logs where the occurrence of a cancellation omits complete events, see also the example log in Table 7.1.

Integration with the *directly-follows abstraction* variant for very large event logs is possible but tricky. This variant recurses on sub directly-follows graphs instead of sublogs. In our cancellation setting, we can integrate with this variant by annotating directly-follows nodes with triggers from the trigger oracle.

The hierarchical extensions introduced in Chapter 6 only overlap on the cancellation trigger base case. When mixing cancellation and naïve hierarchical

or recursion aware discovery, one should allow cancellation triggers annotations on the named subtree and recursive reference operators, i.e., the existing base cases should be combined to discover *named subtrees with cancellation triggers*  $\star_{\nabla_f}^C$  and *recursive references with cancellation triggers*  $\star_{\Delta_f}^C$ . Table 7.5 presents the combined condition base cases for the hierarchical cancellation discovery algorithm. See also Future Work 7.3 in Section 7.7.

**Table 7.5:** Combined conditions base cases for hierarchical cancellation discovery. Each cell is a base case in the hierarchical cancellation discovery algorithm and shows the returned model. The rows distinguish between the cancellation conditions, the columns distinguish between the recursion aware conditions.

$L \neq \emptyset \wedge \exists f \in \mathbb{A} : \forall \sigma \in L \wedge e \in \sigma : \dots$	$\sigma = \langle f \rangle$	$head(e) = f \wedge (\exists \sigma \in L \wedge e \in \sigma :  e  \geq 2)$ $f \notin C$	$f \in C$
$triggers(f) = \emptyset$	$f$	$\nabla_f(?_{C'})$	$\Delta_f$
$triggers(f) \neq \emptyset$	$\star_f^{triggers(f)}$	$\star_{\nabla_f}^{triggers(f)}(?_{C'})$	$\star_{\Delta_f}^{triggers(f)}$

## 7.6 Evaluation

In this section, we compare the *cancellation discovery (CD)* algorithm against related, implemented techniques. The CD algorithm is implemented in the *Statechart* plugin for the ProM framework, see also Chapter 10. Section 7.6.1 investigates the discovery results on a controlled example. Section 7.6.2 provides a comparison on running time and model quality. For large, real-life case studies and tool UI using the cancellation techniques, see Chapter 12.

### 7.6.1 Evaluation using Synthetic Logs

In this evaluation, we focus on model understandability. We use a small synthetic example software log, mine a model with various discovery algorithms, and compare the resulting models on structure and visual appearance. The goal of this evaluation is to investigate how the CD algorithm compares to existing algorithms in discovering accurate and understandable models when cancellation behavior is present.

#### Methodology

For this evaluation, we revisit the program in Listing 7.1. We executed the program three times: once for the successful execution of `processA()`, once for the successful execution of `processB()`, and once while throwing an exception during `processA()`. During execution, we traced and logged the start and end

of each called method as well as the start of the catch block, resulting in an event log similar to the one shown in Table 7.1. This event log has 3 traces, 31 events, 15 activities (including lifecycles), and 2 hierarchical levels. We used the start of the catch block and associated exception data to instantiate our trigger oracle. As a result, `Main.main()+handle` is marked as the only trigger activity in our trigger oracle.

We use the program in Listing 7.1 and model in Figure 7.1 as a baseline comparison. For the various discovered models, we use the default visualization provided by the various tool implementations, and annotate the activities with their one-letter acronyms, see Table 7.6 for a summary. For the Inductive Miner and our cancellation discovery technique, we used the infrequent (IMf) extensions and indicate the “path” threshold. For “path” threshold, 1.0 means all behavior, 0.8 means 80% of the behavior (i.e., an 80/20 model, describing 80% of the behavior using a simpler 20% model).

For this evaluation, we provide the fitness and precision scores calculated on the Petri net translation, using the alignments-based technique from [21]. Recall from Section 1.1.2 on page 7: *Fitness* expresses the part of behavior in the event log that is also captured in the model. *Precision* expresses the part of the behavior in the model that is also present in the event log. A fitness of 1.0 indicates all behavior in the event log can be reproduced by the model; a lower fitness (minimum 0.0) means the modeled behavior less represents the behavior in the event log. A precision of 1.0 indicates all behavior in the model has been observed in the event log; a lower precision (minimum 0.0) means the modeled behavior is less supported by observations in the event log.

**Table 7.6:** One-letter acronyms for the activities used in the cancellation synthetic evaluation. These acronyms correspond with the baseline model in Figure 7.1 for the program in Listing 7.1.

M	<code>Main.main()</code>	a	<code>Main.processA()</code>	Lifecycle suffixes: +s start +c complete
i	<code>Main.input()</code>	b	<code>Main.processB()</code>	
o	<code>Main.output()</code>	p	<code>Main.prepareResult()</code>	
h	<code>Main.main()+handle</code>	r	<code>Main.recover()</code>	

## Results

Figure 7.4 to 7.16 show the discovered models. Below, we will discuss each model in turn.

**Alpha miner [18]** – The Alpha miner result in Figure 7.4 shows a very disconnected model. Since no alpha-relations were inferred between most activities, most transitions are not connected to places and can fire at any time. This model gives us no information about how the different activities are causally

related. The very low fitness score reflects this lack of information. Hence, this model does not aid in understanding the behavior.

**Fuzzy miner [75]** – The Fuzzy miner result in Figure 7.5 shows a reasonably structured model. This model correctly shows the normal and cancelation branches, but only models it as a choice. In this result, activity *a* has to complete normally before the cancelation signaled by *h* can be triggered. In reality, activity *a* was aborted due to the thrown exception. Furthermore, this model does not generalize, i.e., it does not allow activity *b* to throw an exception and cancel with activity *h*. Using the *significance cutoff* in the Fuzzy miner to cluster behavior reveals the same structures in this model.

**Heuristics miner [192]** – The Heuristics miner result in Figure 7.6 again shows a reasonably structured model. This model has the same overall structure and issues as the *Fuzzy miner* model discussed above. However, this model does recognize the unusual way in which activity *a* is completed/aborted in the cancelation case. Sadly, the split-join semantics around the *a*, *b*, and *o* activities are unclear, leading to soundness issues.

**ILP miner [204]** – The ILP miner result in Figure 7.7 again shows a reasonably structured model. Unfortunately, the ILP prefix filtering removed the branch with activity *b*. In addition, this model has the same issue with the completion of activity *a* as the *Fuzzy miner* model discussed above.

**Genetic miner [10]** – The Genetic miner result in Figure 7.8 shows a confusing model with disconnected parts. Parts of the main flow can be deduced, but the activities *a*, *b*, and *r* are modeled incorrectly. Hence, this model gives an incorrect view on the recorded behavior.

**ETMd miner [46]** – The ETMd miner result in Figure 7.8 shows a misleading picture. For one, this model does not show the choice between activities *a* and *b*. In addition, this model incorrectly allows for a mix between activity *p* and the cancelation path *h* followed by *r*. Hence, this model does not really aid in understanding the behavior.

**TS Regions [16]** – The Transition System miner using Regions result in Figure 7.10 shows a reasonably structured model. This model has the same overall structure and issues as the *Heuristics miner* model discussed above. Observe how, due to the lack of lifecycle information, the fitness score dropped.

**MINT ktails [190]** – The MINT ktails result in Figure 7.12 shows a prefix-based trace model fitted to all three traces. This model is too overfitting and lacks any choice joins. In this result, like with the *Fuzzy miner* result, activity *a* incorrectly has to complete normally before the cancelation signaled by *h* can be triggered.

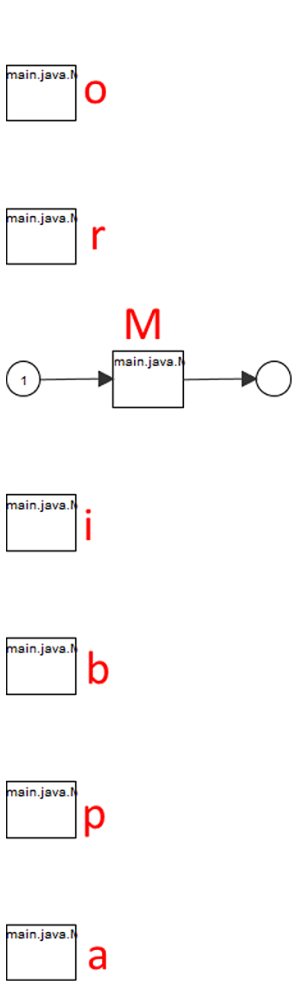
**MINT redblue [190]** – The MINT redblue result in Figure 7.13 shows a slightly inaccurate model. In this model, it is possible to skip activities *i*, *a* and *b*. Hence, this model yields an incorrect understanding of the behavior.

**Synoptic miner [37]** – The Synoptic miner result in Figure 7.11 again

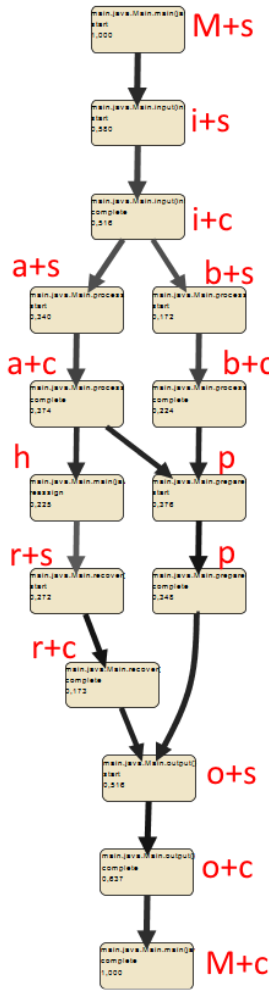
shows a reasonably structured model. This model has the same overall structure and issues as the *Fuzzy miner* model discussed above. Note how due to the lack of lifecycle information (incorrect) self-loops are added to most states.

**Inductive miner [130]** – The Inductive miner result in Figure 7.14 shows a structured but imprecise model. The *path* thresholds indicate the amount of behavior included: 1.0 is all behavior, 0.8 yields an 80/20 model. Ignoring the hierarchy issue involving activity *M*, we do see most of the behavior. Note how, due to a misinterpretation of lifecycle information, activity *h* is hidden in the loop with *M* and the activity *r* is put in a strange choice with the rest of the behavior. The relatively low fitness and precision score reflects these inaccuracies.

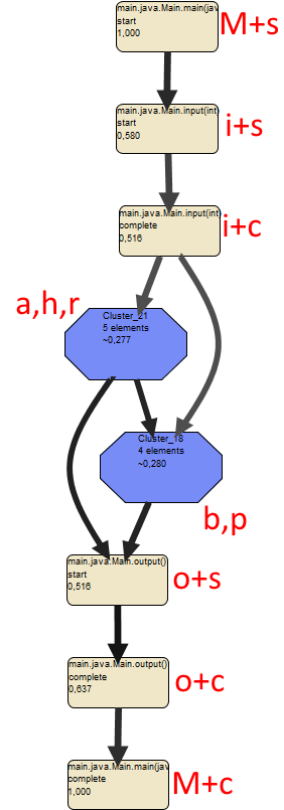
**CD (this chapter)** – The CD algorithm from this chapter yield the model in Figure 7.15 and 7.16. This model correctly identifies the cancellation region and generalized accordingly, thereby reducing the visual complexity. In addition, this model shows how the cancellation discovery algorithm can interplay with the hierarchical discovery algorithms to produce a mixed hierarchical cancellation process tree. Note that this model has perfect fitness and a very high precision. Note that the discovered cancellation process tree in Figure 7.15 matches the baseline model shown in Figure 7.1. In addition, observe how the statechart representation in Figure 7.16 visually separates the good weather from the cancellation behavior.



**Figure 7.4:** The Alpha miner [18] result with fitness 0,88 and precision 0,17. The model is very disconnected and does not aid in understanding the behavior.



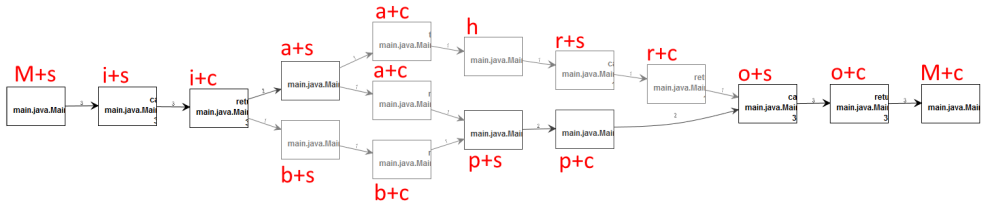
(a) Significance: 0.0



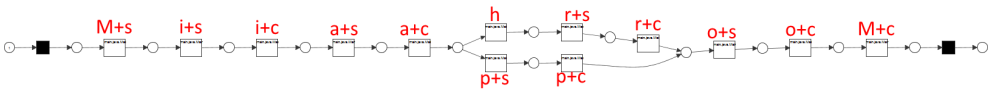
(b) Significance: 0.4

**Figure 7.5:** The Fuzzy miner [75] result. The *significance cutoff* threshold controls how much behavior is clustered (see the octagonal nodes). This model correctly shows the normal and cancelation branches, but only models this as a choice. Furthermore, the model does not generalize.

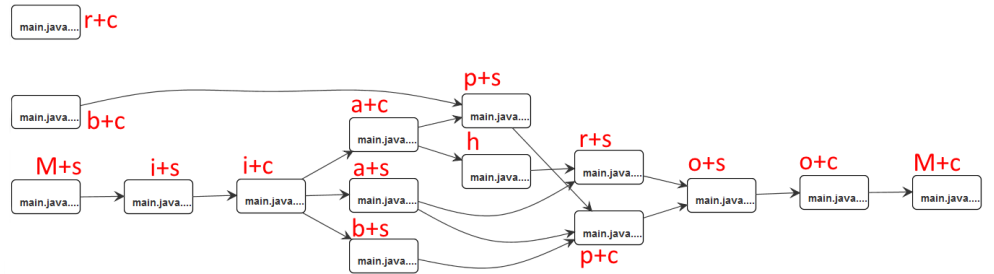




**Figure 7.6:** The Heuristics miner [192] result, model is unsound. This model correctly shows the normal and cancellation branches, and recognizes the unusual way in which activity *a* is completed/aborted in the cancellation case.



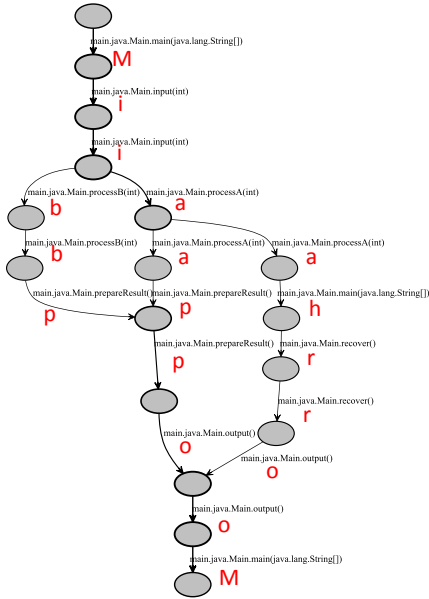
**Figure 7.7:** The ILP miner [204] result with fitness 0,89 and precision 0,29. This model is reasonably structured, but lacks the branch with activity *b* and does not generalize the cancellation behavior.



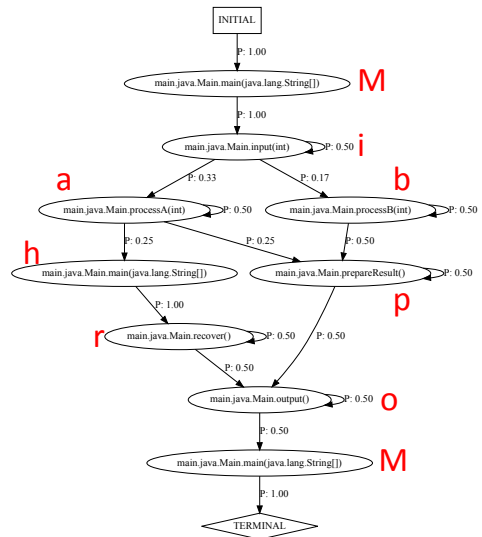
**Figure 7.8:** The Genetic miner [10] result, model is unsound. This confusing model has disconnected parts and gives an incorrect view on the recorded behavior.



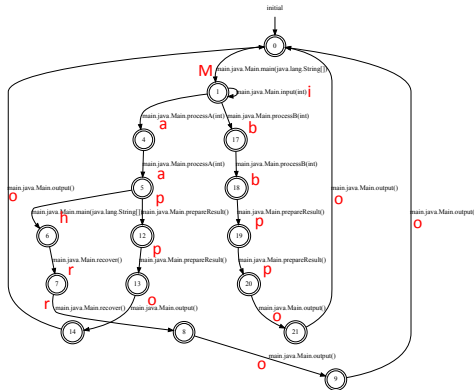
**Figure 7.9:** The ETMd miner [46] result with fitness 0,97 and precision 0,93. This model is misleading: it does not show the choice between activities *a* and *b*, and it incorrectly allows for a mix between activity *p* and the cancellation path *h* followed by *r*.



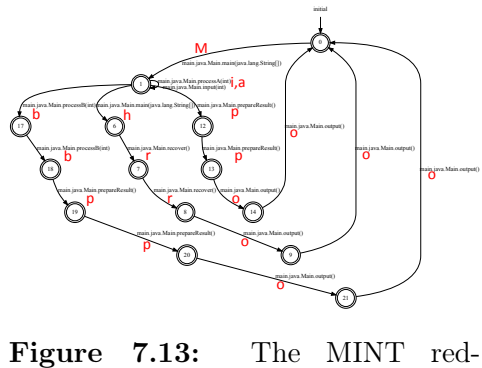
**Figure 7.10:** The Transition System miner using Regions [16] result with fitness 0,59 and precision 0,79. This model correctly shows the normal and cancelation branches, and recognizes the unusual way in which activity *a* is completed/aborted when cancelling.



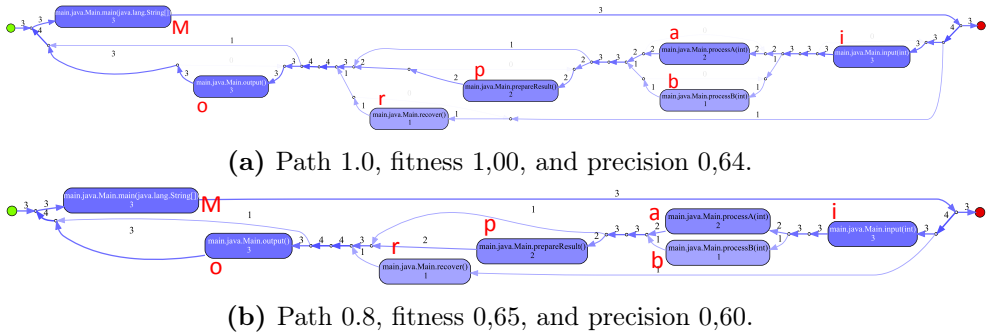
**Figure 7.11:** The Synoptic miner [37] result with fitness 1,00 and precision 0,71. This model correctly shows the normal and cancelation branches, but only models this as a choice. Furthermore, the model does not generalize.



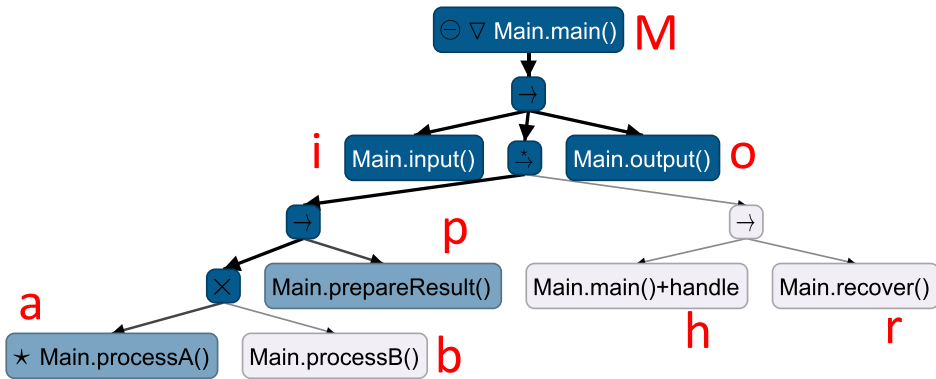
**Figure 7.12:** The MINT ktails [190] result with  $k=1$ , fitness 0,65 and precision 1,00. This model is too overfitting, lacks joins, and does not fully capture the cancelation behavior.



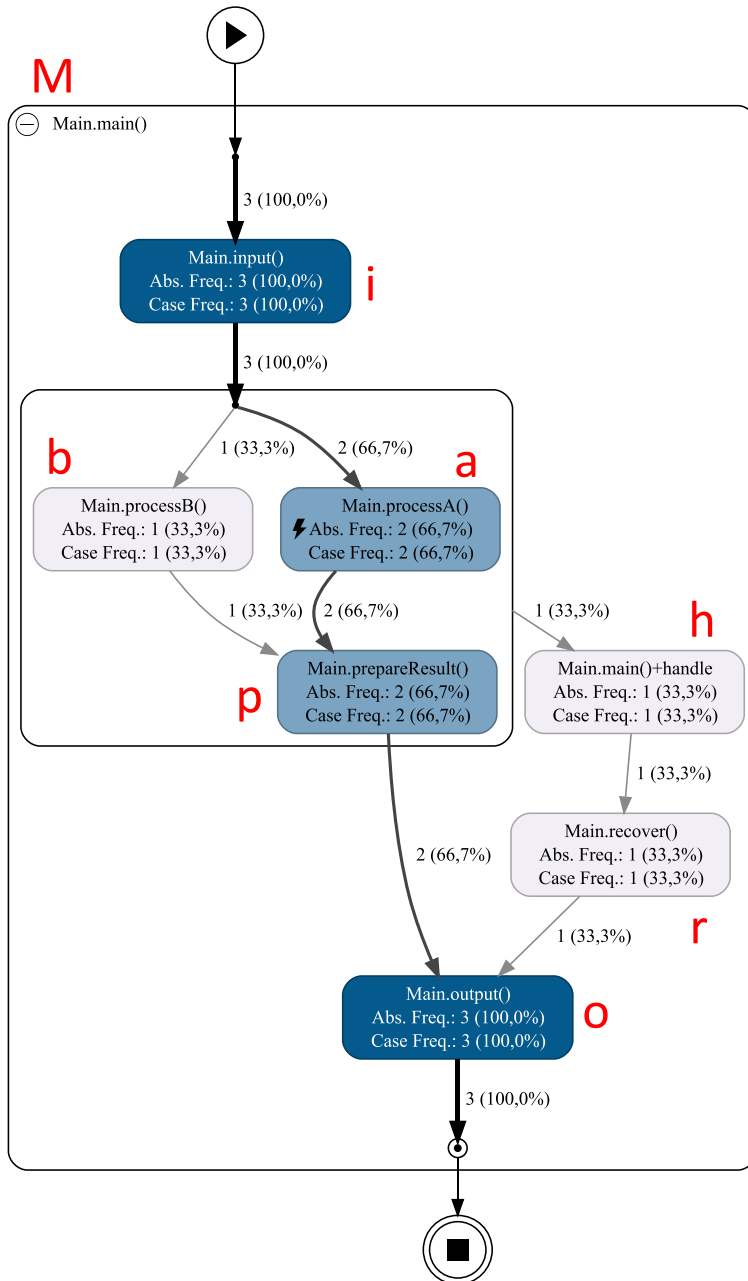
**Figure 7.13:** The MINT red-blue [190] result with  $k=1$ , fitness 0,65, and precision 1,00. This model yields an incorrect understanding of the behavior: it is possible to skip activities *i*, *a* and *b*.



**Figure 7.14:** The Inductive miner [130] result. The *path* thresholds indicate the amount of behavior included: 1.0 is all behavior, 0.8 yields an 80/20 model. Ignoring activity *M*, this models shows most of the behavior (read right to left). However, activity *h* is missing (hidden in activity *M*), and activity *r* is put in a strange choice with the rest of the behavior.



**Figure 7.15:** The Cancellation Discovery (CD) result visualized as a process tree with path 0.8 (amount of behavior included), fitness 1,00, and precision 0,98. This model correctly identifies the cancellation region and generalized accordingly, thereby reducing the visual complexity. In addition, this model shows how the cancellation discovery algorithm can interplay with the hierarchical discovery algorithms to produce a mixed hierarchical cancellation process tree. Note that this model has perfect fitness and a very high precision. In the tool user interface, clicking on activity *a* results in a popup showing how *a* and *h* are related, i.e., it shows the set of corresponding triggers *C* for *a*.



**Figure 7.16:** The Cancellation Discovery (CD) result from Figure 7.15 visualized as a statechart with path 0.8, fitness 1,00, and precision 0,98. The box around activities *a*, *b*, and *p* represents the cancellation region. Observe how the statechart representation visually separates the good weather from the cancellation behavior.

### 7.6.2 Performance and Scalability Evaluation

In this section, we perform a comparative evaluation focusing on the performance and scalability of the various discovery algorithms. The discovery algorithms will be compared on running time and the resulting models will be compared on model quality in terms of fitness and precision.

#### Methodology

All of the algorithms in this comparison are invoked from a Java benchmark setup under the same operating conditions. For these experiments we used a laptop with an i7-4700MQ CPU @ 2.40 GHz, Windows 8.1 and Java SE 1.7.0 67 (64 bit) with 12 GB of allocated RAM.

For the running time, we measured the average running time and associated 95% confidence interval over 30 micro-benchmark executions, after 10 warmup rounds for the Java JVM. Each algorithm is allowed at most 30 seconds for completing a single model discovery. The time for loading event logs or Java classes is excluded from the measurements.

For the model quality, we use fitness and precision calculated on the Petri net translations using the alignments technique from [21, 134] and set a time limit of at most 15 minutes.

#### Event Logs

We selected five event logs for this evaluation, covering a range of input problem sizes. The input problem size is typically measured in terms of four metrics: number of traces, number of events, number of activities (size of the alphabet), and the trace length. The event logs and their input sizes are shown in Table 7.7 and are divided into software and non-software logs.

For the software event logs we used an extended version of the instrumentation tool developed for [119], yielding XES event logs with method-call level and exception catch events. For the NASA CEV software [148], we executed

**Table 7.7:** The event logs used in the performance and scalability evaluation. Shown are input size statistics, indicating the problem sizes of the event logs.

Event Log	# Traces	# Events	# Acts	Trace length		
				Min	Mean	Max
[110] NASA CEV	2	48	17	22	24	26
[112] Alignments	1	17,912	90	17,912	17,912	17,912
[45] WABO	1,434	8,577	27	1	6	25
[63] BPIC 2012, A	13,087	60,849	10	3	5	8
[174] Road Fines, a	150,370	561,470	11	2	4	20
[174] Road Fines, f	150,370	404,009	9	1	3	9

a unit test generated from the source code, covering all of the code branches. The resulting *NASA CEV* [110] event log is filtered to describe two executions (test cases 1 and 10) of a software process with errors. For the alignments software [21, 187], we executed an alignment computation on a typical input log and model. We feed the algorithm an unsound model to trigger a software error.

The *WABO* [45] event log describes the receipt phase of an environmental permit application process (‘WABO’) at a Dutch municipality. The *BPIC12* [63] event log is a BPI challenge log that describes three subprocesses of a loan application process. In this evaluation, we only focus on the “A\_” subprocess. The *Road fine* [60] event log was obtained from an information system managing road traffic fines. We use two variants of this large event log. The *Road fine*, a variant is the largest, most complex event log in our experimental setup. In variant *Road fine*, f, we filtered out the asynchronous activities “no payment” and “add penalty” to decrease the (directly-follows) complexity.

In Table 7.8, we have listed the trigger oracles we used in our cancellation discovery techniques for the above event logs.

**Table 7.8:** The trigger oracles used for the event logs in the cancellation evaluation.

Event Log	Trigger Oracle Activities
[110] NASA CEV	“cev.ErrorLog.last()”
[112] Alignments	Exception detection based on catch-block handle events
[45] WABO	“T15 Print document X request unlicensed”, “T16 Report reasons to hold request”
[63] BPIC12, A	“A_ CANCELLED”, “A_ DECLINED”
[60] Road fine, a	“Send for Credit Collection”
[60] Road fine, f	“Send for Credit Collection”

### Results – Running Time

In Tables 7.9 and 7.10, the results for the runtime benchmark are given for the software and non-software logs respectively.

The first thing we notice is that, in contrast to the TS Cancel technique, our Cancellation algorithm always discovers a model within the allotted time, unlike some other techniques (e.g., [97]). When compared to the baseline Inductive Miner, there seems to be a small overhead in running time. There are two explanations for this small overhead. One is the fact that more tree operator cuts have to be checked at each recursive call of the algorithm. But more importantly, the new cancellation operators potentially uncover more structures in the directly follows graph. In cases where the original Inductive Miner might

give up and falls back to loops with skips and/or flower models, we can find a cancellation pattern, and recurse on a more structured subproblem. The end result is that we have more recursive calls to uncover all the structures/tree operators, and hence have a larger running time. Nevertheless, our technique successfully scales to larger logs and consistently yields results within seconds.

### Results – Model Quality

In Tables 7.11 and 7.12, the results of the model quality measurements are given. The Fuzzy miner is absent due to the lack of semantics for fuzzy models.

Observe that, compared to the original Inductive Miner, our Cancellation algorithm always yields an equal or more fitting model. Moreover, we preserve the perfect fitness guarantee of the original Inductive Miner (for *path* 1.0). In addition, in most cases, the resulting model is also more precise. Based on these results, we find two properties when using cancellation discovery with an appropriate trigger oracle. For one, *the models produced do not degrade in quality when cancellation behavior cannot be (correctly) uncovered*. And second, *when cancellation behavior is present, the model quality is improved*.

In all cases, we can see that we outperform the ILP algorithms on precision, and we outperform the ETMd miner and TS based miners on fitness. Observe that, in specific cases, the MINT and Synoptic model quality equals the model quality of our cancellation models. However, as discussed in Section 7.6.1, such models can be too overfitting. In addition, as can be seen in the running times in this evaluation and the evaluation in Section 6.7.2, these techniques do not scale well for larger problems. Overall, we can conclude that the added expressiveness of modeling the cancellation region have a positive impact on the model quality.

### On the Simplicity of Models

Finally, we manually compared the discovered models based on the number of visual elements such as nodes, arcs, and transitions. In most cases, the discovered cancellation models are comparable in terms of its complexity with the baseline non-cancellation models, where the cancellation models are usually being slightly simpler. For example, in Figure 7.17, two discovered models for the BPIC12 log at paths 0.8 are shown. Note that in the cancellation model (Figure 7.17b), we see that the main, happy flow behavior is neatly discovered inside the cancellation region, and the “negative” behavior is modeled separately after triggering the cancellation region. In the IM model (Figure 7.17a), skips obfuscate the normal happy flow behavior.

Overall, we can conclude that the added expressiveness of modeling the cancellation region has, in most cases, a positive impact on the model simplicity.

**Table 7.9:** Comparison of algorithm running times on software event logs with cancelation. Given are the average running times in milliseconds over 30 runs, with a 95% confidence interval shown in the bar plots. Note that the plots use a logarithmic scale. The paths column indicates the value for the IMF infrequent threshold: 1.0 means all behavior, 0.8 means 80% of the behavior (i.e., an 80/20 model), 0.5 means 50% of the behavior.

Algorithm	Paths	NASA CEV	Alignments
[18] Alpha miner		0.3	15.6
[192] Heuristics		2.3	444.4
[75] Fuzzy miner		5.9	<sup>T</sup>
[193] ILP miner		371.0	<sup>T</sup>
[204] ILP, filtering		381.9	<sup>T</sup>
[10] Genetic miner		3498.1	21945.3
[46] ETMd miner		27836.5	<sup>T</sup>
[16] TS Regions		18.9	<sup>T</sup>
[97] TS Cancel		<sup>T</sup>	<sup>T</sup>
[190] MINT, redblue	k=1	8.7	<sup>S</sup>
[190] MINT, redblue	k=2	6.0	<sup>S</sup>
[190] MINT, redblue	k=3	5.6	<sup>S</sup>
[190] MINT, ktails	k=1	3.0	<sup>S</sup>
[190] MINT, ktails	k=2	2.4	<sup>S</sup>
[190] MINT, ktails	k=3	1.8	<sup>S</sup>
[37] Synoptic		87.2	<sup>T</sup>
[130] IM (baseline)	1.0	1.4	4560.6
[130] IM (baseline)	0.8	0.9	3995.8
[130] IM (baseline)	0.5	0.9	2602.7
Sec. 7.4	Cancelation	1.0	457.3
	Cancelation	0.8	362.1
	Cancelation	0.5	181.2

Avg. runtime (in milliseconds) with 95% conf. int.

<sup>S</sup> Stack overflow

<sup>T</sup> Time limit exceeded (30 sec.)





**Table 7.10:** Comparison of algorithm running times on non-software event logs with cancellation. Given are the average running times in milliseconds over 30 runs, with a 95% confidence interval shown in the bar plots. Note that the plots use a logarithmic scale. The paths column indicates the value for the IMF infrequent threshold: 1.0 means all behavior, 0.8 means 80% of the behavior (i.e., an 80/20 model), 0.5 means 50% of the behavior.

Algorithm	Paths	WABO	BPIC 2012, A	Road fine, a	Road fine, f	
[18] Alpha miner		8.8	18.7	467.8	260.	
[192] Heuristics		49.2	162.8	1641.7	1047.7	
[75] Fuzzy miner		159.9	343.6	4292.3	2829.1	
[193] ILP miner		233.9	501.5	3426.1	2517.8	
[204] ILP, filtering		236.3	497.2	3395.1	2583.1	
[10] Genetic miner		26029.7	3402.9	– <sup>T</sup>	25592.0	
[46] ETMd miner		– <sup>T</sup>	27130.5	28722.1	27557.6	
[16] TS Regions		– <sup>T</sup>	555.1	5089.5	3455.2	
[97] TS Cancel		– <sup>T</sup>	139.6	– <sup>T</sup>	– <sup>T</sup>	
[190] MINT, redblue	k=1	302.1	76.4	5470.2	917.5	
[190] MINT, redblue	k=2	276.8	78.3	5273.8	891.1	
[190] MINT, redblue	k=3	325.0	71.0	5608.3	909.5	
[190] MINT, ktails	k=1	76.4	56.8	885.6	388.2	
[190] MINT, ktails	k=2	138.2	60.4	972.8	375.7	
[190] MINT, ktails	k=3	160.1	70.7	835.6	379.2	
[37] Synoptic		– <sup>T</sup>	16587.8	– <sup>T</sup>	– <sup>T</sup>	
[130] IM (baseline)	1.0	120.1	308.0	5668.9	2859.0	
[130] IM (baseline)	0.8	138.1	300.7	4049.5	2540.2	
[130] IM (baseline)	0.5	135.4	301.4	4132.9	2537.1	
Sec. 7.4	Cancellation	1.0	176.7	373.2	5972.5	3047.9
	Cancellation	0.8	156.4	379.8	6145.4	3289.3
	Cancellation	0.5	154.1	377.7	6180.8	3558.8

Avg. runtime (in milliseconds) with 95% conf. int.

<sup>T</sup> Time limit exceeded (30 sec.)

**Table 7.11:** Comparison of model quality scores on software event logs with cancelation. Given are the fitness and precision values for the discovered models. These values range from 0.0 to 1.0, higher is better.

Algorithm	Paths	NASA CEV		Alignments	
		Fitness	Precision	Fitness	Precision
[18] Alpha miner		0.89	0.08 <sup>U</sup>	1.00	0.01
[192] Heuristics		<sup>U</sup>	<sup>U</sup>	<sup>U</sup>	<sup>U</sup>
[193] ILP miner		1.00	0.33	n/a	n/a
[204] ILP, filtering		1.00	0.33	n/a	n/a
[10] Genetic miner		<sup>U</sup>	<sup>U</sup>	<sup>U</sup>	<sup>U</sup>
[46] ETMd miner		0.84	0.79	n/a	n/a
[16] TS Regions		0.27	0.61	n/a	n/a
[97] TS Cancel		n/a	n/a	n/a	
[190] MINT, redblue	k=1	0.70	0.58	n/a	n/a
[190] MINT, redblue	k=2	0.69	0.64	n/a	n/a
[190] MINT, redblue	k=3	0.69	0.64	n/a	n/a
[190] MINT, ktails	k=1	0.70	0.80	n/a	n/a
[190] MINT, ktails	k=2	0.70	0.80	n/a	n/a
[190] MINT, ktails	k=3	0.66	1.00	n/a	n/a
[37] Synoptic		1.00	0.74	n/a	n/a
[130] IM (baseline)	1.0	1.00	0.69	<sup>T</sup>	<sup>T</sup>
[130] IM (baseline)	0.8	0.74	0.73	<sup>T</sup>	<sup>T</sup>
[130] IM (baseline)	0.5	0.62	0.75	<sup>T</sup>	<sup>T</sup>
Sec. 7.4	Cancelation	1.0	0.70	<sup>T</sup>	<sup>T</sup>
	Cancelation	0.8	0.76	<sup>T</sup>	<sup>T</sup>
	Cancelation	0.5	0.64	0.67	<sup>T</sup>

<sup>T</sup> Time limit exceeded (15 min.)

<sup>U</sup> Unsound model

<sup>R</sup> Not reliable (fitness = 0)

n/a No model (see Table 6.12)

**Table 7.12:** Comparison of algorithm running times on non-software event logs with cancellation. Given are the fitness and precision values for the discovered models. These values range from 0.0 to 1.0, higher is better.

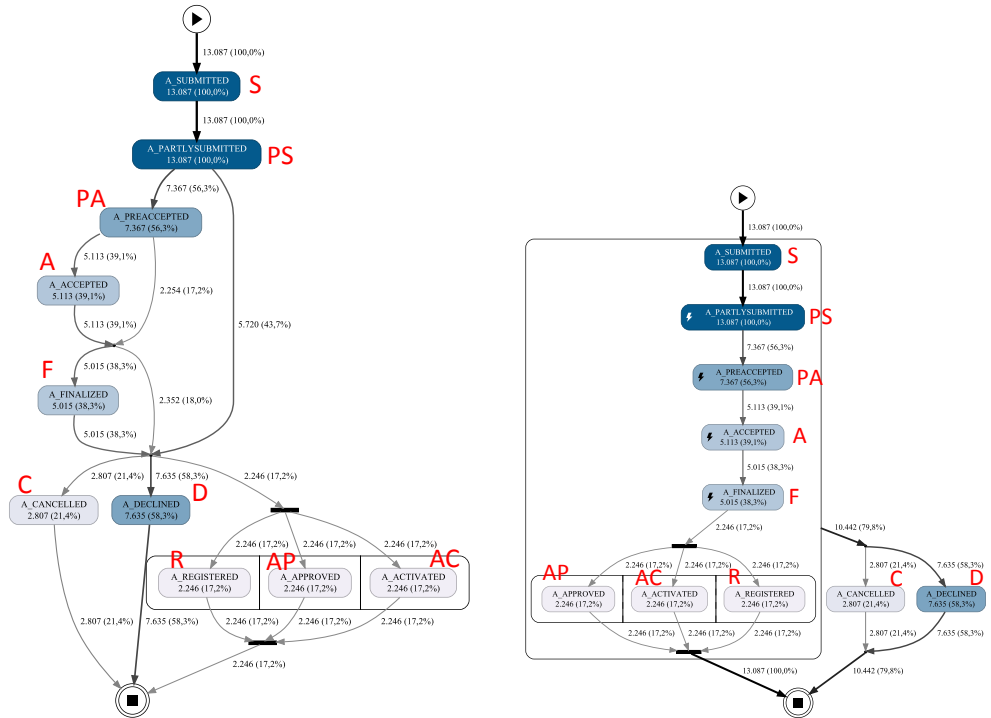
Algorithm	Paths	WABO		BPIC 2012, A		Road fine, a		Road fine, f		
		Fitness	Precision	Fitness	Precision	Fitness	Precision	Fitness	Precision	
[18] Alpha miner		–U	–U	–U	–U	–U	–U	–U	–U	
[192] Heuristics		0.61 █	0.98 █	–U	–U	–U	–U	0.74 █	1.00 █	
[193] ILP miner		1.00 █	0.12 ↓	1.00 █	0.22 ↓	1.00 █	0.50 █	1.00 █	0.53 █	
[204] ILP, filtering		0.97 █	0.35 █	1.00 █	0.28 █	0.78 █	1.00 █	0.81 █	1.00 █	
[10] Genetic miner		–U	–U	–U	–U	n/a	n/a	–U	–U	
[46] ETMd miner		–U	–U	1.00 █	0.86 █	0.79 █	1.00 █	1.00 █	0.41 █	
[16] TS Regions		–U	–U	0.93 █	0.88 █	0.86 █	0.76 █	0.76 █	0.82 █	
[97] TS Cancel		n/a	n/a	0.91 █	0.78 █	n/a	n/a	n/a	n/a	
[190] MINT, redblue	k=1	0.73 █	0.55 █	0.98 █	0.44 █	0.11 ↓	0.32 █	0.12 ↓	0.48 █	
[190] MINT, redblue	k=2	0.67 █	0.55 █	0.98 █	0.44 █	0.11 ↓	0.32 █	0.46 █	0.37 █	
[190] MINT, redblue	k=3	0.67 █	0.56 █	0.98 █	0.86 █	0.11 ↓	0.32 █	0.46 █	0.37 █	
[190] MINT, ktails	k=1	0.00	–R	1.00 █	1.00 █	0.15 ↓	0.76 █	0.00	–R	
[190] MINT, ktails	k=2	0.00	–R	1.00 █	1.00 █	0.15 ↓	0.76 █	0.50 █	0.45 █	
[190] MINT, ktails	k=3	0.00	–R	1.00 █	1.00 █	0.15 ↓	0.85 █	0.50 █	0.36 █	
[37] Synoptic		n/a	n/a	1.00 █	1.00 █	n/a	n/a	n/a	n/a	
[130] IM (baseline)	1.0	1.00 █	0.43 █	1.00 █	0.89 █	1.00 █	0.69 █	1.00 █	0.83 █	
[130] IM (baseline)	0.8	0.94 █	0.64 █	1.00 █	0.92 █	0.99 █	0.48 █	1.00 █	0.82 █	
[130] IM (baseline)	0.5	0.94 █	0.63 █	0.82 █	1.00 █	0.76 █	0.48 █	0.74 █	0.77 █	
Sec. 7.4	Cancelation	1.0	1.00 █	0.62 █	1.00 █	1.00 █	1.00 █	0.66 █	1.00 █	0.76 █
	Cancelation	0.8	0.94 █	0.67 █	1.00 █	1.00 █	1.00 █	0.35 █	1.00 █	0.68 █
	Cancelation	0.5	0.94 █	0.66 █	1.00 █	1.00 █	0.90 █	0.39 █	0.81 █	0.73 █

<sup>T</sup> Time limit exceeded (15 min.)

<sup>R</sup> Not reliable (fitness = 0)

<sup>U</sup> Unsound model

<sup>n/a</sup> No model (see Table 6.12)



(a) IM (baseline) model result, no cancellation. Skips obfuscate the happy flow. (b) Cancel model result, with cancellation region. Happy flow in cancel region.

**Figure 7.17:** The Inductive Miner (IM) and Cancelation Discovery (CD) result for the BPIC12 event log, visualized as a statechart. The encapsulating box represents the cancellation region.

**Legend:** S) A\_Submitted, PS) A\_PartlySubmitted, PA) A\_PreAccepted, A) A\_Accepted, F) A\_Finalized, C) A\_Cancelled, D) A\_Declined, R) A\_Registered, AP) A\_Approved, AC) A\_Activated

## 7.7 Conclusion and Open Challenges

In this chapter, we introduced a modeling notation and discovery technique for cancellation behavior (Contribution 3). With the *cancellation process tree* we provided extensions to capture sequential and loop-back based cancellation regions. With a *trigger oracle* we made the start of cancellation behavior via so-called trigger activities explicit in the input. The *cancellation discovery* algorithm discovers multiple, possibly nested cancellation regions from event logs using a trigger oracle and captures this behavior in cancellation process trees. This discovery algorithm allows us to analyze software processes and other processes containing cancellation behavior such as exceptions and error handling. Moreover, the proposed algorithm offers good discovery guarantees and scales well. In addition, we discussed how cancellation discovery can be combined with the hierarchical discovery solutions from Chapter 6.

With the cancellation solutions presented in this chapter, there are several interesting directions for future research.

■ **Future Work 7.1 — Trigger Oracle Updates during Discovery.** Consider step 2 in Table 7.4, where a sequence cancellation is discovered. If the empty trigger oracle would be used instead, none of the edges would be trigger edges, no valid cut would be detected, and a fallback solution would be used (see also Section 7.4.4). In these type of cases, one could use the directly-follows graph to derive a trigger oracle. In such an approach, when no cut can be found, a trigger oracle hypothesis can be constructed and tested. Alternatively, entropy-based heuristics can check for potential trigger activities before a cut is detected, optimizing for a priori cancellation detection. When the hypothesis oracle allows a cancellation cut to be detected, the oracle is updated for the remaining discovery process. It should be noted that there are many edge cases to be investigated in such a setup. For example, what should happen when the updated trigger oracle invalidates previously discovered tree operators? Recall that, for the normal tree operators, no trigger edges are allowed between cut partitions. Due to the prefix-based semantics, such trigger oracle updates are far from trivial.

■ **Future Work 7.2 — Efficient Optimization Strategy for Trigger Oracle Estimation.** Instead of updating trigger oracles during discovery, one can implement an optimization strategy that selects trigger oracle hypotheses before discovery. As discussed on page 174, the intuition is that, if cancellation behavior is present, modeling this behavior with cancellation operators will yield a more fitting and possibly more precise process tree. However, there are potentially many trigger oracle hypotheses to check, and each check entails discovering a complete model and computing fitness and precision. Hence, an efficient optimization strategy minimizes the amount of candidate hypotheses generated.

On the one hand, smart heuristics (like the directly-follows based approach discussed above) should be used to generate good initial candidates. On the other hand, inclusion properties for oracle triggers should be investigated to optimize the construction of better trigger oracles.

■ **Future Work 7.3 — Hierarchical Cancellation Discovery.** As noted in Section 7.5, the hierarchical extensions from Chapter 6 can be mixed with the cancellation extensions from this chapter. Table 7.5 already presented the combined conditions for the corresponding base cases. However, more research is needed to successfully combine the prefix semantics with named submodels. For example, how should cancellations be propagated across hierarchies? What are the semantics of a named subtree with cancellation triggers? Does this mean that every activity in a named subtree has the corresponding cancellation triggers? And what are the semantics of a recursive reference with cancellation triggers?

■ **Future Work 7.4 — Cancellation Visualization Layout.** As noted in Section 7.1, cancellation features can separate good weather behavior from error/cancellation behavior. Due to the structured, hierarchical nature of cancellation process trees, we can easily identify and localize this separation of good weather behavior and error/cancellation behavior. Hence, a visualization layout algorithm can use this information to aid the user in showing how these types of behavior are separated. Such a layout algorithm should use the tree structure to clearly and deterministically position the normal and mainstream behavior while at the same time clearly indicate where alternative cancellation behavior can occur in the model.



# III | Beyond Model Discovery

<b>8</b>	<b>Hierarchical Performance Analysis</b> . . . . .	<b>209</b>
8.1	Why We Need Hierarchical Performance Analysis	
8.2	Introduction to Alignments	
8.3	Analysis Framework Definition	
8.4	Metrics for Execution Intervals	
8.5	Evaluation	
8.6	Conclusion and Open Challenges	
<b>9</b>	<b>Translations and Traceability</b> . . . . .	<b>259</b>
9.1	The Translations and Traceability Framework	
9.2	Basic and Extended Petri net Interpretations	
9.3	Model to Model Translations	
9.4	Conclusion and Open Challenges	



---

## I Introduction

Chapter 1  
Overview

Chapter 2  
Preliminaries

Chapter 3  
Related Work

Chapter 4  
A Process Mining  
Foundation

---

## II Hierarchical Process Discovery

Chapter 5  
On Software Data  
and Behavior

Chapter 6  
Hierarchical and Recursion  
Aware Discovery

Chapter 7  
Cancellation Discovery

---

## III Beyond Model Discovery

Chapter 8  
Hierarchical Performance  
Analysis

Chapter 9  
Translations  
and Tracability

---

## IV Applications

Chapter 10  
Tool Implementations

Chapter 11  
The Software Process  
Analysis Methodology

Chapter 12  
Case Studies

---

## V Closure

Chapter 13  
Conclusion

Appendix A  
Proofs

In Part III, we further explore hierarchical process mining beyond model discovery.

**Chapter 8** introduces a hierarchical approach to performance analysis, taking into account hierarchical, recursive, and cancellation behavior.

**Chapter 9** provides an extensive model to model transformation framework, taking into account the hierarchical, recursive, and cancellation semantics.

*“Where’s the sense in promising  
to achieve the achievable?”*  
— Terry Pratchett, *Going Postal*

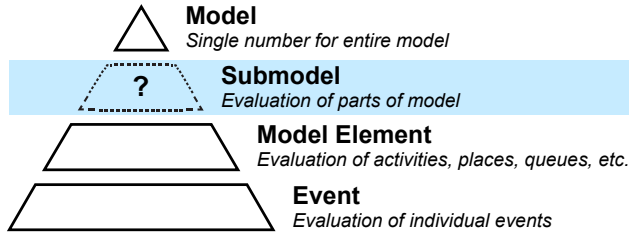
## 8 | Hierarchical Performance Analysis

In this chapter, we introduce a framework and novel formalization for hierarchical performance analysis. Our approach takes into account previously unaddressed notions such as subprocesses and cancelation behavior (Contribution 4). We start by motivating the need for hierarchical performance analysis in Section 8.1. After that, we give a brief introduction to our analysis foundation, Alignments, in Section 8.2. Based on this foundation, we introduce our extended analysis framework in Section 8.3 and we propose a novel formalization of existing and novel performance metrics in Section 8.4. Finally, Section 8.5 will evaluate the introduced analysis approach and Section 8.6 will conclude this chapter.

### 8.1 Why We Need Hierarchical Performance Analysis

In recent years, the process mining field made huge advances in terms of scalability. Both process discovery [3, 4, 133, 186] and conformance checking [3, 4, 133, 145] have become better at handling and analyzing larger and more complex processes. Advances in preprocessing and process discovery enabled the application of process mining in complex settings such as relational databases [55, 69], distributed systems, and software systems. In addition, the work in Chapters 6 and 7 supports advanced process model constructs such as *subprocesses*, *recursive subprocess definitions* and *cancelation* (e.g., exception patterns), and the work in [130] supports *various notions of concurrency*. One has to realize that a simple, small, and flat process model will not suffice anymore, especially when applied to analyzing software system processes. However, state of the art performance analysis is still typically performed either over the whole process model or at the level of an individual model element or event, e.g., a process step, a place or queue, a software method or statement, etc. Hence, there is a need for hierarchical performance analysis, taking into account the notions of submodel abstractions and model execution semantics, see Figure 8.1.

In this chapter, we 1) *present a framework for establishing precise relationships between events and submodels, taking into account execution semantics;*

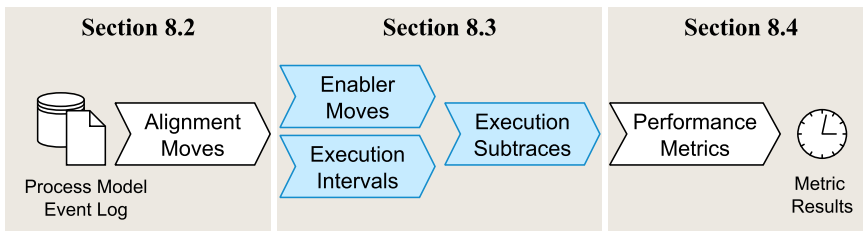


**Figure 8.1:** Our performance analysis approach positioned in the context of the level at which metrics are computed.

and 2) present a novel formalization of existing and novel performance metrics. We will use existing work on alignments [21] as a foundation for our approach, which we will briefly introduce in Section 8.2. Using this foundation, we incorporate execution semantics via so-called *enabler moves* and *execution intervals*, and project aligned event logs onto *execution subtraces*. We use Petri net semantics to define and realize our approach, but the ideas are independent of this. Using the semantic-aware execution subtraces, we compute our metrics at various abstraction levels. The approach is outlined in Figure 8.2. In Chapter 9, we will show how the results can be mapped between Petri nets, BPMN models, statecharts, message sequence diagrams, and extended process trees.

Our approach is especially useful for analyzing software system processes, as we will show in the evaluation in Section 8.5, but is generic enough to apply to any kind of operational process, including business processes. This approach enables performance analysis on a range of abstraction levels; submodels can be identified and analyzed in isolation and in the context of the rest of the process model. These submodels can arise from all kinds of (semantical) structures in process models: subprocesses [55, 94], (Chapter 6), control-flow structures [97, 130], (Chapter 7), graph-based structures [145, 158], abstraction patterns [93, 94], business instance object relations [55, 69], activity semantics, etc.

We assume that all start and complete events are present in an event log, with appropriate timestamp data available. This is not a limitation, as in



**Figure 8.2:** Outline of the hierarchical performance analysis approach.

---

**Listing 8.1** Running example Java code illustrating concurrent behavior. Upon execution, this program is logged at the method level, excluding library calls.

---

```
1  import java.util.concurrent.ExecutorService;
2  import java.util.concurrent.Future;
3
4  public class Main {
5      // Entry point
6      public static void main(int i) {
7          setup();
8          boolean done = false;
9          while(!done) {
10             done = read_input();
11             if (!done) { calculate(); }
12         }
13         report();
14     }
15
16     public static void calculate() {
17         // Run two computations concurrently (multi-threaded), i.e.:
18         //   parallel compute_f1()
19         //   parallel compute_f2()
20         ExecutorService executor = Executors.newFixedThreadPool(2);
21         Future<Void> task1 = executor.submit(this::compute_f1);
22         Future<Void> task2 = executor.submit(this::compute_f2);
23
24         // synchronize and wait until both tasks are completed
25         task1.get();
26         task2.get();
27     }
28
29     private static void setup() { ... }
30     private static boolean read_input() { ... }
31     private static void report() { ... }
32     private static Void compute_f1() { ... }
33     private static Void compute_f2() { ... }
34 }
```

many cases information can be completed from the rest of the event log and/or model where needed [21]. Various approaches have been proposed in literature to address the lack of data. For example, the work in [22] assumes that missing events or timestamps indicate instantaneous activities. Alternatively, the work in [147] estimates missing timestamps based on the notion of resource availability [147].

As a running example, consider the program in Listing 8.1. During the `main()` function, we start with `setup()`, followed by looping over `read_input()` and `calculate()`, and we finish with `report()`. In `compute()`, we execute `compute_f1()` and

**Table 8.1:** Example snippet of an event log for the program in Listing 8.1. Note that, for the examples, case 3 intentionally captures deviating behavior.

Case id	Event id	Attributes					
		Activity	Lifecycle	Timestamp	Resource	...	
1	1.1	<i>m</i>	main()	start	11:02:44.900	thread-1	...
	1.2	<i>s</i>	setup()	start	11:02:45.000	thread-1	...
	1.3	<i>s</i>	setup()	complete	11:02:45.230	thread-1	...
	1.4	<i>i</i>	read_input()	start	11:02:45.250	thread-1	...
	1.5	<i>i</i>	read_input()	complete	11:02:48.010	thread-1	...
	1.6	<i>r</i>	report()	start	11:02:48.120	thread-1	...
	1.7	<i>r</i>	report()	complete	11:02:49.000	thread-1	...
	1.8	<i>m</i>	main()	complete	11:02:49.100	thread-1	...
2	2.1	<i>m</i>	main()	start	11:06:01.900	thread-1	...
	2.2	<i>s</i>	setup()	start	11:06:02.000	thread-1	...
	2.3	<i>s</i>	setup()	complete	11:06:02.470	thread-1	...
	2.4	<i>i</i>	read_input()	start	11:06:02.510	thread-1	...
	2.5	<i>i</i>	read_input()	complete	11:06:02.930	thread-1	...
	2.6	<i>c</i>	calculate()	start	11:06:03.110	thread-1	...
	2.7	<i>f1</i>	compute_f1()	start	11:06:03.320	thread-2	...
	2.8	<i>f2</i>	compute_f2()	start	11:06:03.340	thread-3	...
	2.9	<i>f1</i>	compute_f1()	complete	11:06:03.850	thread-2	...
	2.10	<i>f2</i>	compute_f2()	complete	11:06:03.900	thread-3	...
	2.11	<i>c</i>	calculate()	complete	11:06:04.070	thread-1	...
	2.12	<i>i</i>	read_input()	start	11:06:04.160	thread-1	...
	2.13	<i>i</i>	read_input()	complete	11:06:04.770	thread-1	...
	2.14	<i>c</i>	calculate()	start	11:06:05.000	thread-1	...
	2.15	<i>f1</i>	compute_f1()	start	11:06:05.100	thread-2	...
	2.16	<i>f1</i>	compute_f1()	complete	11:06:05.210	thread-2	...
	2.17	<i>f2</i>	compute_f2()	start	11:06:05.280	thread-2	...
	2.18	<i>f2</i>	compute_f2()	complete	11:06:05.340	thread-2	...
	2.19	<i>c</i>	calculate()	complete	11:06:05.510	thread-1	...
	2.20	<i>i</i>	read_input()	start	11:06:05.600	thread-1	...
	2.21	<i>i</i>	read_input()	complete	11:06:05.670	thread-1	...
	1.22	<i>r</i>	report()	start	11:06:05.800	thread-1	...
	1.23	<i>r</i>	report()	complete	11:06:05.850	thread-1	...
	1.24	<i>m</i>	main()	complete	11:06:05.950	thread-1	...
3	3.1	<i>m</i>	main()	start	11:09:42.000	thread-1	...
	3.2	<i>s</i>	setup()	start	11:09:42.050	thread-1	...
	3.3	<i>s</i>	setup()	complete	11:09:45.230	thread-1	...
	3.4	<i>c</i>	calculate()	start	11:09:47.110	thread-1	...
	3.5	<i>c</i>	calculate()	complete	11:09:48.010	thread-1	...
	3.6	<i>r</i>	report()	start	11:09:48.120	thread-1	...
	3.7	<i>r</i>	report()	complete	11:09:49.000	thread-1	...
	3.8	<i>m</i>	main()	complete	11:09:49.050	thread-1	...
4	4.1	<i>m</i>	main()	start	11:12:07.020	thread-1	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	

`compute_f2()` in parallel using multi-threading. Table 8.1 shows a corresponding execution event log and Figure 8.3 shows a corresponding process model. Using the hierarchical performance analysis introduced in this chapter, we will show, amongst others, how we can investigate the efficiency of this multi-threading.

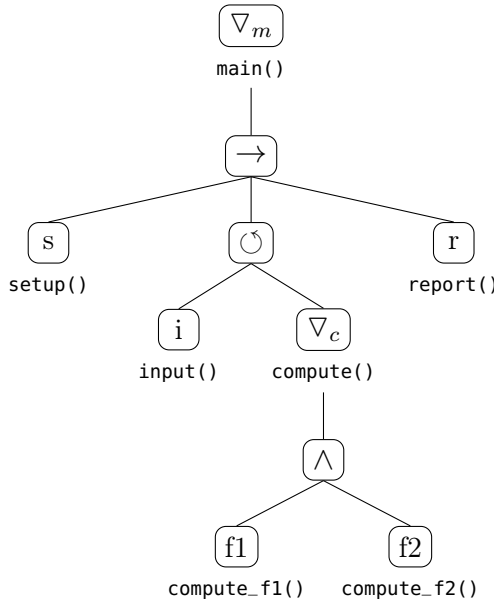
## 8.2 Introduction to Alignments

For performance analysis, we assume that a process model and an event log are given. The model may have been discovered through process discovery or handmade. One of the main challenges is to find the best way in which observed behavior in an event log can be replayed on a process model. That is, given an event log and a Petri net, to find a valid mapping between events in the log to transition executions in the net. In a situation where the net is relatively simple and only allows for the behavior observed in the event log and vice versa, mapping events to transitions is trivial. Problems arise when the observed behavior in the log is not following the same behavior as the behavior allowed by the net. Another issue is that a Petri net may have invisible transitions (which are not recorded in event logs) and multiple transitions with the same label, i.e., duplicate transitions [21]. In the rich hierarchical models discovered by the approaches in Chapters 6 and 7, all of these elements occur. Moreover, when discovering an 80/20 model using the infrequent discovery extensions, the behavior in the event log does not necessarily perfectly fit the model.

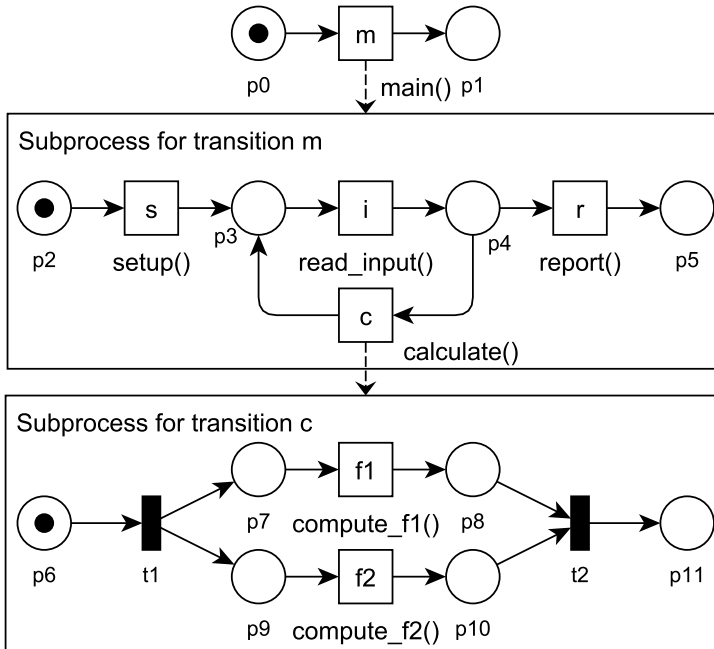
The work on alignments [21] provides a robust foundation for aligning observed behavior in the log to a Petri net model and is, at the time of writing, considered the state-of-the-art solution for reliable and robust alignment results. The produced alignments provide a robust mapping between events in the log and transitions in the model, even in the presence of non-fitting behavior, invisible transitions, and duplicate transitions in the Petri net. Furthermore, the alignment approach shows explicitly where, when, and why deviations occur, thus providing a basis for further analysis.

### 8.2.1 Unfolding Hierarchical Models

Although the work on alignments [21] provides a robust foundation for aligning observed behavior to a Petri net, an alignment can only be computed over a flat Petri net. Even though it should be feasible to adapt the alignments algorithm to the type of hierarchical models discovered by the techniques in this thesis (see Future Work 6.6 on page 161), we consider this out of scope for now. Instead, we will rely on a simple reduction technique to flatten hierarchical models to traditional Petri nets. We will use these flat Petri nets for our internal computations only. For the end result presented to the user, we can trace the results back to the original hierarchy-aware representations and visualizations.

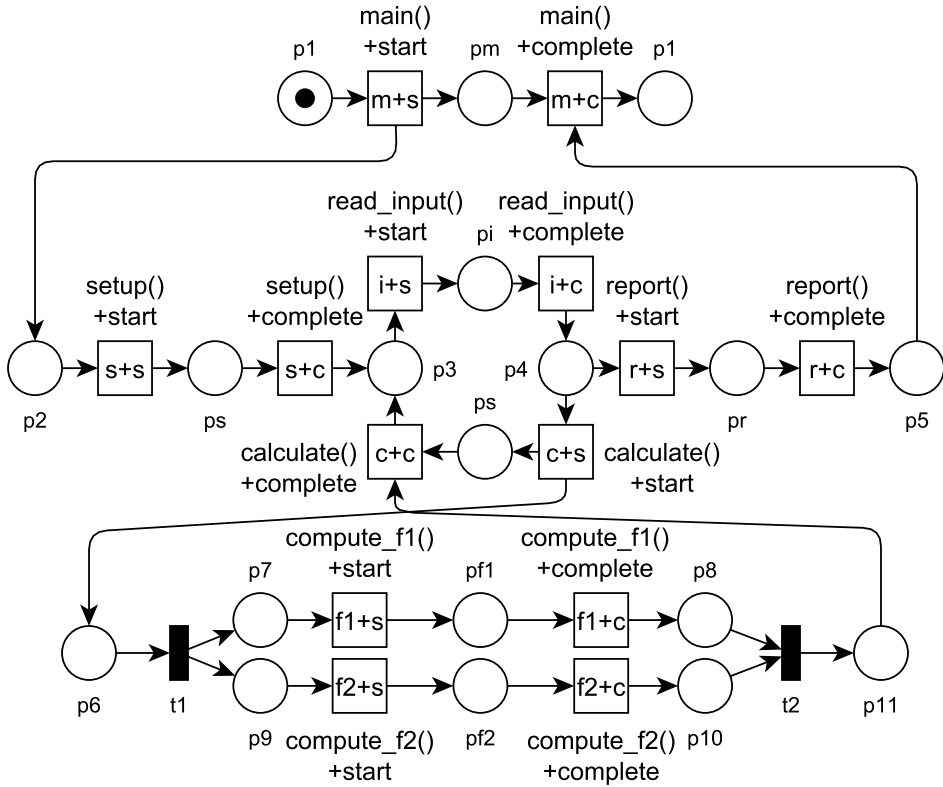


(a) Model depicted as a hierarchical process tree.



(b) Model depicted as a Petri net with hierarchy.

**Figure 8.3:** Example model for the program in Listing 8.1. This model could either be provided, or be discovered from the event log in Table 8.1 with, for example, the technique from Chapter 6.



**Figure 8.4:** The flat Petri net model corresponding to the model shown in Figure 8.3. Note that we used the non-atomic lifecycle notation from Definition 2.3.7 on page 49: +s denotes the *start* lifecycle transition, +c denotes the *complete* lifecycle transition. See Section 8.2.1 for more details.

As an example, consider the model in Figure 8.3 again. The corresponding flat Petri net shown in Figure 8.4 is derived by applying two rewriting steps: 1) unfolding each transition into a start and complete transition, and 2) gluing the subprocess into the main process. Note that we used the non-atomic lifecycle notation from Definition 2.3.7 on page 49. We refer to Section 9.2 for a detailed description of how the various hierarchy constructs can be translated.

### 8.2.2 Alignments

To establish an alignment between a process model and event log, we need to relate “moves” in the log to “moves” in the model, known as *alignment moves*. In the case that some of the moves in the log cannot be mimicked by the model and vice versa, we denote a *no move* ( $\gg$ ). Such a *no move* models a mismatch and possibly a deviation.





Consider a single *alignment move*, i.e., a single mapping between an event and a transition. In case an event is mapped to a  $\gg$ , we could not find a corresponding transition in the model for that event. This case is also known as a *log move*. In case a transition is mapped to a  $\gg$ , we could not find a corresponding event in the event log. This case is also known as a *model move*. In case an event is mapped to a transition, we denote such a move as a *synchronous move*. Below, we formalise these notions of alignment moves. Recall that system nets were introduced in Definition 2.2.5 on page 27.

**Definition 8.2.1 — Alignment Moves.** Let  $SN = (P, T, F, M_{ini}, M_{fin}, \ell)$  be a system net, and let  $L \subseteq \mathbb{E}^*$  be an event log. Let  $\gg \notin (\mathbb{E} \cup T)$  denote “no move”. We denote  $T^{\gg} = T \cup \{\gg\}$  and  $\mathbb{E}^{\gg} = \mathbb{E} \cup \{\gg\}$ .

Let  $\mathcal{A}$  be a set of alignment move identifiers. Given an alignment move  $m \in \mathcal{A}$ , let  $\#_T(m) \in T^{\gg}$  denote the associated move on model, and let  $\#_L(m) \in \mathbb{E}^{\gg}$  denote the associated move on log. If an alignment move  $m \in \mathcal{A}$  has no move on model, then  $\#_T(m) = \gg$ ; if an alignment move  $m \in \mathcal{A}$  has no move on log, then  $\#_L(m) = \gg$ . An alignment move  $m \in \mathcal{A}$  is a *legal alignment move* if and only if either:

- $m$  is a *log move*, i.e.,  $\#_L(m) \in \mathbb{E}$  and  $\#_T(m) = \gg$ ,
- $m$  is a *model move*, i.e.,  $\#_L(m) = \gg$  and  $\#_T(m) \in T$ , or
- $m$  is a *synchronous move*, i.e.,  $\#_L(m) \in \mathbb{E}$ ,  $\#_T(m) \in T$ , and both the event and transition agree on the label  $\lambda_{\#}(\#_L(m)) = \ell(\#_T(m))$ .

The set of all legal alignment moves is denoted as:

$$\mathcal{A}_{L,SN} = \{ m \in \mathcal{A} \mid (\#_L(m), \#_T(m)) \neq (\gg, \gg) \\ \wedge (\#_L(m) = \gg \vee \#_T(m) = \gg \vee \lambda_{\#}(\#_L(m)) = \ell(\#_T(m))) \}$$

Using the above notions of alignment moves as building blocks, we can now define the notion of an alignment, which relates an entire trace from an event log to a full firing sequence in a model.

**Definition 8.2.2 — Alignments.** Let  $SN = (P, T, F, M_{ini}, M_{fin}, \ell)$  be a system net, and let  $L \subseteq \mathbb{E}^*$  be an event log.

The *alignment* of a trace  $\sigma \in L$  and model  $SN$  is a sequence  $\gamma \in \mathcal{A}_{L,SN}^*$  such that no move is duplicated in  $\gamma$ , the projection on  $\#_L$ , ignoring  $\gg$ , yields the original trace  $\gamma|_L = \sigma$ , and the projection on  $\#_T$ , ignoring  $\gg$ , yields a full firing sequence  $\gamma|_{SN} \in S_{SN}$  (recall Definition 2.2.6 on page 28).

Analogous to a firing sequence, we can define the Petri net markings related to an alignment as follows. We define  $M_{ini} = M_0$  and let  $\gamma = \langle m_1, \dots, m_n \rangle$ . We define the “firing” of a “no move”  $\gg$  as an identity operation:  $M[\gg]M$ . We relate markings to alignment moves as follows:  $M_{i-1}[\#_T(m_i)]M_i$ .

■ **Example 8.1** Consider event log in Table 8.1 and the Petri net in Figure 8.4. For case 1, a possible alignment is given below. Note that we represent alignment moves vertically, with events on top and transitions below.

$$\gamma_1 = \begin{array}{c|c|c|c|c|c|c|c} m_1 & m_2 & m_3 & m_4 & m_5 & m_6 & m_7 & m_8 \\ \hline e_{1.1} & e_{1.2} & e_{1.3} & e_{1.4} & e_{1.5} & e_{1.6} & e_{1.7} & e_{1.8} \\ \hline m+s & s+s & s+c & i+s & i+c & r+s & r+c & m+c \end{array}$$

For example, in move  $m_1$  of alignment  $\gamma_1$ , the event  $e_{1.1}$  is mapped to transition  $m+s$ , indicating that the log and model both start (+s) the `main()` ( $m$ ) function. Observe that, since there are no log and model moves, this alignment shows that case 1 and the model can mimic each other perfectly.

Now consider case 3, which does not fit the Petri net model. The alignment now has to account for non-fitting behavior. Two possible alignments are:

$$\begin{array}{c} \gamma_{3,1} = \begin{array}{c|c|c|c|c|c|c|c|c|c} m_1 & m_2 & m_3 & m_4 & m_5 & m_6 & m_7 & m_8 & m_9 & m_{10} \\ \hline e_{3.1} & e_{3.2} & e_{3.3} & \gg & \gg & e_{3.4} & e_{3.5} & e_{3.6} & e_{3.7} & e_{3.8} \\ \hline m+s & s+s & s+c & i+s & i+c & \gg & \gg & r+s & r+c & m+c \end{array} \\ \\ \gamma_{3,2} = \begin{array}{c|c|c|c|c|c|c|c|c} m_1 & m_2 & m_3 & m_4 & m_5 & m_6 & m_7 & m_8 & m_9 \\ \hline e_{3.1} & e_{3.2} & e_{3.3} & \gg & \gg & e_{3.4} & \gg & \gg & \gg \\ \hline m+s & s+s & s+c & i+s & i+c & c+s & t1 & f1+s & f1+c \\ \\ m_{10} & m_{11} & m_{12} & m_{13} & m_{14} & m_{15} & m_{16} & m_{17} & m_{18} \\ \hline \gg & \gg & \gg & e_{3.5} & \gg & \gg & e_{3.6} & e_{3.7} & e_{3.8} \\ \hline f2+s & f2+c & t2 & c+c & i+s & i+c & r+s & r+c & m+c \end{array} \end{array}$$

In alignment  $\gamma_{3,1}$ , the non-fitting behavior is explained by skipping the deviating `calculate()` ( $c$ ) from the log (moves on log  $m_6, m_7$ ). In this case, the model still has to fire the `read_input()` ( $i$ ) to complete properly (moves on model  $m_4, m_5$ ). In alignment  $\gamma_{3,2}$ , the non-fitting behavior is explained by forcing the model to accept the deviating `calculate()` ( $c$ ) from the log (synchronous moves  $m_6, m_{13}$ ). In this case, the model fires `read_input()` ( $i$ ) multiple times to complete properly (moves on model  $m_4, m_5$  and  $m_{14}, m_{15}$ ), and the body of `calculate()` has to be skipped (moves on model  $m_7$  through  $m_{12}$ ). Observe that both  $\gamma_{3,1}|_L$  and  $\gamma_{3,2}|_L$  yield the original case 3. Likewise, both  $\gamma_{3,1}|_{SN}$  and  $\gamma_{3,2}|_{SN}$  yields full firing sequences in the Petri net. ■

### 8.2.3 Optimal Alignments

In the previous examples we showed that there are multiple possible alignments for the same trace and model. In practice, some alignments may be more desirable or likely than others. For example, alignment  $\gamma_{3,1}$  indicates only two deviations: a move on model for `read_input()` ( $m_4, m_5$ ) and a move on log for `calculate()` ( $m_6, m_7$ ). Alignment  $\gamma_{3,2}$  reported four deviations for the same trace and process model: two moves on model for `read_input()` ( $m_4, m_5$  and  $m_8, m_9$ ), a move on model for `compute_f1()` ( $m_8, m_9$ ), and a move on model

for `compute_f2()` ( $m_{10}, m_{11}$ ). Both alignments are valid and it depends on the application scenario which explanation is preferable.

In order to capture the explanation preferences, we can define the severity or cost of a deviation. This way, we can reduce the problem to finding an alignment with minimal deviation cost. Below, we will first introduce a cost function over individual legal alignment moves and then generalize this cost notion to complete alignments. The alignment with the lowest cost is called an *optimal alignment*.

**Definition 8.2.3 — Cost Function and Optimal Alignments.** Let  $SN$  be a system Petri net, and let  $\sigma \in L$  be a trace in an event log. Let  $\mathcal{A}_{L,SN}$  be the set of all legal alignment moves. A *cost function*  $\kappa : \mathcal{A}_{L,SN} \mapsto \mathbb{R}^+$  assigns a non-negative cost to each legal move.

The cost of an alignment  $\gamma \in \mathcal{A}_{L,SN}^*$  of trace  $\sigma$  and model  $SN$  is computed as the sum of the costs for all alignment moves:

$$\mathcal{K}(\gamma) = \sum_{m \in \gamma} \kappa(m)$$

An alignment  $\gamma$  is an *optimal alignment* if and only if for any alignment  $\gamma'$  of  $\sigma$  and  $SN$  the cost of  $\gamma'$  is greater or equal:  $\mathcal{K}(\gamma) \leq \mathcal{K}(\gamma')$ .

Using an alignment oracle, we assume that the optimal alignment of  $\sigma$  and  $SN$  is denoted as:  $opt(\sigma, SN) = \gamma \in \mathcal{A}_{L,SN}^*$ .

Many cost function schemes exist, favoring one type of explanation for deviations over the other. For our purpose, our goal is to find an alignment with a minimal amount of deviations (log and model moves). Therefore, for the remainder of this chapter, we will use the so-called *standard cost function* defined below, which assigns cost 0 to synchronous moves and tau moves, and cost 1 to log moves and non-tau model moves [21].

**Definition 8.2.4 — Standard Cost Function for Alignments.** The *standard cost function*  $\kappa_s$  for alignments assigns equal cost to moves on log and non-tau moves on model, as defined below.

$$\kappa_s(m) = \begin{cases} 1 & \text{if } \#_T(m) = \gg & \text{log moves} \\ 1 & \text{if } \#_L(m) = \gg \wedge \ell(\#_T(m)) \neq \tau & \text{non-tau model moves} \\ 0 & \text{otherwise} \end{cases}$$

Using the standard cost function, we can calculate the cost of the alignments  $\gamma_{3,1}$  and  $\gamma_{3,2}$  introduced before in Example 8.1. We have:  $\mathcal{K}_s(\gamma_{3,1}) = 4$  and  $\mathcal{K}_s(\gamma_{3,2}) = 8$ . Hence, in this setting, we can conclude that alignment  $\gamma_{3,1}$  is preferred over alignment  $\gamma_{3,2}$ . In fact, for case 3, alignment  $\gamma_{3,1}$  is an optimal alignment. That is, we have  $opt(\sigma_3, SN) = \gamma_{3,1}$ , where  $\sigma_3$  is case 3 and  $SN$  is the model from Figure 8.4.

## 8.3 Analysis Framework Definition

In this section, we present our hierarchical performance analysis framework for establishing precise relationships between events and submodels, taking into account execution semantics. The cornerstone of our framework is the *execution subtrace*: we take an aligned event log and a given submodel, and derive the parts of the aligned log that correspond to the given submodel. These execution subtraces are used to calculate the metrics from Section 8.4.

Before we can define the execution subtraces in Subsection 8.3.4, we need to address two subproblems: 1) how do we correlate subsequent alignment moves, taking into account the model semantics, and 2) how do we correlate alignment moves dealing with the same activity instance (start and complete)? For problem 1, we rely on the notion of *enabler moves* (Subsection 8.3.1). For problem 2, we rely on the notion of *execution intervals* (Subsection 8.3.2). In addition, we will rely on the correlations and computations detailed in Subsection 8.3.3. For an overview of the above concepts, see the outline in Figure 8.2 on page 210 and the summary provided by Figure 8.6 on page 225.

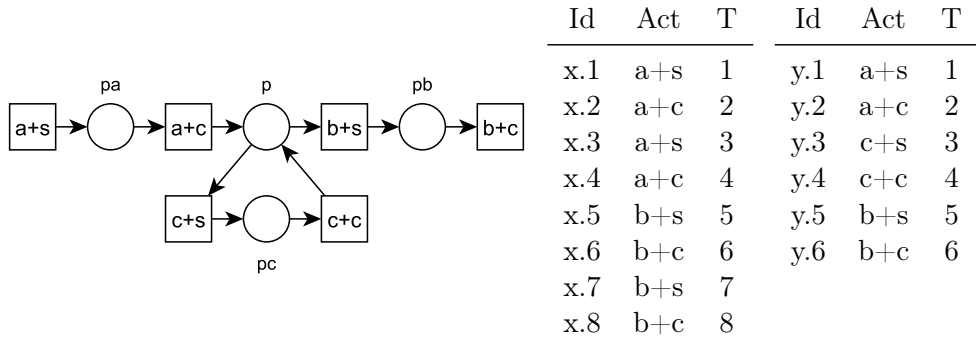
### 8.3.1 Move Enablers and Execution Policies

Recall, in Petri nets, a transition  $t$  is considered *enabled* in a marking  $M$  if and only if in that marking there are enough tokens on all the input places of  $t$ , see also Definition 2.2.3 on page 27. Hence, this notion of enabledness reflects the state where a transition can fire but is still waiting for the actual event where it is fired. In a given alignment sequence  $\gamma$ , this can be interpreted as the state between the alignment move  $m'$  that enabled a given transition  $t$  and the alignment move  $m$  in which transition  $t$  is actually fired. In this section, we make this type of model-based semantical information explicit via the notion of *move enablers*. That is, a *move enabler captures the alignment move  $m'$  that enabled a given alignment move  $m$* . In short, these move enablers combine information from both the event log (firing of a transition) and the model *execution policy* semantics (when is transition considered enabled).

**Definition 8.3.1 — Move Enabler.** A move enabler function  $en : \mathcal{A}_{L,SN} \mapsto \mathcal{A}_{L,SN} \cup \{\perp\}$  relates alignment moves  $m$  to preceding moves  $m'$  denoting the alignment move/transition that enabled the transition  $\#_T(m)$ , or  $\perp$  if no eligible preceding alignment move is available.

#### Execution Policies

In order to instantiate the move enabler function, we have to consider the model *execution policy* semantics. For our purpose, the execution policy determines when a transition is considered enabled. Below, we will first introduce a motivating example, and afterwards discuss two common execution policies.



**Figure 8.5:** Example Petri net with unbounded behavior, and two event log traces  $x$  and  $y$  with event id (Id), activity label (Act) and timestamp (T).

Consider the Petri net and event log in Figure 8.5. Observe that this model is unbounded: place  $p$  may contain multiple tokens. For this example, the corresponding optimal alignments for traces  $x$  and  $y$  are:

$$\gamma_x = \begin{array}{c|c|c|c|c|c|c|c} m_1 & m_2 & m_3 & m_4 & m_5 & m_6 & m_7 & m_8 \\ \hline e_{x.1} & e_{x.2} & e_{x.3} & e_{x.4} & e_{x.5} & e_{x.6} & e_{x.7} & e_{x.8} \\ \hline a+s & a+c & a+s & a+c & b+s & b+c & b+s & b+c \end{array}$$

$$\gamma_y = \begin{array}{c|c|c|c|c|c} m_1 & m_2 & m_3 & m_4 & m_5 & m_6 \\ \hline e_{y.1} & e_{y.2} & e_{y.3} & e_{y.4} & e_{y.5} & e_{y.6} \\ \hline a+s & a+c & c+s & c+c & b+s & b+c \end{array}$$

In alignment  $\gamma_x$ , one can debate which moves enabled move  $m_5$  (transition  $b+s$ ). After all, transition  $a+c$  has fired twice, and there are two tokens in place  $p$  before move  $m_5$  is performed. Some of the possible answers are:

1. move  $m_5$  is enabled by move  $m_4$  and move  $m_7$  is enabled by move  $m_2$ .
2. move  $m_5$  is enabled by move  $m_2$  and move  $m_7$  is enabled by move  $m_4$ .
3. both moves  $m_5$  and  $m_7$  are enabled by move  $m_4$ .

In these type of situations, the move enabler function has to make a choice and provide a consistent answer.

In alignment  $\gamma_y$ , another choice has to be made. After move  $m_2$ , the transition  $b+s$  is enabled. However, due to move  $m_3$ , transition  $b+s$  is disabled again. And after move  $m_4$ , transition  $b+s$  is enabled again. Do we consider the period between moves  $m_2$  and  $m_3$  for the purpose of when transition  $b+s$  was enabled? Again, the move enabler function has to make a choice and provide a consistent answer. If we only consider the period between moves  $m_4$  and  $m_5$ , we could calculate the waiting time for  $b+s$  as  $5 - 4 = 1$  time units, see events  $y.5$  and  $y.4$  in Figure 8.5. If we also consider the period between moves  $m_2$  and  $m_3$ , we could calculate the waiting time for  $b+s$  as  $(5 - 4) + (3 - 2) = 2$  time units.

To provide a consistent move enabler function, we rely on an execution policy. Various execution policies have been defined in the literature, in particular in the context of (generalized) stochastic Petri nets [140, 165]. Below, we will discuss two of the most common execution policies.

■ **Execution Policy 8.1 — Continuously Enabled (Race with Enabling Memory).**

In the *continuously enabled* policy, also known as the *race with enabling memory* policy, we look at the last move  $m'$  from which point on a move  $m$  / transition  $\#_T(m)$  is enabled, and not disabled until it is fired. We formally define the associated move enabler function below, and give some concrete examples afterwards.

**Definition 8.3.2 — Continuously Enabled Move Enabler.** Let  $\gamma \in \mathcal{A}_{L,SN}^*$  be an alignment of trace  $\sigma \in L$  and system net  $SN = (P, T, F, M_{ini}, M_{fin}, \ell)$ .

The *continuously enabled move enabler* is the last move that enabled the transition  $\#_T(m_i) \in T$  in alignment move  $m_i \in \gamma$ . We define  $en(m_1) = \perp$  as a base case. For  $1 < i \leq n$  we define  $en(m_i) = m_j$  such that:

1. move  $m_j$  occurs before  $m_i$ , i.e.,  $1 \leq j < i$ ,
2. move  $m_j$  enables the transition  $\#_T(m_i)$ , i.e.:

$$\neg(\bullet(\#_T(m_i)) \leq M_{j-1}) \wedge \bullet(\#_T(m_i)) \leq M_j$$

3. there does not exist an  $m_k$  with  $j < k < i$  such that:

$$\neg(\bullet(\#_T(m_i)) \leq M_k)$$

For example, under the continuously enabled policy, in alignment  $\gamma_x$ , both moves  $m_5$  and  $m_7$  are enabled by move  $m_4$ , as described in answer 3 on page 220. Likewise, in alignment  $\gamma_y$ , for move  $m_5$  we only consider the period between moves  $m_4$  and  $m_5$ .

The continuously enabled move enabler is a simple model to capture the waiting time between transitions, and can be used to calculate “hidden durations”. For example, consider Figure 8.4 and Table 8.1 again. Using the notion of continuous enabledness, we can measure the time between  $c+s$ ,  $f1+s$  and  $f2+s$ , thus capturing the startup overhead of the multi-threadedness for the program in Listing 8.1.

■ **Execution Policy 8.2 — Race with Age Memory.**

In the *race with age memory* policy, we consider all intervals where a transition is enabled and re-enabled. Hence, under the race with age memory policy, a move enabler function would not yield a single alignment move, but a series of enabled periods. In this policy, special care has to be taken to deal with multiple firings of a transition in a loop. For example, consider the Petri net in Figure 8.5 again. If transition  $c+s$  would fire multiple times, the enabled period of  $c+s$  before the first firing should not be included in the enabled period of

$c+s$  before the second firing, i.e., different executions should be separated. For example, under the race with age memory policy, in alignment  $\gamma_y$ , for move  $m_5$  we consider both the period between moves  $m_2$  and  $m_3$  and the period between moves  $m_4$  and  $m_5$ .

As an example of when to apply the race with age memory policy, suppose that the example in Figure 8.5 models a multi-threaded producer-consumer. Here, activity  $a$  models a producer, putting elements in queue  $p$ , and activity  $b$  models a consumer, taking elements from queue  $p$ . In this setup, activity  $c$  could represent a critical section that temporarily locks an element in queue  $p$ . Hence, instead of re-enabling activity  $b$ , activity  $c$  merely puts activity  $b$  on hold. Therefore, it makes sense to use all the intervals where transition  $b$  is enabled and re-enabled to calculate the time it takes for a produced element to be consumed. Note that, in order to correctly track each individual element in queue  $p$ , the tokens should be colored, i.e., have an associated identifier.

For the remainder of this chapter, we will assume Execution Policy 8.1 and the move enabler from Definition 8.3.2 for the sake of simplicity. Note that this is not a limitation and the remainder of the definitions can easily be adapted to alternative execution policies such as for example Execution Policy 8.2.

■ **Example 8.2** Consider the Petri net in Figure 8.4 and case 2 from the event log in Table 8.1. Assume we use the following optimal alignment:

$$\gamma_2 =$$

$m_1$	$m_2$	$m_3$	$m_4$	$m_5$	$m_6$	$m_7$	$m_8$	$m_9$	$m_{10}$
$e_{2.1}$	$e_{2.2}$	$e_{2.3}$	$e_{2.4}$	$e_{2.5}$	$e_{2.6}$	$\gg$	$e_{2.7}$	$e_{2.8}$	$e_{2.9}$
m+s	s+s	s+c	i+s	i+c	c+s	t1	f1+s	f2+s	f1+c
$m_{11}$	$m_{12}$	$m_{13}$	$m_{14}$	$m_{15}$	$m_{16}$	$m_{17}$	$m_{18}$	$m_{19}$	$m_{20}$
$e_{2.10}$	$\gg$	$e_{2.11}$	$e_{2.12}$	$e_{2.13}$	$e_{2.14}$	$\gg$	$e_{2.15}$	$e_{2.16}$	$e_{2.17}$
f2+c	t2	c+c	i+s	i+c	c+s	t1	f1+s	f1+c	f2+s
$m_{21}$	$m_{22}$	$m_{23}$	$m_{24}$	$m_{25}$	$m_{26}$	$m_{27}$	$m_{28}$		
$e_{2.18}$	$\gg$	$e_{2.19}$	$e_{2.20}$	$e_{2.21}$	$e_{2.22}$	$e_{2.23}$	$e_{2.24}$		
f2+c	t2	c+c	i+s	i+c	r+s	r+c	m+c		

Using the *continuously enabled move enabler* function, we have:

$$\begin{array}{llll}
 en(m_1) = \perp & en(m_2) = m_1 & en(m_3) = m_2 & en(m_4) = m_3 \\
 en(m_5) = m_4 & en(m_6) = m_5 & en(m_7) = m_6 & en(m_8) = m_7 \\
 en(m_9) = m_7 & en(m_{10}) = m_8 & en(m_{11}) = m_9 & en(m_{12}) = m_{11} \\
 en(m_{13}) = m_{12} & en(m_{14}) = m_{13} & en(m_{15}) = m_{14} & en(m_{16}) = m_{15} \\
 en(m_{17}) = m_{16} & en(m_{18}) = m_{17} & en(m_{19}) = m_{18} & en(m_{20}) = m_{17} \\
 en(m_{21}) = m_{20} & en(m_{22}) = m_{21} & en(m_{23}) = m_{22} & en(m_{24}) = m_{23} \\
 en(m_{25}) = m_{24} & en(m_{26}) = m_{25} & en(m_{27}) = m_{26} & en(m_{28}) = m_{27}
 \end{array}$$

Note the common move enabler for  $m_8$  and  $m_9$  and for  $m_{18}$  and  $m_{20}$  due to the AND-split transition  $t1$  in the model. ■

### Observable Move Enablers

When dealing with  $\tau$ -transitions (unobservable transitions), we make a distinction between normal and *observable move enablers*. In general, we can define the observable move enablers in terms of the normal move enablers.

**Definition 8.3.3 — Observable Move Enabler.** Given a move enabler function  $en$ , an *observable move enabler* function  $en^* : \mathcal{A}_{L,SN} \mapsto \mathcal{A}_{L,SN} \cup \{\perp\}$  relates an alignment move  $m$  to a preceding move  $m'$  denoting the observable move (non- $\tau$ -transition) that enabled move  $m$ :

$$en^*(m) = \begin{cases} en^*(en(m)) & \text{if } en(m) \neq \perp \wedge \ell(\#_T(en(m))) = \tau \\ en(m) & \text{otherwise} \end{cases}$$

Using the above example again, we have amongst others the following observable move enablers, which can differ from the normal move enablers:

$$\begin{array}{lll} en^*(m_7) = m_6 & en^*(m_8) = m_6 & en^*(m_9) = m_6 \\ en^*(m_{12}) = m_{11} & en^*(m_{13}) = m_{11} & en^*(m_{14}) = m_{13} \end{array}$$

### 8.3.2 Execution Intervals

In practice, activity instances are not atomic. Each execution of an activity has a start and a completion, with associated data like start and complete timestamps. In event logs, we capture this information via multiple events using lifecycle information, see also Table 8.1. Likewise, as explained in Section 8.2.1, we will explicitly use lifecycle information in the low-level Petri nets. In order to interpret the analysis results in the original, high-level representation, we need to know how to correlate start and complete alignment moves dealing with the same activity instance.

In this subsection, we will correlate alignment moves via intervals. We define *execution intervals* as a special type of interval that correlates alignment moves dealing with the same activity instance.

**Definition 8.3.4 — Interval.** Let  $\gamma \in \mathcal{A}_{L,SN}^*$  be an alignment of  $\sigma$  and  $SN$ . An *interval*  $i = (s, c)$ , with  $s \in \gamma$  and  $c \in \gamma$ , is a tuple of alignment moves such that  $s$  happened before  $c$  in  $\gamma$ .

The set of all intervals  $I_\gamma$  for an alignment  $\gamma$ , is denoted as:

$$I_\gamma = \{ (s, c) \mid s \in \gamma \wedge c \in \gamma \wedge \gamma = \gamma_1 \cdot \langle s \rangle \cdot \gamma_2 \cdot \langle c \rangle \cdot \gamma_3 \}$$

Given a set of intervals  $I \subseteq I_\gamma$ , we denote  $set(I) = \bigcup_{(s,c) \in I} \{s, c\}$ .

Execution intervals are a special subset of intervals that describe the start and complete of activity instances. Since every activity instance is described by one start event and one complete event, no two execution intervals can share



an alignment move. That is, every alignment move is unique (i.e., occurs at most once) in the set of execution intervals.

**Definition 8.3.5 — Execution Interval.** Let  $\gamma \in \mathcal{A}_{L,SN}^*$  be an alignment of a trace  $\sigma$  and a model  $SN$ . Assume that the label of each transition  $t \in T$  is defined by an *activity* and a *lifecycle transition*, i.e.,  $\ell(t) \in (\mathbb{A} \times LC)$ . Let  $\ell(t)|_{act} \in \mathbb{A}$  denote the projection of a label to an *activity* and let  $\ell(t)|_{life} \in LC$  denote the projection of a label to a *lifecycle transition*.

An *execution interval*  $i = (s, c) \in I_\gamma$  is a tuple of alignment moves that describes the start and completion of an activity on the model, with start before completion, i.e., we have a model or synchronous move ( $\#_T(s) \in T$  and  $\#_T(c) \in T$ ) over the same activity ( $\ell(\#_T(s))|_{act} = \ell(\#_T(c))|_{act}$ ) with start and complete ( $\ell(\#_T(s))|_{life} = start \wedge \ell(\#_T(c))|_{life} = complete$ ).

The set of all execution intervals for an alignment  $\gamma$  is denoted as the set  $I_\gamma^E \subseteq I_\gamma$  such that:

- all non- $\tau$  moves are in  $I_\gamma^E$ , i.e.:

$$set(I_\gamma^E) = \{m \in \gamma \mid \ell(\#_T(m)) \neq \tau\}$$

- no two execution intervals  $(s, c), (s', c') \in I_\gamma^E$  share alignment moves:

$$\forall (s, c), (s', c') \in I_\gamma^E : \{s, c\} \cap \{s', c'\} \neq \emptyset \Rightarrow (s, c) = (s', c')$$

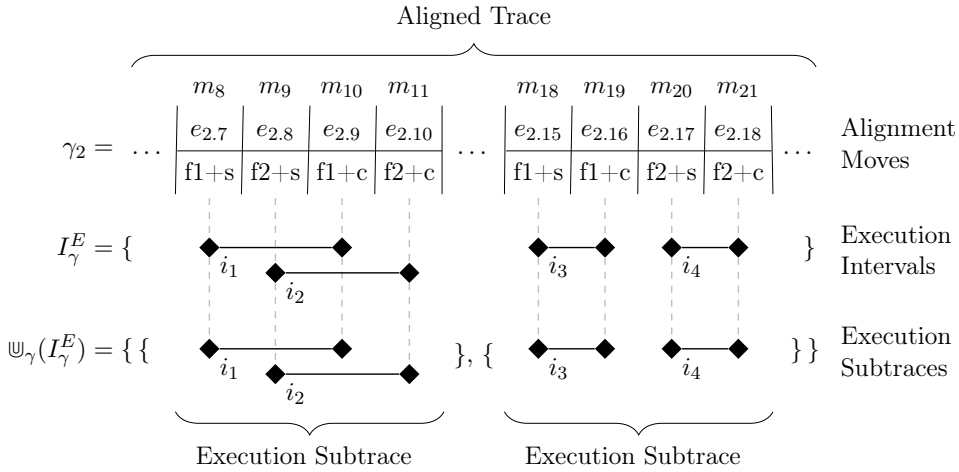
As an example, consider alignment  $\gamma_2$  from Example 8.2 on page 222 again. The set of execution intervals for this alignment equals:

$$I_{\gamma_2}^E = \{(m_1, m_{28}), (m_2, m_3), (m_4, m_5), (m_6, m_{13}), (m_8, m_{10}), (m_9, m_{11}), \\ (m_{14}, m_{15}), (m_{16}, m_{23}), (m_{18}, m_{19}), (m_{20}, m_{21}), (m_{24}, m_{25}), \\ (m_{26}, m_{27})\}$$

Using the above set, we can, for example, conclude that activity `main()` ( $m$ ) has one instance (execution interval  $(m_1, m_{28})$ ), while activity `read_input()` ( $i$ ) has three instances (execution intervals  $(m_4, m_5)$ ,  $(m_{14}, m_{15})$ , and  $(m_{24}, m_{25})$ ).

Observe that  $(m_7, m_8) \in I_{\gamma_2}$  and  $(m_8, m_{11}) \in I_{\gamma_2}$  are also valid intervals, but they are not valid *execution* intervals, i.e.,  $(m_7, m_8) \notin I_{\gamma_2}^E$  and  $(m_8, m_{11}) \notin I_{\gamma_2}^E$  because they do not describe the start and completion of the same activity.

Figure 8.6 depicts some execution intervals using events from Table 8.1 and the model shown in Figure 8.4. Note that, when only looking at the event log, the start-complete pairing for activity `f1` is ambiguous: there are two events labeled `f1+c`. However, when using an aligned trace, we can rely on the unfolded Petri net transitions to remove most of the ambiguity.



**Figure 8.6:** The concepts and terminology of Section 8.3 depicted using the optimal alignment  $\gamma_2$  of case 2, see Example 8.2 on page 222, and assuming the submodel involving  $T' = \{f_{1+s}, f_{1+c}, f_{2+s}, f_{2+c}\}$ .

### 8.3.3 Interval Correlation and Computation

The intervals introduced in the previous section express a period between two events. When we recognize the timing information and associated ordering in an event log, we can express several interesting computations.

In this section, we start with a generic definition for interval correlation and partitioning. We use this partitioning in Subsection 8.3.4 for grouping execution intervals into execution substraces. In addition, we introduce several useful computations, which we leverage for our performance metrics in Section 8.4.

**Definition 8.3.6 — Interval Correlation and Partitioning.** Let  $I \subseteq I_\gamma$  be a set of intervals from an alignment  $\gamma$ .

An *interval correlation* is a relation  $C_\gamma \subseteq (I_\gamma \times I_\gamma)$  that correlates intervals in the set  $I_\gamma$ . We write  $C_\gamma^+$  to denote the reflexive, symmetric and transitive closure of  $C_\gamma$ .

We can partition an interval set  $I$  into a set of equivalence sets based on an interval correlation  $C_\gamma$ , denoted  $C_\gamma(I) \subset \mathcal{P}(I)$ , such that:

- the union of all partitions equals the original set:  $I = \bigcup_{J \in C_\gamma(I)} J$ ,
- for any partition  $J \in C_\gamma(I)$  we have  $\forall i, i' \in J : (i, i') \in C_\gamma^+$ , and
- for any  $J, J' \in C_\gamma(I)$  we have  $\forall i \in J, i' \in J' : (i, i') \in C_\gamma^+ \Rightarrow J = J'$ .

A useful interval correlation relation is the *overlapping interval correlation*. This relation expresses whether two intervals overlap with respect to the timing information and associated ordering. Recall that  $\#_{time}$  denotes the timestamp attribute, see also Definition 2.3.1 on page 45.



**Definition 8.3.7 — Overlapping Interval Correlation.** The interval correlation  $\uplus_\gamma$  correlates intervals if and only if they overlap with respect to  $\#_{time}$ , i.e.:

$$(s, c) \uplus_\gamma (s', c') \Leftrightarrow \#_{time}(s \downarrow_L) \leq \#_{time}(c' \downarrow_L) \wedge \#_{time}(s' \downarrow_L) \leq \#_{time}(c \downarrow_L)$$

For example, consider the execution intervals from Figure 8.6. Executions intervals  $i_1 = (m_8, m_{10})$  and  $i_2 = (m_9, m_{11})$  are overlapping ( $i_1 \uplus_{\gamma_2} i_2$ ), but  $i_3 = (m_{18}, m_{19})$  and  $i_4 = (m_{20}, m_{21})$  do not ( $\neg(i_3 \uplus_{\gamma_2} i_4)$ ).

When working with sets of intervals  $I$ , we define the additional operations detailed below. These operations enable interval arithmetics used for calculating some of the metrics in Section 8.4.

The *smallest containing interval* denotes the smallest interval that contains all the intervals in  $I$ .

**Definition 8.3.8 — Minimum, Maximum, and Smallest Containing Interval.**

Let  $I \subseteq I_\gamma$  be a set of intervals. We define:

- $min(I)$  The minimum equals  $min(I) = s \in set(I)$  such that for all  $s' \in set(I)$  we have  $\#_{time}(s \downarrow_L) \leq \#_{time}(s' \downarrow_L)$ .
- $max(I)$  The maximum equals  $max(I) = c \in set(I)$  such that for all  $c' \in set(I)$  we have  $\#_{time}(c' \downarrow_L) \leq \#_{time}(c \downarrow_L)$ .
- $sci(I)$  The smallest containing interval equals:  
 $sci(I) = (min(I), max(I))$ .

As an example, consider the intervals from Figure 8.6. We can calculate the following minimum, maximum and smallest containing intervals:

$$\begin{array}{lll} min(\{i_1, i_2\}) = m_8 & min(\{i_1, i_3\}) = m_8 & min(\{i_2\}) = m_9 \\ max(\{i_1, i_2\}) = m_{11} & max(\{i_1, i_3\}) = m_{19} & max(\{i_2\}) = m_{11} \\ sci(\{i_1, i_2\}) = (m_8, m_{11}) & sci(\{i_1, i_3\}) = (m_8, m_{19}) & sci(\{i_2\}) = i_2 \end{array}$$

The *condensed overlapping intervals* merges intervals in  $I$  that overlap according to  $\uplus$ , and returns the resulting set of intervals.

**Definition 8.3.9 — Condensed overlapping intervals.** Let  $I \subseteq I_\gamma$  be a set of intervals and let  $\uplus_\gamma$  be the overlapping interval correlation.

The set of condensed overlapping intervals  $coi(I)$  is denoted as:

$$coi(I) = \{ sci(I') \mid I' \in \uplus_\gamma(I_\gamma) \}$$

Note that  $\uplus_\gamma(I_\gamma)$  is the set of equivalence sets in  $I$  under  $\uplus_\gamma$ .

As an example, consider the intervals from Figure 8.6 again. We can calculate the following condensed overlapping intervals:

$$\begin{array}{ll} coi(\{i_2\}) = \{i_2\} & coi(\{i_1, i_2\}) = \{(m_8, m_{11})\} \\ coi(\{i_3, i_4\}) = \{i_3, i_4\} & coi(\{i_1, i_2, i_3, i_4\}) = \{(m_8, m_{11}), i_3, i_4\} \end{array}$$

Observe that the resulting sets of smallest containing and condensed overlapping intervals are not necessarily a set of *execution* intervals anymore.

### 8.3.4 Execution Subtraces

The cornerstone of our framework is the *execution subtrace*: we take an aligned event log and a given submodel, and derive the parts of the aligned log that correspond to the given submodel. In this subsection, we will define execution subtraces as a *set of execution intervals*, correlated using the model execution policy semantics by relying on the *move enabler* function.

As explained in Subsection 8.3.1, multiple execution policies are possible. For the *continuously enabled* move enabler policy from Definition 8.3.2, we define the execution correlation relation below.

**Definition 8.3.10 — Execution Correlation for Continuously Enabled.** The execution correlation  $\mathbb{U}_\gamma$  for the continuously enabled execution policy correlates intervals if and only if they semantically represent the same instance or subrun under said execution policy. More precisely, two intervals  $(s, c)$ ,  $(s', c')$  semantically contribute to the same subrun  $((s, c)\mathbb{U}_\gamma(s', c'))$  if either:

1. they are enabled by the same move ( $en(s) = en(s')$ ), or
2. they subsequently follow one another ( $en(s') = c$ ), or
3. they subsequently complete in a subprocess relation ( $en(c') = c$ ).

As an example, consider alignment  $\gamma_2$  from Example 8.2 on page 222 again. Using the execution correlation  $\mathbb{U}_{\gamma_2}$ , we have:

$$\begin{array}{ll} (m_8, m_{10}) \mathbb{U}_{\gamma_2} (m_9, m_{11}) & \textbf{Condition 1: } en(m_8) = en(m_9) = m_7 \\ (m_2, m_3) \mathbb{U}_{\gamma_2} (m_4, m_5) & \textbf{Condition 2: } en(m_4) = m_3 \\ (m_7, m_{12}) \mathbb{U}_{\gamma_2} (m_6, m_{13}) & \textbf{Condition 3: } en(m_{13}) = m_{12} \end{array}$$

However, the following pairs are not execution correlated:

$$\neg((m_4, m_5) \mathbb{U}_{\gamma_2} (m_{25}, m_{27})) \quad \neg((m_8, m_{11}) \mathbb{U}_{\gamma_2} (m_6, m_{13}))$$

Using the *execution correlation*, we can now derive the *execution subtraces* from a given trace in an event log and a given submodel. In Section 8.1 we listed numerous ways of deriving submodels, be it subprocesses, semantical or graph patterns, hierarchies, or other graph, data, or semantic based structures. For our framework, we support all of the above structures by assuming a very generic definition of a submodel: a subset of transitions  $T' \subseteq T$ .

**Definition 8.3.11 — Execution Subtrace.** Let  $SN = (P, T, F, M_{ini}, M_{fin}, \ell)$  be a system net, and let  $\sigma \in L$  be a trace in an event log. Let  $\mathbb{U}_\gamma$  be an execution correlation relation. Assume a submodel given by the subset of transitions  $T' \subseteq T$ .

The *set of execution subtraces* for trace  $\sigma$  and submodel  $T' \subseteq T$ , denoted  $\mathcal{I} = \text{subtraces}(\sigma, SN, T')$ , is derived by filtering the execution intervals  $I_\gamma^E$  on  $T'$ , denoted  $I_\gamma^E \upharpoonright_{T'}$ , and then partition the result using the execution correlation  $\Psi_\gamma$ , i.e.:

$$\mathcal{I} = \text{subtraces}(\sigma, SN, T') = \Psi_\gamma(\{ (s, c) \in I_\gamma^E \upharpoonright_{T'} \mid \gamma = \text{opt}(\sigma, SN) \})$$

**where**  $I_\gamma^E \upharpoonright_{T'} = \{ (s, c) \in I_\gamma^E \mid \#_T(s) \in T' \wedge \#_T(c) \in T' \}$

Observe that each *execution subtrace*  $I \in \mathcal{I}$  is a set of correlated execution intervals. We extend execution subtraces to event logs as follows:

$$\text{subtraces}(L, SN, T') = \{ \text{subtraces}(\sigma, SN, T') \mid \sigma \in L \}$$

As an example, consider alignment  $\gamma_2$  from Example 8.2 on page 222 again. Using the submodel  $T' = \{s+s, s+c, i+s, i+c\}$ , we get the following three execution subtraces:

$$\mathcal{I} = \text{subtraces}(\sigma_2, SN, T') = \{ \overbrace{\{ (m_2, m_3), (m_4, m_5) \}}^{\text{execution subtrace } I_1}, \underbrace{\{ (m_{14}, m_{15}) \}}_{\text{exec. subtrace } I_2}, \underbrace{\{ (m_{24}, m_{25}) \}}_{\text{exec. subtrace } I_3} \}$$

If we would use the submodel  $T' = \{f1+s, f1+c, f2+s, f2+c\}$  instead, we would get the following two execution subtraces, see also Figure 8.6:

$$\text{subtraces}(\sigma_2, SN, T') = \{ \{ (m_8, m_{10}), (m_9, m_{11}) \}, \{ (m_{18}, m_{19}), (m_{20}, m_{21}) \} \}$$

Note that, using the ordering imposed by the move enabler function, the execution intervals in an execution subtrace are partially ordered. This in contrast to an aligned trace, which is totally ordered. In the next section, we will use the above framework notions to define metrics over execution intervals.

## 8.4 Metrics for Execution Intervals

In the previous section, we have defined our framework for semantic-aware execution subtraces, taking into account model execution semantics. In this section, we formalize a selection of existing and novel performance metrics using the above framework. We will first discuss semantic-aware filters (Subsection 8.4.1). Next, we introduce the formulas to compute the proposed metrics (Subsection 8.4.2). After that, we discuss how to interpret metric results (Subsection 8.4.3), and show an integrated example (Subsection 8.4.4).

### 8.4.1 Semantic-aware Filters

Consider the incoming arcs for the input place of transition  $i+s$  in the model from Figure 8.4. Semantically, two control-flow parts “flow” into transition  $i+s$ : one flow originates from  $s+c$  and the other from  $c+c$ . When calculating metrics on the incoming arcs, it is useful to be able to distinguish these two flows. *Semantic-aware filters* enable this kind of filtering on (execution) intervals. That is, instead of only looking at the event log, we leverage knowledge from both the log and the model via the notions introduced in Section 8.3. Below, we introduce a *move enabler filter* and an *accept filter*, which we will leverage in the metric formalizations in Section 8.4.2.

The *move enabler filter* allows us to distinguish incoming flows. This filter uses the enabler move notions from Definition 8.3.1 to check where alignment moves “originate from” in the process model, and allows filtering based on this information.

**Definition 8.4.1 — Move Enabler Filter.** Let  $SN = (P, T, F, M_{ini}, M_{fin}, \ell)$  be a system net, let  $T_F \subseteq T$  be a subset of transitions in  $SN$ , and let  $en$  be a move enabler function.

In the *move enabler filter*, we accept an alignment move  $s \in \mathcal{A}_{L,SN}$  if and only if it is enabled by a transition in  $T_F$ . We accept an interval  $i = (s, c)$  if and only if move  $s$  is accepted. And we accept a set of intervals  $I \subseteq I_\gamma$  if and only if the “first” or minimum move is accepted.

$$\begin{aligned} filter(s, T_F) &\Leftrightarrow \#_T(en(s)) \in T_F && \text{where } s \in \mathcal{A}_{L,SN} \\ filter(i, T_F) &= filter(s, T_F) && \text{where } i = (s, c) \\ filter(I, T_F) &= filter(min(I), T_F) && \text{where } I \subseteq I_\gamma \end{aligned}$$

For example, consider the model in Figure 8.4 and the alignment  $\gamma_2$  from Example 8.2 on page 222. When we use the submodel  $T' = \{i+s, i+c\}$ , and the subset of transitions  $T_F = \{c+c\}$ , we have:

$$\begin{aligned} \mathcal{I} = subtraces(\sigma_2, SN, T') &= \{ \{ (m_4, m_5) \}, \{ (m_{14}, m_{15}) \}, \{ (m_{24}, m_{25}) \} \} \\ \{ I \in \mathcal{I} \mid filter(I, T_F) \} &= \{ \{ (m_{14}, m_{15}) \}, \{ (m_{24}, m_{25}) \} \} \end{aligned}$$

Note that execution interval  $(m_4, m_5)$  was dropped because  $en(m_4) = m_3$  with  $\#_T(m_3) = i+c \notin T_F$ .

The *accept filter* checks whether an interval contains event data (i.e., filtering on synchronous moves).

**Definition 8.4.2 — Accept Filter (Event Data Filter).** In the *accept filter*, we accept an interval  $i = (s, c)$  if and only if it is a synchronous move. We

accept a set of intervals  $I$  if it contains a synchronous move.

$$\begin{aligned} \text{accept}(i) &\Leftrightarrow (\#_L(s) \neq \gg \wedge \#_L(c) \neq \gg) && \text{where } i = (s, c) \\ \text{accept}(I) &= (\exists i \in I : \text{accept}(i)) && \text{where } I \subseteq I_\gamma \end{aligned}$$

We combine the accept and move enabler filter as follows, where  $T_F \subseteq T$  is a subset of transitions as per Definition 8.4.1.

$$\text{accept}(I, T_F) = (\text{accept}(I) \wedge \text{filter}(I, T_F))$$

For example, consider the alignment  $\gamma_{3,1}$  from Example 8.1 on page 216. When we use the submodel  $T' = \{s+s, s+c, i+s, i+c\}$ , and the subset of transitions  $T_F = \{c+c\}$ , we have:

$$\begin{aligned} \mathcal{I} &= \text{subtraces}(\sigma_3, SN, T') = \{ \{ (m_2, m_3), (m_4, m_5) \} \} \\ \{ I \in \mathcal{I} \mid \text{accept}(I) \} &= \{ \{ (m_2, m_3) \} \} \\ \{ I \in \mathcal{I} \mid \text{accept}(I, T_F) \} &= \emptyset \end{aligned}$$

Note that execution interval  $(m_4, m_5)$  was dropped by  $\text{accept}(I)$  because both alignment moves  $m_4$  and  $m_5$  are model moves, i.e., the interval contains no event data. Execution interval  $(m_2, m_3)$  was dropped by  $\text{accept}(I, T_F)$  because  $\text{en}(m_2) = m_1$  with  $\#_T(m_1) = m+c \notin T_F$ .

### 8.4.2 Metric Formalizations

Below we formalize our performance metrics using the framework and filters we defined before. We use the following input (left) and notations (right):

$L$	An event log	$\mathcal{I}$	A set of execution subtraces
$SN$	A system net	$I \in \mathcal{I}$	An execution subtrace
$T', T''$	Submodels in $SN$	$(s, c) \in I$	An execution interval
$T_F$	A set of incoming transitions to filter on	$s, c$	Alignment moves

For the example metric calculations below, we will reuse the model in Figure 8.4 and the log in Table 8.1. Recall that the optimal alignments  $\gamma_1$  and  $\gamma_{3,1}$  were introduced in Example 8.1 on page 216, and optimal alignment  $\gamma_2$  was introduced in Example 8.2 on page 222.

#### Frequency-based Metrics

The frequency-based metrics described below count the occurrences of various properties. The *absolute* and *case frequencies* are well-known metrics; they illustrate the workings of the framework for simple metrics. The *model-move frequency* illustrates a conformance metric, i.e., it measures local deviations between an event log and a model. The *resource frequency* metric uses the

resource classifier  $\#_{res}$  (see Definition 2.3.1 on page 45) to illustrate how additional data in event logs can be used in metrics. The *followed-by frequency* metric measures executions which are followed by a specific control-flow part of the model.

■ **Performance Metric 8.1 — Absolute Frequency.** *[Numerical]*

*Description:* Counts the number of submodel executions.

*Definition:*

$$\begin{aligned} AbsFreq(L, SN, T', T_F) = \\ |\{I \mid I \in subtraces(L, SN, T') \wedge I \in \mathcal{I} \wedge accept(I, T_F)\}| \end{aligned}$$

*Example:* Consider the submodel  $T' = \{i+s, i+c\}$ . With the identity move enabler filter  $T_F = T$ , we count one execution in alignment  $\gamma_1$  (subtrace  $\{(m_4, m_5)\}$ ), three executions in alignment  $\gamma_2$  (subtraces  $\{(m_4, m_5)\}$ ,  $\{(m_{14}, m_{15})\}$ ,  $\{(m_{24}, m_{25})\}$ ), and zero executions in alignment  $\gamma_{3,1}$  (move on models are filtered by  $accept()$ ). Hence, we have  $AbsFreq(L, SN, T', T_F) = 4$ . With the move enabler filter  $T_F = \{s+c\}$ , we count one execution in alignment  $\gamma_1$  (subtrace  $\{(m_4, m_5)\}$ ), one execution in alignment  $\gamma_2$  (subtrace  $\{(m_4, m_5)\}$ ), and zero executions in alignment  $\gamma_{3,1}$ . Hence, we have  $AbsFreq(L, SN, T', T_F) = 2$ .

■ **Performance Metric 8.2 — Case Frequency.** *[Numerical]*

*Description:* Counts the number of traces/cases with submodel executions.

*Definition:*

$$\begin{aligned} CaseFreq(L, SN, T', T_F) = \\ |\{I \mid I \in subtraces(L, SN, T') \wedge \exists I \in \mathcal{I} : accept(I, T_F)\}| \end{aligned}$$

*Example:* Consider the submodel  $T' = \{i+s, i+c\}$  and the identity move enabler filter  $T_F = T$ . There exists an execution in alignment  $\gamma_1$  (subtrace  $\{(m_4, m_5)\}$ ), an execution in alignment  $\gamma_2$  (any of the subtraces  $\{(m_4, m_5)\}$ ,  $\{(m_{14}, m_{15})\}$ ,  $\{(m_{24}, m_{25})\}$ ), and no executions in alignment  $\gamma_{3,1}$  (move on models are filtered by  $accept()$ ). Hence, we have  $CaseFreq(L, SN, T', T_F) = 2$ . Consider the submodel  $T' = \{c+s, c+c\}$ . There exists no executions in alignment  $\gamma_1$ , an execution in alignment  $\gamma_2$  (subtraces  $\{(m_6, m_{13})\}$ ,  $\{(m_{16}, m_{23})\}$ ), and no executions in alignment  $\gamma_{3,1}$  (move on logs yield no execution intervals). Hence, we have  $CaseFreq(L, SN, T', T_F) = 1$ .

■ **Performance Metric 8.3 — Model-move Frequency.** *[Numerical]*

*Description:* Counts the number of executions involving model-moves.

*Definition:*

$$\begin{aligned} ModelFreq(L, SN, T', T_F) = \\ |\{I \mid I \in subtraces(L, SN, T') \wedge I \in \mathcal{I} \wedge filter(I, T_F) \\ \wedge (\exists m \in set(I) : \#_L(m) = \gg \wedge \ell(\#_T(m)) \neq \tau)\}| \end{aligned}$$



*Example:* Consider the submodel  $T' = \{i+s, i+c\}$ . With the identity move enabler filter  $T_F = T$ , we count no model-move executions in alignment  $\gamma_1$ , no model-move executions in alignment  $\gamma_2$  (tau moves yield no execution intervals), and one model-move execution in alignment  $\gamma_{3,1}$  (subtrace  $\{(m_4, m_5)\}$ ). Hence, we have  $ModelFreq(L, SN, T', T_F) = 1$ . With the move enabler filter  $T_F = \{c+c\}$ , we count no model-move executions in alignment  $\gamma_1$ , no model-move executions in alignment  $\gamma_2$ , and no model-move executions in alignment  $\gamma_{3,1}$  (alignment move  $m_4$  is enabled by  $s+c$  and not by  $c+c$ ). Hence, we have  $ModelFreq(L, SN, T', T_F) = 0$ .

■ **Performance Metric 8.4 — Resource Frequency.** [*Multiset*]

*Description:* Counts the number of resources involved in each execution.

*Definition:*

$$ResFreq(L, SN, T', T_F) = \\ [ |\{ \#_{res}(\#_L(m)) \mid m \in set(I) \} | \\ | \mathcal{I} \in subtraces(L, SN, T') \wedge I \in \mathcal{I} \wedge accept(I, T_F) | ]$$

*Example:* Consider the submodel  $T' = \{i+s, i+c\}$ . With the identity move enabler filter  $T_F = T$ , we count one time 1 resource in alignment  $\gamma_1$  (subtrace  $\{(m_4, m_5)\}$  yields  $\{\text{thread-1}\}$ ), three times 1 resource in alignment  $\gamma_2$  (subtraces  $\{(m_4, m_5)\}$ ,  $\{(m_{14}, m_{15})\}$ ,  $\{(m_{24}, m_{25})\}$  each yield  $\{\text{thread-1}\}$ ), and no resources in alignment  $\gamma_{3,1}$  (move on logs yield no execution intervals). Hence, we have  $ResFreq(L, SN, T', T_F) = [1^4]$ . Consider the submodel  $T' = \{f1+s, f1+c, f2+s, f2+c\}$ . We count no resources in alignments  $\gamma_1$  and  $\gamma_{3,1}$  (no execution intervals for  $f1$  and/or  $f2$ ), and we count one time 1 and one time 2 resources in alignment  $\gamma_2$  (subtrace  $\{(m_8, m_{10}), (m_9, m_{11})\}$  yields  $\{\text{thread-2}, \text{thread-3}\}$  and the subtrace  $\{(m_{18}, m_{19}), (m_{20}, m_{21})\}$  yields  $\{\text{thread-2}\}$ ). Hence, we have  $ResFreq(L, SN, T', T_F) = [1, 2]$ .

■ **Performance Metric 8.5 — Followed-by Frequency.** [*Number, Two Submodels*]

*Description:* Calculates the number of executions of submodel  $T'$  that continue with submodel  $T''$ .

*Definition:*

$$FollowFreq(L, SN, T', T'', T_F) = \\ [ \{ I \mid \mathcal{I} \in subtraces(L, SN, T') \wedge I \in \mathcal{I} \wedge accept(I, T_F) \\ \wedge \exists I' \in subtraces(L, SN, T'') \wedge I' \in \mathcal{I}' : max(I) = en^*(min(I')) \} | ]$$

*Example:* Consider the submodel  $T' = \{i+s, i+c\}$  and the submodel  $T'' = \{c+s, c+c\}$ . With the identity move enabler filter  $T_F = T$ , we have one execution in alignment  $\gamma_1$  (subtrace  $\{(m_4, m_5)\}$ ), three executions in alignment  $\gamma_2$  (subtraces  $\{(m_4, m_5)\}$ ,  $\{(m_{14}, m_{15})\}$ ,  $\{(m_{24}, m_{25})\}$ ), and zero executions in

alignment  $\gamma_{3,1}$  (move on models are filtered by  $accept()$ ). In alignment  $\gamma_1$ , move  $m_5$  is followed by  $r+s \notin T''$  and hence the corresponding subtrace is discarded ( $en^*(m_6) = m_5$ ,  $\#_T(m_6) = r+s$ ). Similarly, in alignment  $\gamma_2$ , move  $m_{25}$  is followed  $r+s$  and thus discarded, but moves  $m_5$  and  $m_{15}$  are followed by  $c+s$  and thus the corresponding subtraces are counted. Hence, we have  $FollowFreq(L, SN, T', T'', T_F) = 2$ . Note that this metric can prove especially useful when the followed-by submodel  $T''$  represents a meaningful or insightful branch or choice. For example, this metric can be used to check how often certain activities are followed by a cancelation trigger, e.g., see Figure 7.1 in Chapter 7 with  $T' = \{a, b, p\}$  and  $T'' = \{h\}$ .

### Time-based Metrics

The time-based metrics described below use the time information available via the time classifier  $\#_{time}$  (see Definition 2.3.1 on page 45). We denote the difference in time between two events  $e, e'$ , notation  $\delta_{time}(e, e')$ , as:

$$\delta_{time}(e, e') = (\#_{time}(e') - \#_{time}(e))$$

The *duration*, *waiting*, and *sojourn time* are well-known time-based metrics, and show how the (execution) intervals can be used in metric computation. Note that the waiting time is based on *observable* move enablers. During the analysis of software system event logs in various case studies, we have found several additional performance metrics users found useful in their tasks. The *cumulative duration* expresses the total time spent in a submodel per case, or in software analysis terms, a single run of the entire software system process. The *own duration* captures the “own time” notion commonly found in software profilers: it denotes the time spent in a subprocess  $T'$ , while ignoring the time spent in lower-level subprocesses  $T''$ . The *duration efficiency* proved useful for analyzing multi-threaded software systems: it expresses the “amount of overlap” in execution intervals.

#### ■ Performance Metric 8.6 — Duration / Service Time. [Multiset]

*Description:* Calculates the time spent during each submodel execution.

*Definition:*

$$\begin{aligned} Dur(L, SN, T', T_F) = \\ [\delta_{time}(\#_L(s), \#_L(c)) \mid \mathcal{I} \in subtraces(L, SN, T') \wedge \mathbf{I} \in \mathcal{I} \\ \wedge accept(\mathbf{I}, T_F) \wedge (s, c) = sci(\mathbf{I})] \end{aligned}$$

*Example:* Consider the submodel  $T' = \{f1+s, f1+c, f2+s, f2+c\}$ . With the identity move enabler filter  $T_F = T$ , we have no subtraces in alignments  $\gamma_1$  and  $\gamma_{3,1}$  (no execution intervals for  $f1, f2$ ), and we have two subtraces in alignment  $\gamma_2$ :  $\{(m_8, m_{10}), (m_9, m_{11})\}$  and  $\{(m_{18}, m_{19}), (m_{20}, m_{21})\}$ . The smallest

containing intervals ( $sci(I)$ ) for these subtraces are  $(m_8, m_{11})$  and  $(m_{18}, m_{21})$  respectively. The duration for these intervals are calculated as follows:

$$\begin{aligned}\delta_{time}(\#_L(m_8), \#_L(m_{11})) &= \#_{time}(e_{2.9}) - \#_{time}(e_{2.7}) \\ &= 11:06:03.850 - 11:06:03.320 = 0.530 \\ \delta_{time}(\#_L(m_{18}), \#_L(m_{21})) &= \#_{time}(e_{2.18}) - \#_{time}(e_{2.15}) \\ &= 11:06:05.340 - 11:06:05.100 = 0.240\end{aligned}$$

Hence, we have  $Dur(L, SN, T', T_F) = [0.530, 0.240]$ .

■ **Performance Metric 8.7 — Waiting Time.** [*Multiset*]

*Description:* Calculates the time spent between enablement and start.

*Definition:*

$$\begin{aligned}Wait(L, SN, T', T_F) = \\ [\delta_{time}(\#_L(en^*(s)), \#_L(s)) \mid \mathcal{I} \in subtraces(L, SN, T') \wedge I \in \mathcal{I} \\ \wedge accept(I, T_F) \wedge (s, c) = sci(I)]\end{aligned}$$

*Example:* Consider the submodel  $T' = \{f1+s, f1+c, f2+s, f2+c\}$ . With the identity move enabler filter  $T_F = T$ , we have no subtraces in alignments  $\gamma_1$  and  $\gamma_{3,1}$  (no execution intervals for  $f1, f2$ ), and we have two subtraces in alignment  $\gamma_2$ :  $\{(m_8, m_{10}), (m_9, m_{11})\}$  and  $\{(m_{18}, m_{19}), (m_{20}, m_{21})\}$ . The smallest containing intervals ( $sci(I)$ ) for these subtraces are  $(m_8, m_{11})$  and  $(m_{18}, m_{21})$  respectively, and we have  $en^*(m_8) = m_6$  and  $en^*(m_{18}) = m_{16}$ . We calculate:

$$\begin{aligned}\delta_{time}(\#_L(m_6), \#_L(m_8)) &= \#_{time}(e_{2.7}) - \#_{time}(e_{2.6}) \\ &= 11:06:03.320 - 11:06:03.110 = 0.210 \\ \delta_{time}(\#_L(m_{16}), \#_L(m_{18})) &= \#_{time}(e_{2.15}) - \#_{time}(e_{2.14}) \\ &= 11:06:05.100 - 11:06:05.000 = 0.100\end{aligned}$$

Hence, we have  $Wait(L, SN, T', T_F) = [0.210, 0.100]$ .

■ **Performance Metric 8.8 — Sojourn Time.** [*Multiset*]

*Description:* Calculates the time spent between enablement and completion.

*Definition:*

$$\begin{aligned}Sojourn(L, SN, T', T_F) = \\ [\delta_{time}(\#_L(en^*(s)), \#_L(c)) \mid \mathcal{I} \in subtraces(L, SN, T') \wedge I \in \mathcal{I} \\ \wedge accept(I, T_F) \wedge (s, c) = sci(I)]\end{aligned}$$

*Example:* Consider the submodel  $T' = \{f1+s, f1+c, f2+s, f2+c\}$ . With the identity move enabler filter  $T_F = T$ , we have no subtraces in alignments  $\gamma_1$

and  $\gamma_{3,1}$  (no execution intervals for  $f1, f2$ ), and we have two subtraces in alignment  $\gamma_2$ :  $\{(m_8, m_{10}), (m_9, m_{11})\}$  and  $\{(m_{18}, m_{19}), (m_{20}, m_{21})\}$ . The smallest containing intervals ( $sci(I)$ ) for these subtraces are  $(m_8, m_{11})$  and  $(m_{18}, m_{21})$  respectively, and we have  $en^*(m_8) = m_6$  and  $en^*(m_{18}) = m_{16}$ . We calculate:

$$\begin{aligned} \delta_{time}(\#_L(m_6), \#_L(m_{11})) &= \#_{time}(e_{2.9}) - \#_{time}(e_{2.6}) \\ &= 11:06:03.850 - 11:06:03.110 = 0.740 \\ \delta_{time}(\#_L(m_{16}), \#_L(m_{21})) &= \#_{time}(e_{2.18}) - \#_{time}(e_{2.14}) \\ &= 11:06:05.340 - 11:06:05.000 = 0.340 \end{aligned}$$

Hence, we have  $Sojourn(L, SN, T', T_F) = [0.740, 0.340]$ .

■ **Performance Metric 8.9 — Cumulative Duration.** [*Multiset*]

*Description:* Calculates the total time spent per trace during executions.

*Definition:*

$$\begin{aligned} CulDur(L, SN, T', T_F) &= \\ &[\sum_{I \in \mathcal{I} \wedge accept(I, T_F) \wedge (s,c)=sci(I)} \delta_{time}(\#_L(s), \#_L(c)) \\ &| \mathcal{I} \in subtraces(L, SN, T') |] \end{aligned}$$

*Example:* Consider the submodel  $T' = \{f1+s, f1+c, f2+s, f2+c\}$ . With the identity move enabler filter  $T_F = T$ , we have no subtraces in alignments  $\gamma_1$  and  $\gamma_{3,1}$  (no execution intervals for  $f1, f2$ ), and we have two subtraces in alignment  $\gamma_2$ :  $\{(m_8, m_{10}), (m_9, m_{11})\}$  and  $\{(m_{18}, m_{19}), (m_{20}, m_{21})\}$ . We derive  $\delta_{time}(\#_L(m_8), \#_L(m_{11})) = 0.530$  and  $\delta_{time}(\#_L(m_{18}), \#_L(m_{21})) = 0.240$  (see also the *Duration* metric example above). Thus, for alignment  $\gamma_2$ , we calculate:  $0.530 + 0.240 = 0.770$ . Hence, we have  $CulDur(L, SN, T', T_F) = [0.770]$ .

■ **Performance Metric 8.10 — Own Duration.** [*Multiset, Subset-model*]

*Description:* Calculates the time spent during execution of submodel  $T'$  minus time spent during execution of subset  $T''$ .

*Definition:*

$$\begin{aligned} OwnDur(L, SN, T', T'', T_F) &= \\ &[\delta_{time}(\#_L(s), \#_L(c)) - (\sum_{(s',c') \in J} \delta_{time}(\#_L(s'), \#_L(c')) \\ &| \mathcal{I} \in subtraces(L, SN, T' \cup T'') \wedge I \in \mathcal{I} \wedge accept(I, T_F) \\ &\wedge (s, c) = sci(I \upharpoonright_{T'}) \wedge J = coi(I \upharpoonright_{T''}) |] \end{aligned}$$

*Example:* Consider the submodel  $T' = \{c+s, c+c\}$  with the subset-model  $T'' = \{f1+s, f1+c, f2+s, f2+c\}$ . With the identity filter  $T_F = T$ , we have no subtraces in alignments  $\gamma_1$  and  $\gamma_{3,1}$  (no execution intervals for  $c$ ), and we have two subtraces  $I \upharpoonright_{T'}$  in alignment  $\gamma_2$ :  $\{(m_6, m_{13})\}$  and  $\{(m_{16}, m_{23})\}$ . The corresponding subset subtraces  $I \upharpoonright_{T''}$  are:  $\{(m_8, m_{10}), (m_9, m_{11})\}$  and

$\{(m_{18}, m_{19}), (m_{20}, m_{21})\}$  respectively. For each subtrace, we compute the smallest containing interval  $sci(I|_{T'})$ , and for each subset subtrace we compute the set of condensed overlapping intervals  $coi(I|_{T''})$ . We derive:  $sci$  interval  $(m_6, m_{13})$  with  $coi$  subset interval  $\{(m_8, m_{11})\}$  and  $sci$  interval  $(m_{16}, m_{23})$  with  $coi$  subset intervals  $\{(m_{18}, m_{19}), (m_{20}, m_{21})\}$ . We calculate:

$$\begin{aligned}
& \delta_{time}(\#_L(m_6), \#_L(m_{13})) - (\delta_{time}(\#_L(m_8), \#_L(m_{11}))) \\
&= (\#_{time}(e_{2.11}) - \#_{time}(e_{2.6})) - ((\#_{time}(e_{2.10}) - \#_{time}(e_{2.7}))) \\
&= (11:06:04.070 - 11:06:03.110) - (11:06:03.900 - 11:06:03.320) \\
&= 0.960 - 0.580 = 0.380 \\
& \delta_{time}(\#_L(m_{16}), \#_L(m_{23})) - \\
& (\delta_{time}(\#_L(m_{18}), \#_L(m_{19})) + \delta_{time}(\#_L(m_{20}), \#_L(m_{21}))) \\
&= (\#_{time}(e_{2.19}) - \#_{time}(e_{2.14})) - \\
& ((\#_{time}(e_{2.16}) - \#_{time}(e_{2.15})) + (\#_{time}(e_{2.18}) - \#_{time}(e_{2.17}))) \\
&= (11:06:05.510 - 11:06:05.000) - \\
& ((11:06:05.210 - 11:06:05.100) + (11:06:05.340 - 11:06:05.280)) \\
&= 0.510 - (0.110 + 0.060) = 0.340
\end{aligned}$$

Hence, we have  $OwnDur(L, SN, T', T'', T_F) = [0.380, 0.340].0$

■ **Performance Metric 8.11 — Duration Efficiency.** [*Multiset, Subset-model*]

*Description:* Calculates the amount of work performed during execution of subset  $T''$  divided by the associated execution timespan of  $T'$ .

*Definition:*

$$\begin{aligned}
DurEff(L, SN, T', T'', T_F) = \\
& \left[ \frac{\sum_{(s', c') \in J} \delta_{time}(\#_L(s'), \#_L(c'))}{\delta_{time}(\#_L(s), \#_L(c))} \mid \mathcal{I} \in \text{subtraces}(L, SN, T' \cup T'') \right. \\
& \left. \wedge \mathbf{I} \in \mathcal{I} \wedge \text{accept}(\mathbf{I}, T_F) \wedge (s, c) = sci(\mathbf{I}|_{T'}) \wedge J = \mathbf{I}|_{T''} \right]
\end{aligned}$$

*Example:* Consider the submodel  $T' = \{c+s, c+c\}$  with the subset-model  $T'' = \{f1+s, f1+c, f2+s, f2+c\}$ . With the identity move enabler filter  $T_F = T$ , we have no subtraces in alignments  $\gamma_1$  and  $\gamma_{3,1}$  (no execution intervals for  $c$ ), and we have two subtraces  $I|_{T'}$  in alignment  $\gamma_2$ :  $\{(m_6, m_{13})\}$  and  $\{(m_{16}, m_{23})\}$ . We derive the corresponding subset subtraces  $I|_{T''}$ :  $\{(m_8, m_{10}), (m_9, m_{11})\}$  and  $\{(m_{18}, m_{19}), (m_{20}, m_{21})\}$  respectively. For each subtrace, we compute the smallest containing interval  $sci(I)$ . We derive  $sci$  intervals

$(m_6, m_{13})$  and  $(m_{16}, m_{23})$  respectively. We calculate:

$$\begin{aligned}
& \frac{\delta_{time}(\#_L(m_8), \#_L(m_{10})) + \delta_{time}(\#_L(m_9), \#_L(m_{11}))}{\delta_{time}(\#_L(m_6), \#_L(m_{13}))} \\
&= \frac{(\#_{time}(e_{2.9}) - \#_{time}(e_{2.7})) + (\#_{time}(e_{2.10}) - \#_{time}(e_{2.8}))}{\#_{time}(e_{2.11}) - \#_{time}(e_{2.6})} \\
&= \frac{(11:06:03.850 - 11:06:03.320) + (11:06:03.900 - 11:06:03.340)}{11:06:04.070 - 11:06:03.110} \\
&= \frac{0.530 + 0.560}{0.960} = 1.135 \\
& \frac{\delta_{time}(\#_L(m_{18}), \#_L(m_{19})) + \delta_{time}(\#_L(m_{20}), \#_L(m_{21}))}{\delta_{time}(\#_L(m_{16}), \#_L(m_{23}))} \\
&= \frac{(\#_{time}(e_{2.16}) - \#_{time}(e_{2.15})) + (\#_{time}(e_{2.18}) - \#_{time}(e_{2.17}))}{\#_{time}(e_{2.19}) - \#_{time}(e_{2.16})} \\
&= \frac{(11:06:05.210 - 11:06:05.100) + (11:06:05.340 - 11:06:05.280)}{11:06:05.510 - 11:06:05.000} \\
&= \frac{0.110 + 0.060}{0.510} = 0.334
\end{aligned}$$

Hence, we have  $DurEff(L, SN, T', T'', T_F) = [1.135, 0.334]$ . Observe that the first execution subtrace ( $\{(m_6, m_{13})\}$ ) has a duration efficiency (1.135) greater than 1, indicating that during this run the multi-threading was effective. The second execution subtrace ( $\{(m_{16}, m_{23})\}$ ) has a duration efficiency (0.334) less than 1, indicating that during this run the multi-threading was not effective.

### 8.4.3 Interpreting Metric Results

The metrics defined in Section 8.4.2 provide either a single *numerical* value or a *multiset* of values for a given submodel. Here, a numerical metric counts the occurrences of some property, for example, the number of executions of a specific submodel. A multiset metric derives a specific property for each execution or subrun of a submodel. Interpreting such results in the right way has a large impact on the actual performance analysis. In some cases, an average value is appropriate, in other cases, one can derive a histogram, heatmap, box-and-whisker plot, apply statistical significance tests, etc.

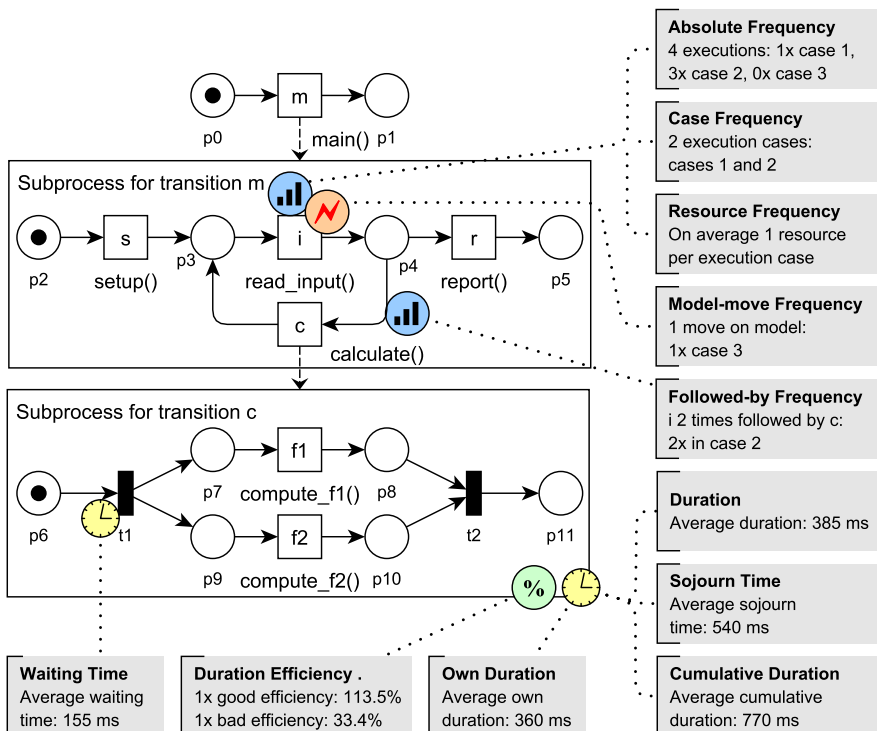
As we will show in Section 8.5, for a given model, usually an array of abstraction levels or submodels can be identified and analyzed in isolation. An effective strategy is to evaluate a performance metric for each and every identifiable submodel, and projecting the results across all elements in the (abstraction) hierarchy using, for example, a heatmap. This way, we enable the user to compare and evaluate all parts of the model at each abstraction level and across the abstraction levels as they need.

### 8.4.4 Example Metric Applications

In Section 8.4.2, we have detailed a collection of performance metrics taking into account model execution semantics. The detailed examples for each metric showed how the metric values are computed using the model from Figures 8.3 and 8.4 and event log from Table 8.1. In Figure 8.7, we put these example metric calculations into context, by projecting them back onto the model.

For example, the *duration efficiency* annotation on the subprocess for transition  $c$  show how this entire submodel performs in the various executions. Instead of looking at the individual executions of  $f1$  and  $f2$ , we now have brought the event data and performance metrics to the abstraction level of submodels.

Likewise, by using the execution semantics via the *followed-by* annotation, we have systematically calculated the frequency relations between activities  $i$  and  $c$ . In the evaluation on the next page, we will show how this *followed-by* metric can also be used in combination with cancellation model elements to determine the root causes for cancellation region triggers.



**Figure 8.7:** Hierarchical Petri net model from Figure 8.3 annotated with performance metrics. The used metric values are computed in detail in Section 8.4.2 and based on the event log data given in Table 8.1. The metrics express performance characteristics taking into account model execution semantics and using the modeled hierarchical subprocesses.

## 8.5 Evaluation

In this section, we evaluate and compare our technique against related implemented performance analysis techniques. The proposed framework and metrics are implemented in the *Statechart* plugin for the ProM framework, which will be explained in more detail in Chapter 10. Section 8.5.1 compares our approach against related performance analysis techniques. Section 8.5.2 evaluates the performance and scalability of our approach. For large, real-life case studies and tool UI using the hierarchy techniques, see Chapter 12.

### 8.5.1 Comparative Evaluation

In this section, we compare our technique against related, implemented performance analysis techniques. The goal of this evaluation is to investigate if and how the added notions of hierarchical submodels and control-flow based performance analysis can aid in performance analysis.

#### Methodology

Even though we focus on software system processes in this thesis, recall that our approach is generic enough to apply to any kind of operational process, including business processes. For this evaluation, we investigate the following event logs of two software systems and one business process:

- The *Alignments algorithm* implements the foundation introduced in Section 8.2. This piece of software is a nice, self-contained multi-threaded example: the algorithm computes multiple alignments in parallel.
- The *JUnit* library is a single-threaded software example with a rich hierarchical structure and many interacting classes.
- The *BPIC 2012* business log describes three subprocesses of a loan application process. In this evaluation, we only focus on the “A\_” subprocess, which contains a nice structured example of cancelation behavior.

We analyzed the above processes using the Yourkit Profiler [136] (for the software examples only), the Dotted Chart [172], “traditional” alignments [21] (our baseline), and the framework introduced in this chapter. We guide this performance analysis with the following *performance questions*:

1. Where are the main bottlenecks in the process in terms of duration?
2. What are the root causes for these bottlenecks and how can they be broken down across control flow, method calls, and class objects?
3. Where does multi-threading occur, and is this used efficiently?
4. Where do exceptions or cancelations occur, and which activities cause these cancelations?

#### Analysis Results

Figures 8.9 to 8.18 on pages 246 to 252 show the analysis results. Below, we will discuss each analysis technique in turn.



**Yourkit Profiler [136]** – Figure 8.8 shows two screens obtained after profiling the alignment algorithm with Yourkit. The stack-trace analysis (Figure 8.8a) shows various stacks with a breakdown of the total time spent in each method. The stacks are all rooted in `<All threads>`, with most stacks sharing a root in `Thread.run()`. From this view, we cannot derive which stacks are sequential (e.g., two different methods called one after another from `Thread.run()`) or which stacks are concurrently (i.e., executions from different threads).

Zooming in on the two stacks visible in Figure 8.8a, we recognize a stack with the method `BalancedDataXAlignmentPlugin.alignLog()` and another stack with the method `BalancedReplayResultProcess.call()`. Code inspection on these methods reveals that `BalancedDataXAlignmentPlugin.alignLog()` schedules a parallel computation, which is implemented by `BalancedReplayResultProcess.call()` on another thread. That is, these methods are the stack-trace points where control is switched between threads, but the profiler visualizes these points as two disconnected stack-traces. As a result, the stack-trace analysis view fails to show how the two threads are related control-flow wise (performance question 3). A direct consequence is that we cannot use this view to investigate performance characteristics across these thread-boundaries. In addition, the aggregate timing values also limit further investigation. Since we have no notion of thread interactions, we cannot reliably find bottlenecks (performance question 1) or examine an accurate breakdown of such bottlenecks (performance question 2).

Switching to the thread state analysis (Figure 8.8b), we get a view of all threads and their status. We can see at which point in time threads are active, and when they are blocked. From this view, we can deduce that the `BalancedReplayResultProcess.call()` method was executed from multiple threads. Moreover, we see that there is little interruption in the threads, indicating efficient concurrent computations (performance question 3). However, this view fails to link the thread status back to the main thread and the invoking method `BalancedDataXAlignmentPlugin.alignLog()` in terms of performance or control-flow.

For the JUnit software, the stack-trace analysis view shown in Figure 8.9 provides only a single stack-trace: everything is executed from the same thread. We can get a feeling for which methods account for most of the computational time, but these aggregated timing values again limit further investigation. For example, for the JUnit case, we see that the string manipulation libraries, reflection libraries and class loading account for most of the running time (performance question 1). However, we cannot tell the context in which these libraries are used. Hence, it is difficult to determine the root cause for these delays (performance question 2).

Since the BPIC 2012 case is only an event log and not a runnable program, we cannot investigate this process using the Yourkit Profiler.

**Dotted Chart [172]** – Figure 8.11 shows a dotted chart analysis on the alignment algorithm event log. Each dot in this scatterplot resembles a method

call or return, and the lines indicate the directly-follows relation between such calls and returns. Using time on the x-axis, the spacing of the dots in the zoomed-out view (Figure 8.11a) gives us an idea for where delays occur (performance question 1). For example, we see large gaps around the method `AbstractBalancedDataConformancePlugin.createMaxCostHelper()`, indicating a large delay. By using the logged thread-id information for coloring, the dotted chart can give us an idea for where thread-interaction occurs (performance question 3). For example, we see several colors/thread-ids clustered near the top right of the chart, where the `BalancedReplayResultProcess.call()` methods occur. However, since the dotted chart gives us only a high-level overview without any abstractions applied, it is difficult to see the (hierarchical) relations between different parts of the process. Zooming in or using the detailed tooltip event information (Figure 8.11b) does not help. That is, we get a good overall idea of how time is spent across the various methods, but it is difficult to identify root causes for bottlenecks and delaying paths (performance question 2). Likewise, we are able to explicitly see and investigate thread interactions (in contrast to the profiler approach above), but we cannot make any solid conclusions regarding multi-threading efficiency (performance question 3).

For the JUnit software event log, the dotted chart analysis view shown in Figure 8.10 provides a view similar to the one for the alignment algorithm event log. We can again identify some general patterns, such as the delays around the methods `AllDefaultPossibilitiesBuilder.junit4Builder()` and `AllDefaultPossibilitiesBuilder.junit3Builder()`. However, the same issue apply: the scatterplot gives a good overview and a feeling for what the data looks like, but it is difficult to identify root causes and efficiencies of various parts of the software process without any hierarchical control-flow context.

When applying the dotted chart analysis to the BPIC 2012 event log, we start to see the real potential of the dotted chart approach. Figure 8.12 shows the corresponding overview across all traces (case ids are plotted on the y-axis now). In this view, we can clearly see patterns across cases, such as arrival rates (visible as the diagonal line with starting activities), batched processing (visible as vertical lines), and more. When using the dotted chart overview for investigating the causes for the cancelation activities `A_DECLINED` and `A_CANCELLED`, we can use the colors to quickly identify where these activities occur and see examples of what preceded these activities. For example, we can see that `A_CANCELLED` is sometimes directly preceded by `A_PREACCEPTED`. But we cannot conclude that this order happens significantly frequently. Although this addresses performance question 4 to some degree, this does not give us a complete overview at a glance.

Although we do not have the proper event logs at the time of writing, we hypothesize that if one would record an event log for a software system like a web server, a dotted chart analysis would be highly valuable. For example,

effects like load on a web server, and the corresponding performance impacts, would be visible in a dotted chart, just like the global cross-case patterns in Figure 8.12. However, when investigating more detailed questions, like the performance questions posed in this analysis, the dotted chart is less suited.

**Alignments [21]** – Figure 8.13 shows a part of the *replay for performance* result on the alignment algorithm event log produced by the “traditional” alignments algorithm itself [21]. The model used for this alignment replay is a flattened Petri net version of the hierarchical model mined using the techniques from Chapter 6. Only a part/snippet of the entire model is shown here; the complete, flattened result is too large to show. In this *replay for performance* result, the transitions and places of the Petri net are highlighted using a heatmap based on durations and waiting times. These durations and waiting time are only computed between directly succeeding transitions. Since each method start/call and complete/return is a transition, and methods may be nested according to their call relation, we can only investigate delays between directly-succeeding calls and returns. That is, there is no way to investigate, for example, the duration of a method  $m$  if that method  $m$  invokes other methods  $m'$ . As a result, very few parts in the model actually indicate any performance issues. That is, most transitions and places are colored the same, and no significant performance differences are highlighted (performance questions 1 and 2). Only some low-level methods like `AbstractBalancedDataConformancePlugin.createMaxCostHelper()` are highlighted, but these results cannot be trusted for aforementioned reasons. By closely inspecting the control-flow of the model, we can locate the methods `BalancedDataXAlignmentPlugin.alignLog()` and `BalancedReplayResultProcess.call()`. Hence, using the process models together with the replay approach, we can determine where the multi-threading occurs. However, we cannot derive a duration or multi-threaded efficiency (performance question 3).

For the JUnit software, the *replay for performance* result in Figure 8.14 shows similar issues. The model used for this alignment replay is a flattened Petri net version of the hierarchical model mined using the techniques from Chapter 6. Again, since each method start and end is a transition, very few parts in the model actually indicate any performance issues. Upon further inspection of the entire model, we find possible performance issue indicators: the low-level methods `AllDefaultPossibilitiesBuilder.junit4Builder()` and `AllDefaultPossibilitiesBuilder.junit3Builder()` show a relatively large delay (performance question 1). However, without an explicit hierarchical call relation, it is difficult to place the above results in context and determine root causes (performance question 2).

When applying the replay for performance analysis to the BPIC 2012 event log, we get better results. Figure 8.12 shows the corresponding *replay for performance* result. The model used for this alignment replay is a flattened Petri

net version of the cancellation model mined using the techniques from Chapter 7. We used the activities *A\_DECLINED* and *A\_CANCELLED* for our cancellation error oracle. Since this model has little hierarchy, the replay technique is much better suited for this case, and we can get reliable duration results (performance question 1). That is, we can reliably investigate where most of the time is spent in this model. In addition, we can determine which activities lead up to the problematic areas, allowing for some root-cause analysis (performance question 2). However, when using a frequency overlay to investigate the cancellation behavior, we again start to see the limit of the replay technique. Yes, we can determine how often our cancellation activities *A\_DECLINED* and *A\_CANCELLED* are executed, and what might have preceded these activities. However, to gain any more detailed, non-local information, we would have to switch to a log-view and inspect the result event by event. Hence, it is difficult to explain the causes for the above cancellation activities on a model-centric view (performance question 4).

**Hierarchical Performance Analysis (this chapter)** – Figure 8.16 shows a part of the results for the alignment algorithm event log produced by the hierarchical performance analysis techniques from this chapter. By leveraging the subprocesses and applying our metrics at every level in this hierarchy, we get a very accurate breakdown of the performance across various parts of the software process. For example, the process tree in Figure 8.16a shows a *duration* breakdown across different abstraction levels. Using this visualization, we can quickly see at a high abstraction level where most of the time is spent in this model (performance question 1). Using this annotated model, we can quickly spot where the alignment computation start, and which methods during the computation contribute the most to the running time. See for example the indicated path A, B, C in Figure 8.16a. Next, we can interactively unfold the highlighted submodels and explore lower levels in the model. This way, we get a performance breakdown of the duration in the context of both the call relation (hierarchy) and the control flow (choices, loops, recursions, etc.). For example, at the lower levels in the path from Figure 8.16a, we see that most of the running time is spent in `AbstractBalancedDataConformancePlugin.createMaxCostHelper()` at the beginning of the process and in `BalancedReplayResultProcess.call()` later during the actual alignment computations itself. Hence, this performance breakdown allows a user to reliably investigate root causes for performance issues (performance question 2).

Likewise, a *duration efficiency* overlay, as shown in Figure 8.16b, quickly shows the user where the modeled software is single threaded (indicated by neutral white-yellow color), and where multi-threading occurs (indicated by green or red). Using such an overlay, we can see the relation between threads as we go down the call relation hierarchy. We recognize the `BalancedDataXAlignmentPlugin.alignLog()` and `BalancedReplayResultProcess.call()` methods we saw in the Yourkit

Profiler, but now the control-flow threads are linked together. Furthermore, the heatmap based on duration efficiency highlights whether the multi-threading is efficient, indicated by green for  $> 100\%$ , or where computational time is potentially lost, indicated by red for  $< 100\%$  (performance question 3). In our case, we see an efficiency of  $161,8\%$ , so the multi-threading is efficient. By investigating the performance metrics around the corresponding submodels, we can further investigate how efficient this part of the software is. Observe that the analyses in Figure 8.13 and Figure 8.16 use the same model language and event data. However, by leveraging the hierarchical structure in our performance analysis, we gain much more insight at various levels of granularity in a user-friendly way, even for large models.

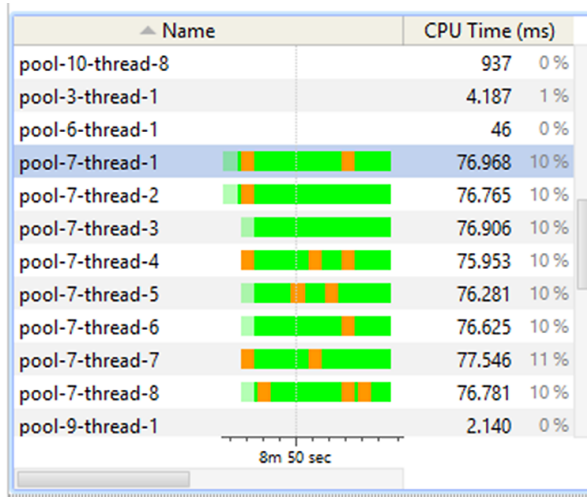
Figure 8.17 shows a message sequence diagram derived for the JUnit software event log. The vertical lifelines in this model are based on the class names recorded in the event log and the model is overlaid with the *Duration* metric. Using a combination of class names, control-flow, and call-relation, we get an insightful breakdown of performance across multiple perspectives. For example, Figure 8.17 shows that in this part of the model, most time is spent in the class *Request* (B) as a result of invocation *JUnitCommandLineParser.createRequest()* (A). Such a view can aid in determining root causes in the context of various architectural structures such as classes, components, etc. (performance question 2).

Figure 8.18 shows a statechart derived for the BPIC 2012 event log. We used the techniques from Chapter 7 and marked the activities *A\_DECLINED* and *A\_CANCELLED* in our cancellation error oracle. Note how most of the model is wrapped in a cancellation region. We overlay this model with the *Followed-by Frequency* metric configured to show how often an activity is followed by a cancellation region trigger. The effective result is that activities are highlighted when they often trigger the modeled cancellation region. We can clearly see that the activities indicated by A, B, and C account for most of the *A\_DECLINED* and *A\_CANCELLED* cancellation triggers. Hence, by using the semantics of the model, we are able to determine where these cancellations occur and which activities cause these cancellations (performance question 4).

Overall, we can conclude that the added expressiveness of our hierarchical performance metrics are beneficial for various performance questions. Especially when analyzing larger models with hierarchical constructs such as sub-processes and cancellation region, one needs to 1) include the semantics in the model to get more meaningful and reliable performance results, and 2) leverage the hierarchical constructs to navigate the large amount of performance results. The interaction with hierarchical notions, guided by the hierarchical performance analysis results, proved essential for understanding large, complex (software) behavior.

Name	Time (ms)	Samples
<All threads>	1,703 100%	69 100%
java.lang.Thread.run()	1,515 89%	54 78%
FutureTask.java:262 org.processmining.plugins.balancedconformance.BalancedReplayResultProcessor.call()	1,203 71%	2 3%
BalancedReplayResultProcessor.java:11 org.processmining.plugins.balancedconformance.BalancedReplayResultProcessor.call()	1,203 71%	2 3%
BalancedReplayResultProcessor.java:38 org.processmining.plugins.balancedconformance.DataAlignmentTraceProcessor.computeDat	1,203 71%	2 3%
SwingWorker.java:296 org.processmining.ui.statechart.workbench.discovery.DiscoveryWorkbenchController\$30.donInBackground()	140 8%	1 1%
DiscoveryWorkbenchController.java:1 org.processmining.ui.statechart.workbench.discovery.DiscoveryWorkbenchController\$30.donInBack	140 8%	1 1%
DiscoveryWorkbenchController.java:986 org.processmining.recipes.statechart.RecipeProcess.getArtifact(RecipeArtifact)	140 8%	1 1%
...		
AlignLog2Tree.java:164 org.processmining.algorithms.statechart.align.AlignLog2Tree.performAlignment(XLog, IEPTree)	140 10%	1 3%
AlignLog2Tree.java:186 org.processmining.plugins.balancedconformance.BalancedDataXAlignmentPlugin.alignLog(Prog	140 10%	1 3%
BalancedDataXAlignmentPlugin.java:152 org.processmining.plugins.balancedconformance.AbstractBalancedDataCor	140 10%	1 3%
AbstractBalancedDataConformancePlugin.java:297 java.util.concurrent.ThreadPoolExecutor.awaitTermination(o	140 10%	1 3%

(a) Stack-trace analysis. Highlighted are the methods `BalancedDataXAlignmentPlugin.alignLog()` in the main thread and `BalancedReplayResultProcess.call()` in child threads. These stacks show a breakdown of the total time spent in each method. However, from this view, we cannot derive which stacks are sequential or which stacks are concurrently: the stack-trace analysis view fails to show how the two threads are related control-flow wise.

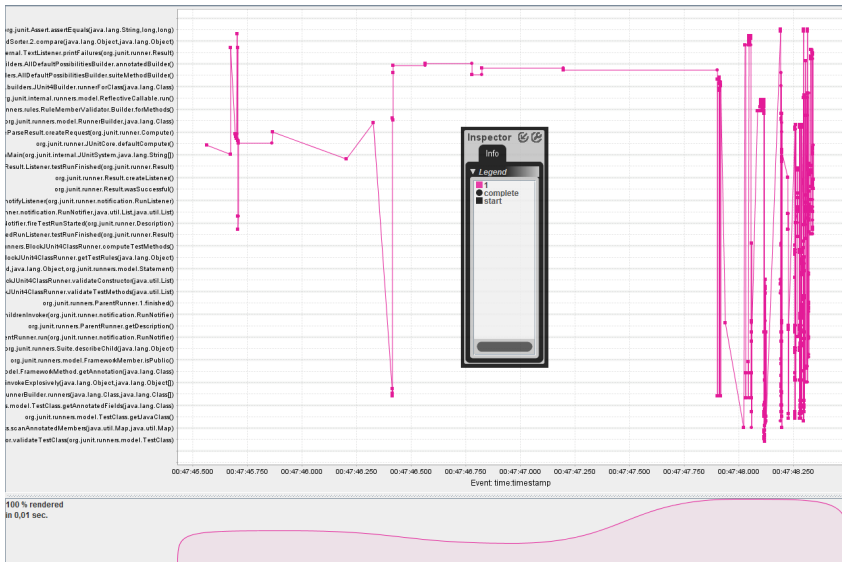


(b) Corresponding thread state analysis. This view shows all threads and their status, indicating when threads are active and when they are blocked. We get an idea of the efficiency of concurrent computations, but we cannot link these results back to the invoking methods in terms of performance or control-flow.

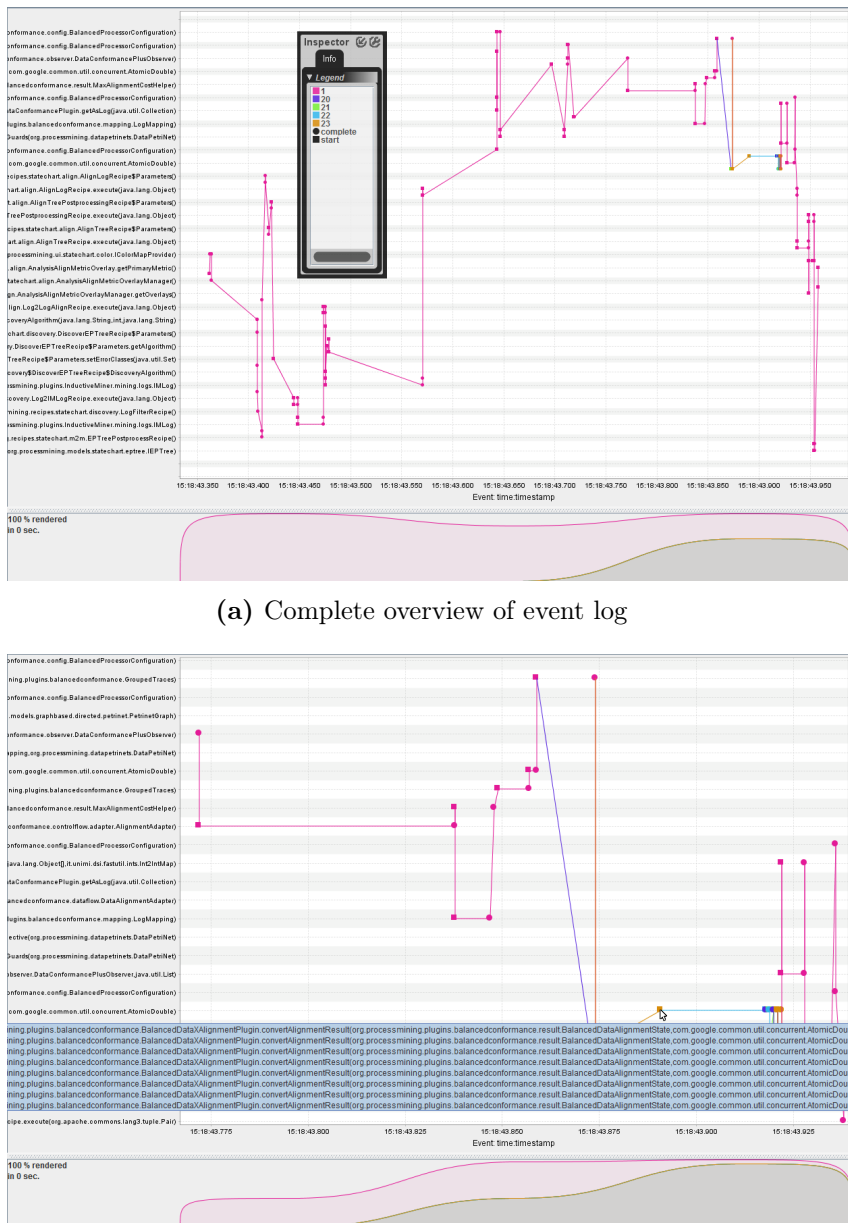
**Figure 8.8:** The Yourkit Profiler [136] results on the alignments algorithm. The two views shown each tell a part of the story (i.e., control-flow, computation times, or thread efficiency), but do not allow to link the results together.

Name	Time (ms)	Avg. Time (...)	Own Time (ms)	Invocation C...
<All threads>	143	100 %		
org.junit.runner.JUnitCore.main(String[])	128	90 %	128	1
CalculatorTest.evaluatesExpression()	1	1 %	1	1
Calculator.evaluate(String)	0.7	0 %	0.7	< 0.1
java.lang.ClassLoader.loadClass(String)	0.6	0 %	0.3	0.6
java.lang.StringBuilder.append(int)	< 0.1	0 %	< 0.1	< 0.1
java.lang.Class.getDeclaredFields()	< 0.1	0 %	< 0.1	< 0.1
java.lang.StringBuilder.append(char)	< 0.1	0 %	< 0.1	< 0.1
java.lang.StringBuilder.append(String)	< 0.1	0 %	< 0.1	< 0.1
java.lang.Class.getName()	< 0.1	0 %	< 0.1	< 0.1
java.lang.StringBuilder.<init>()	< 0.1	0 %	< 0.1	< 0.1
java.lang.ClassLoader.checkPackageAccess(Class,	< 0.1	0 %	< 0.1	< 0.1
Calculator.<init>()	< 0.1	0 %	< 0.1	< 0.1
java.lang.String.equals(Object)	0	0 %	0	0
java.lang.String.isEmpty()	0	0 %	0	0
java.lang.String.replace(char, char)	0	0 %	0	0
java.lang.StringBuilder.toString()	0	0 %	0	0
org.junit.Assert.assertEquals(long, long)	0	0 %	0	0
CalculatorTest.<init>()	< 0.1	0 %	< 0.1	< 0.1
sun.launcher.LauncherHelper.checkAndLoadMain(boolea	14	10 %	14	1
sun.launcher.LauncherHelper.makePlatformString(boole	0.3	0 %	0.1	0.3

**Figure 8.9:** The Yourkit Profiler [136] result on the JUnit software. We can get a feeling for which methods account for most of the computational time, but due to the aggregated values and lack of control-flow context it is difficult to determine the root cause for the observed delays.



**Figure 8.10:** The Dotted Chart [172] results on the JUnit event log. Each dot in this scatterplot is an event, with time on the x-axis and activities on the y-axis. The colors indicate thread-ids, the shapes the lifecycle-transition, and the lines indicate the directly-follows relation. This dotted chart gives us a good high-level overview without any abstractions applied, but it is difficult to see the detailed control-flow relations and performance root causes.

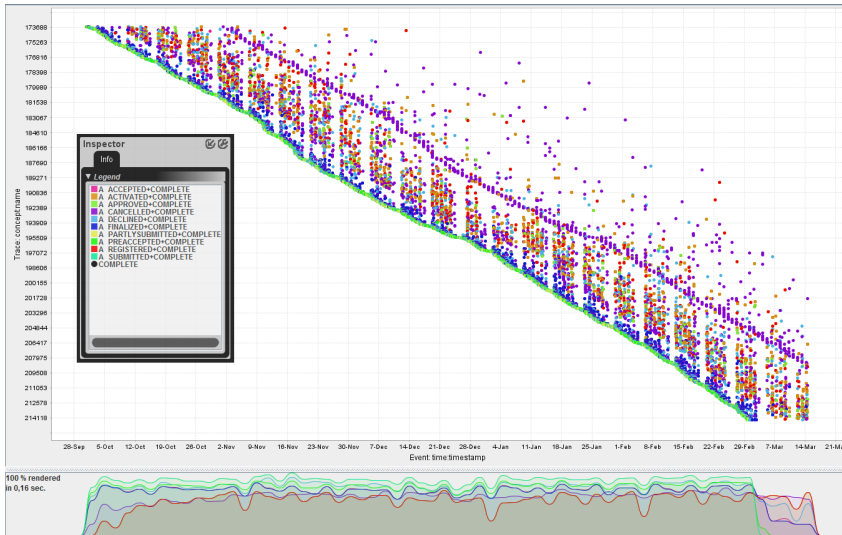


(a) Complete overview of event log

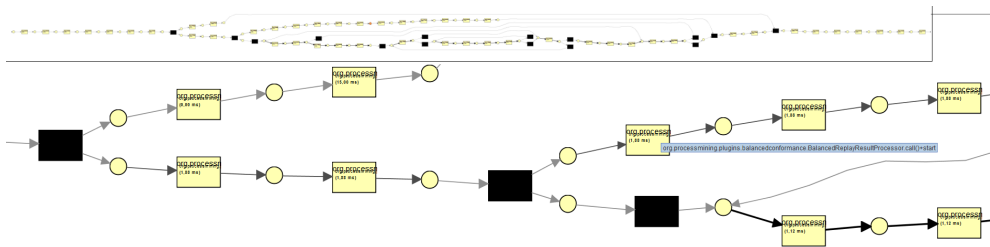
(b) Zoomed in view on top right part.

**Figure 8.11:** The Dotted Chart [172] results on the alignments event log. Each dot in this scatterplot is an event, with time on the x-axis and activities on the y-axis. The colors indicate thread-ids, the shapes the lifecycle-transition, and the lines indicate the directly-follows relation. These dotted charts give us a good high-level overview without any abstractions applied, but it is difficult to see the detailed control-flow relations and performance root causes.

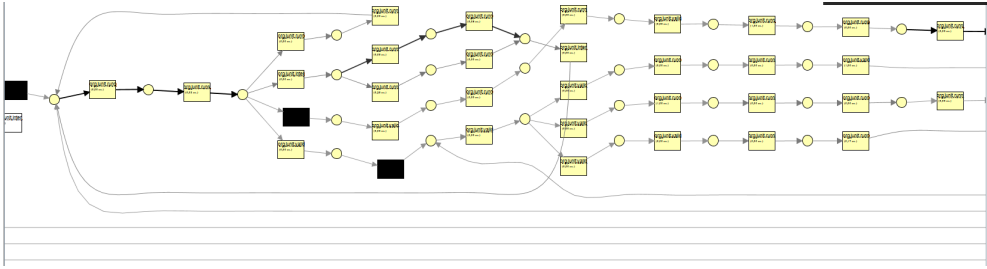




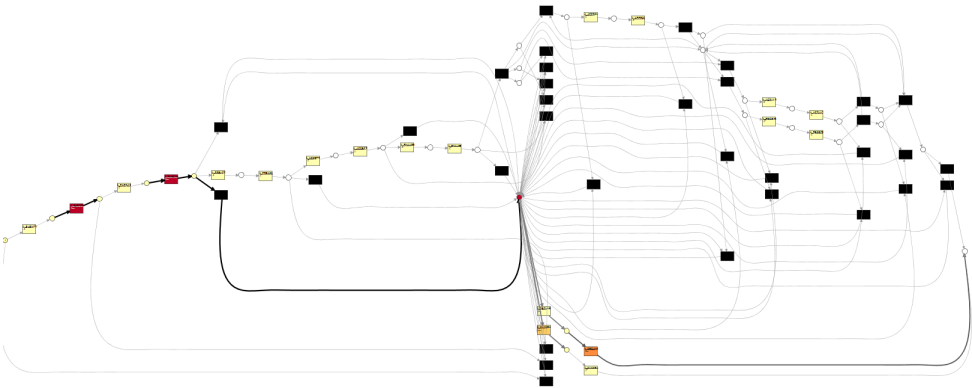
**Figure 8.12:** The Dotted Chart [172] results on the BPIC 2012 event log. Each dot in this scatterplot is an event, with time on the x-axis and case id on the y-axis. The colors indicate activity names and the shapes the lifecycle-transition. We can clearly see patterns across cases, such as arrival rates (visible as the diagonal line with starting activities), batched processing (visible as vertical lines), and more.



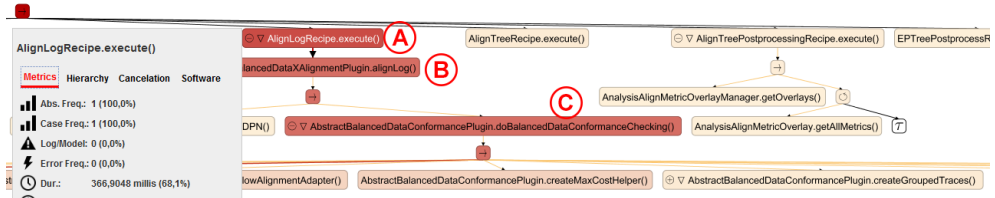
**Figure 8.13:** Snippet of the *Replay for Performance Alignment* [21] result on the alignments event log. The transitions are colored according to the sojourn time, the places are colored according to the waiting time. Only a part/snippet of the entire model is shown here; the complete, flattened result is too large to show. Very few parts in the model actually indicate any performance issues: most transitions and places are colored the same, and no significant performance differences are highlighted.



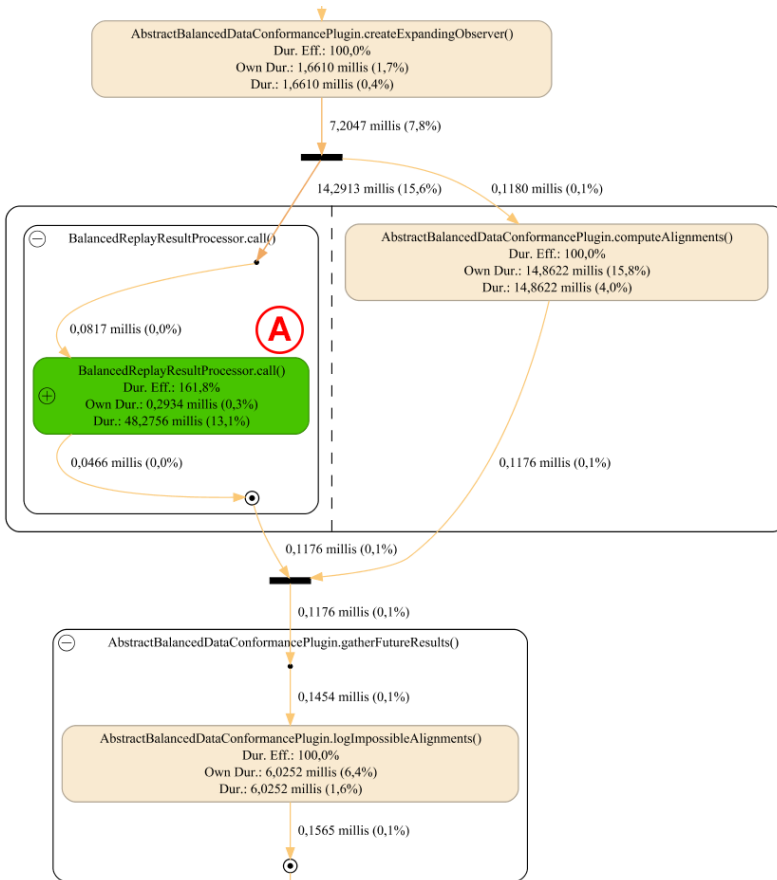
**Figure 8.14:** Snippet of the *Replay for Performance Alignment* [21] result on the JUnit event log. The transitions are colored according to the sojourn time, the places are colored according to the waiting time. Only a part/snippet of the entire model is shown here; the complete, flattened result is too large to show. Very few parts in the model actually indicate any performance issues: most transitions and places are colored the same, and no significant performance differences are highlighted.



**Figure 8.15:** Snippet of the *Replay for Performance Alignment* [21] result on the BPIC 2012 event log. The transitions are colored according to the absolute frequency, the places are colored according to the waiting time. We can reliably investigate where most of the time is spent in this model. In addition, we can determine which activities lead up to the problematic areas, allowing for some root-cause analysis. However, to gain any more detailed, non-local information, we would have to switch to a log-view and inspect the result event by event.

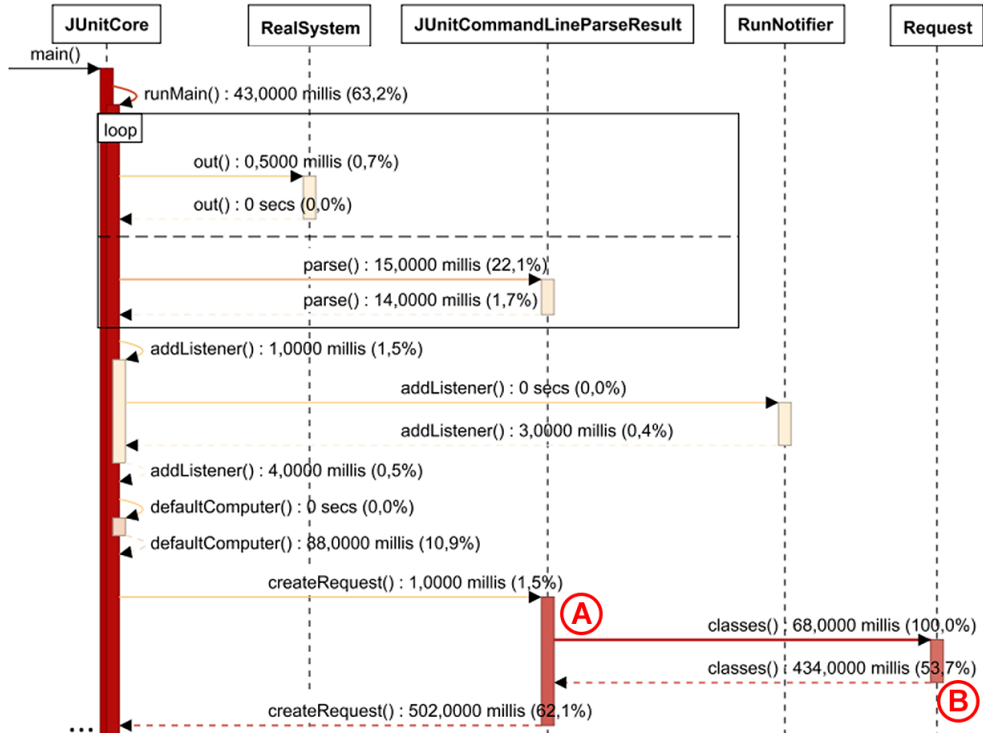


(a) A *process tree* overlaid with the *Duration* metric. By leveraging the subprocesses and applying our metrics at every level in this hierarchy, we get a very accurate breakdown of the performance across various parts of the software process. The indicated path A, B, C show a duration breakdown of the alignment computation.

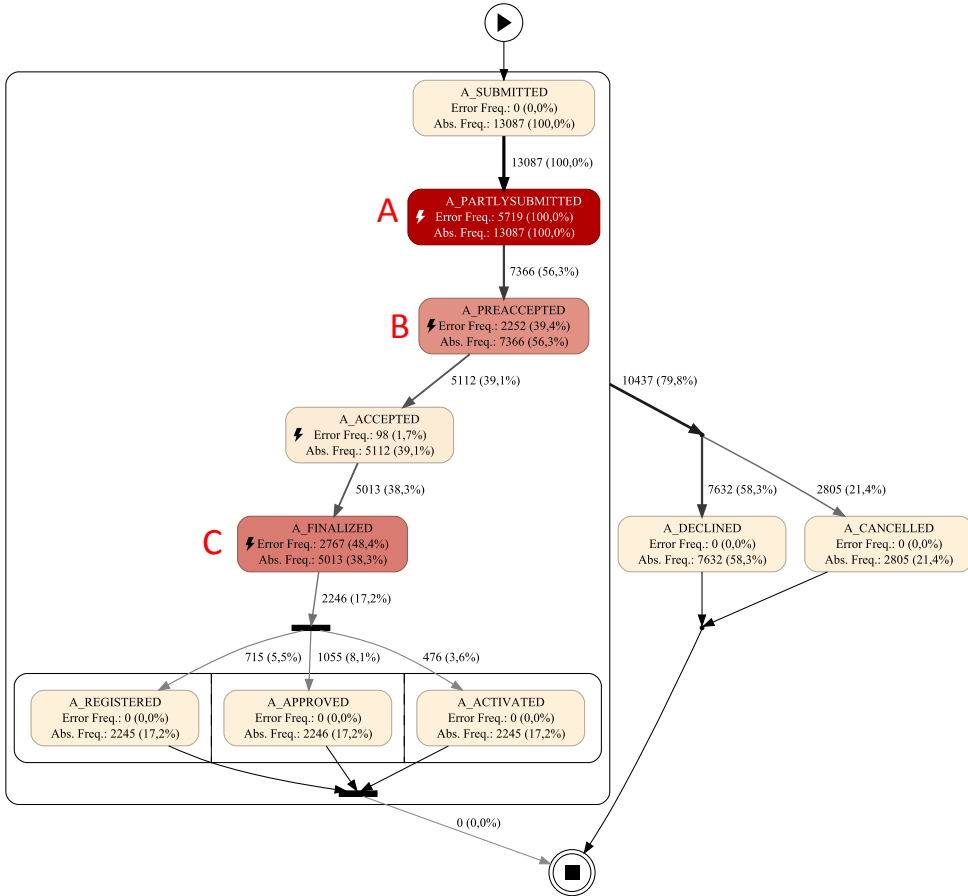


(b) A *statechart* overlaid with the *Duration Efficiency* metric. The heatmap based on duration efficiency highlights whether the multi-threading is efficient, indicated by green for  $> 100\%$ . We see an efficiency of 161,8%, so the multi-threading is efficient.

**Figure 8.16:** The *hierarchical performance analysis* results on the alignments event log. Shown is the same model using two different visualizations and two different hierarchical performance metric overlays.



**Figure 8.17:** The *hierarchical performance analysis* results on the JUnit event log. Shown is a *message sequence diagram* (the vertical lifelines are based on the class names) overlaid with the *Duration* metric. Using a combination of class names, control-flow, and call-relation, we get an insightful breakdown of performance across multiple perspectives. In this part of the model, most time is spent in the class *Request* (B) as a result of invocation *JUnitCommandLineParser.createRequest()* (A).



**Figure 8.18:** The *hierarchical performance analysis* results on the BPIC 2012 event log. Shown is a *statechart* overlaid with the *Followed-by Frequency* metric configured to show how often an activity is followed by a cancellation region trigger (*Error Freq.*). That is, activities are highlighted when they often trigger the modeled cancellation region. We can clearly see that the activities indicated by A, B, and C account for most of the *A\_DECLINED* and *A\_CANCELLED* cancellation triggers.

## 8.5.2 Performance and Scalability Evaluation

In this section, we evaluate the computational overhead of applying our hierarchical performance analysis on top of the alignment framework.

### Computational Phases

To accurately measure the overhead of calculating hierarchical performance metrics, we will look at an end-to-end computation time. That is, we will consider the time from discovering a process model to an actual visualization of the model plus various performance metric annotations. In this end-to-end setup, we recognize the computational phases below, in order:

1. During the **Process Discovery** phase, an event log is processed, and a process tree is derived. In this phase, the algorithms from Chapters 6 and 7 are applied.
2. After discovery the **Aligning Log and Model** phase is performed. During this phase, the process tree is translated to a flattened Petri net model, and the *alignment* algorithm (Section 8.2) is applied.
3. After aligning all traces in the event log, the **Alignment Projection** phase projects the results back onto the process tree. Essentially, this phase precomputes several data-structures to implement the *enabler moves*, *execution intervals*, and *execution subtraces* notions from Section 8.3.
4. Next, in the **Graph Layout** phase, the actual visualization is determined and computed. In this phase, the process tree is converted to a statechart graph for which a layout is computed using Graphviz DOT [32]. In this step, some levels of the hierarchy are excluded based on user input, e.g., some submodels are “collapsed” to hide details, see also Chapter 10.
5. Lastly, the **Compute Metrics** phase evaluates the metrics from Section 8.4 for all visible model elements and submodels. Note that the data-structures from the *Alignment Projection* phase can be reused when switching between various visualizations and metrics.

Note that in the *Aligning Log and Model* phase, no notion of hierarchy or filtering is introduced yet, we just align the complete event log on the complete, flattened model. See also Section 8.2.

### Measurement Setup

All of the algorithms evaluated in this comparison are invoked from a Java benchmark setup under the same operating conditions. For these experiments we used a laptop with an i7-4700MQ CPU @ 2.40 GHz, Windows 8.1 and Java SE 1.7.0 67 (64 bit) with 12 GB of allocated RAM.

For the running time, we measured the average running time and associated 95% confidence interval over 30 micro-benchmark executions, after 10 warmup rounds for the Java JVM. The time for loading event logs or Java classes is excluded from the measurements.

As input, we selected the JUnit 4.12 software event log [108]. For this event log, the JUnit 4.12 software [67] was executed once, using the example input found at [40]. The resulting event log has one trace of 946 events and consists of 182 unique activities (including lifecycles) across 25 levels of hierarchy. This large hierarchical depth allows us to investigate the influence of the number of hierarchical constructs on the overall performance.

## Results

Table 8.2 shows the average running times for the end-to-end performance metric computation across the different computational phases for various model sizes. In Figure 8.19, the stacked bars show how the total running time is composed of the different phases. Note that the computational phases are stacked from bottom to top in the order detailed on the previous page, starting with *Process Discovery* on the bottom.

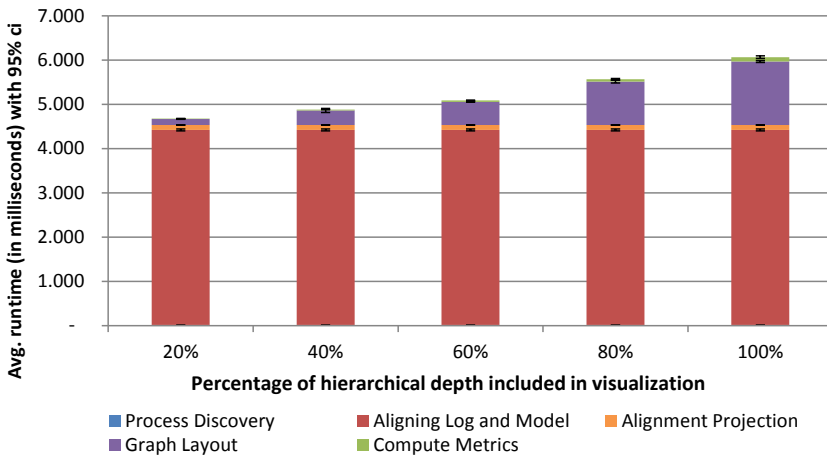
The phases *Process Discovery*, *Aligning Log and Model*, and *Alignment Projection* are not dependent on the percentage of hierarchical depth included in the visualization, i.e., these phases are always computed over the entire model. Furthermore, for these three phases, the running time for *Process Discovery* is negligible (between 0.21% and 0.27% or around 13 ms), while the running time for *Aligning Log and Model* (i.e., the alignments algorithm) dominates the entire computation time (between 72.66% and 94.20% or around 4 seconds). Note how the *Alignment Projection*, although computed over the entire model, only accounts for a small, affordable fraction of the total running time (between 1.83% and 2.37% or around 111 ms).

The phases *Graph Layout* and *Compute Metrics* are dependent on the percentage of hierarchical depth included in the visualization. We immediately see that the running time for computing the layout of the resulting statechart graphs increases with the model size. This makes sense: the more levels of hierarchical depth are included, the more graph elements for which a layout needs to be computed. Note however that the layout computation of these graph can take up a considerable portion of the computation time (ranging from 2.88% to 23.65% or between 135 ms and 1,4 seconds). In comparison, the running time of the actual metric computation is negligible (ranging from 0.28% to 1.66% or between 13 ms and 101 ms).

Overall, when looking at the end-to-end performance metric computation time, we can conclude that the added computational overhead of our hierarchical performance metrics (i.e., phases *Alignment Projection* and *Compute Metrics*) is essentially negligible. Most of the computation time is spent on the alignment and graph layout algorithm (i.e., phases *Aligning Log and Model* and *Graph Layout*). However, the added benefits of our hierarchical performance metrics are significant, as we will discuss in the next section.

**Table 8.2:** Average running time for the end-to-end performance metric computation across the different computational phases for various model sizes. Given are the average running times in milliseconds over 30 runs (left) and the percentage of the total running time (right) for each phase and model size.

	Percentage of hierarchical depth included in visualization									
	20%		40%		60%		80%		100%	
Process Discovery	13	0.27%	13	0.26%	13	0.25%	13	0.23%	13	0.21%
Aligning Log and Model	4,408	94.20%	4,408	90.30%	4,408	86.63%	4,408	79.17%	4,408	72.66%
Alignment Projection	111	2.37%	111	2.27%	111	2.18%	111	1.99%	111	1.83%
Graph Layout	135	2.88%	330	6.76%	525	10.31%	983	17.66%	1,435	23.65%
Compute Metrics	13	0.28%	20	0.40%	32	0.63%	53	0.96%	101	1.66%
Total	4,679 ms		4,881 ms		5,088 ms		5,568 ms		6,067 ms	



**Figure 8.19:** Stacked bar chart showing a breakdown of the running times for the end-to-end performance metric computation across the different computational phases for various model sizes. Given are the average running times in milliseconds over 30 runs, with a 95% confidence interval shown for each bar.

## 8.6 Conclusion and Open Challenges

In this chapter, we introduced a hierarchical approach to performance analysis, taking into account notions such as subprocesses and cancellation behavior (Contribution 4). Building upon the work on alignments, we introduced a *hierarchical performance analysis framework* for semantic-aware execution subtraces, taking into account model execution semantics. These *execution subtraces* are based on the parts of an aligned log that correspond to a given submodel. Based on this analysis framework, we formalized a selection of existing and novel performance metrics. We evaluated the proposed framework against existing analysis approaches and showed the benefits of the added ex-



pressiveness of our hierarchical performance metrics. Moreover, the interaction with hierarchical notions, guided by the hierarchical performance analysis results, proved essential for understanding large, complex (software) behavior. In addition, we showed that the added computational overhead of our hierarchical performance metrics, compared to the end-to-end performance metric computation time, is essentially negligible.

Given the hierarchical performance analysis framework and evaluation results presented in this chapter, there are several interesting directions for future research.

■ **Future Work 8.1 — Extended Lifecycle-Aware Performance Analysis.** The analysis framework introduced in this chapter includes the notion of *start* and *complete* events as well as the notion of *enabledness* available via models and their execution semantics. However, as already stated in Section 2.3.1 on page 43, the *transactional lifecycle model* [8] supports many more detailed transitions, such as schedule, pause, resume, abort, and reassign. Some of these notions have already been investigated in, for example, the work on *queue mining* [169]. However, to the best of our knowledge, no integrated approach has been defined. We propose to investigate and integrate the above notions into an analysis framework in a similar fashion as we have done for the notions like *enabledness*, enabling richer metrics and performance analysis techniques.

■ **Future Work 8.2 — Multi-Threaded/Multi-Process Software Analysis.** In Future Work 6.3 and 6.4 on page 159 we already argued that support is needed for rediscovering multi-threaded/multi-process forks and joins as well as multi-instance constructs. After all, there is enough non-causal information in running software, like thread and process identification, to suggest where behavior happens concurrently. Similar available information can be used to enhance the multi-threaded/multi-process performance analysis. For example, the information from a thread state analysis (see the Yourkit Profiler result in Figure 8.8b) can and should be used in a model-centric performance analysis. In this example, one should be able to discover a model with known points where multi-threading occurs, and be able to overlay such a model with for example a metric indicating when and where a lot of thread-blocking occurs. This way, one could combine the knowledge of concepts like a thread state analysis in the richer context of a multi-threaded aware control-flow model.

■ **Future Work 8.3 — Hierarchy-Aware Graph Layout Computation.** As discussed in Section 8.5.2 and shown in Figure 8.19, the main bottlenecks for our hierarchical performance analysis are the alignments algorithm and the graph layout algorithm. In Future Work 6.6 on page 161, we already suggested to investigate the use of hierarchical structures for computing alignments to improve

performance. Similarly, future research should look into leveraging the structured information available in hierarchical (process tree) models to improve the scalability of computing graph layouts. We already suggested to use the available structured information for the sake of a robust and deterministic layout in Future Work 6.7 on page 161 and Future Work 7.4 on page 205. However, we foresee that there is also an opportunity to greatly improve the computation time needed for calculating the layout for such highly structured end-to-end process models.

■ **Future Work 8.4 — Alignment Approximations.** The Inductive Miner discovery framework used throughout this thesis splits logs until a base case applies. The net result of this divide and conquer approach is that we get an approximate mapping between discovered transitions and events in the event log for free. Note that in the case of fallbacks, frequency filtering, and the like, we cannot trust this mapping completely. However, we hypothesize that the discovered mapping is still an accurate approximation of the mapping between events and transitions that the alignment algorithm searches for. Since in many cases one wishes to both discover a model and analyze various performance aspects for this discovered model, the approximate mapping we get for free could potentially reduce the computation time for an end-to-end process analysis (see also Section 8.5.2). Therefore, future work should investigate 1) the accuracy of the event-transition mapping produced by this type of process discovery, and 2) investigate how such approximated mappings can be used for computing alignments and/or performance metrics.

■ **Future Work 8.5 — Include Source Code and other Static Artifacts.** When logging and analyzing software systems, we usually have some form of traceability back to the source code, see also Section 5.3.3 on page 105. In Chapter 9, we will discuss this type of tracabilty in more detail. When analyzing models obtained from such software systems, this link back to the source code can and should be used. A trivial use is to allow the user to open and view the location (method, code line, etc.) in the source code corresponding to a model element, as we will show in Chapter 10. However, this traceability can also be used in the opposite direction. For example, one could overlay the discovered software process model with metrics derived from the source code. A typical example is to highlight parts in the model with high complexity (for some source code based complexity metric), indicating where in the control-flow errors or mistakes are more likely. Taking this traceability link one step further, one could link the discovery and results back to various design and architecture models. One useful analysis could be to use the techniques in the chapter to replay event logs onto parts of a design or architecture model, showing which components or classes were instantiated at which time during execution.

**■ Future Work 8.6 — Include Conformance Results in Performance Analysis.**

As stated in Section 8.2, the observed behavior in the log does not always precisely align with the behavior allowed by the net. As a result, log moves and model moves are introduced into an alignment, hinting at conformance issues. Consequently, performance metrics calculated in the vicinity of such conformance issues are less reliable. For example, what does the waiting time between two synchronous moves mean when several non-conforming log and model moves happen in between? Therefore, one should investigate the notion of reliability of performance metrics, combining the worlds of conformance and performance analysis. Such a combination can help by informing the user when metrics are less reliable due to non-conforming behavior, thus aiding the users and preventing them from making the wrong conclusions.

**■ Future Work 8.7 — Metric-Guided Model Exploration.** So far, performance metrics are only used as static overlays. The user analyzing a model tries to navigate and find points of interest, and queries more detailed information once an interesting point is found. Especially in larger, hierarchical models, this task of finding points of interest becomes more difficult. Performance metrics could help the user here by taking a more active role in the performance analysis. For example, performance metrics can be used to automatically focus the user view on certain parts of the model, suggesting where to start an analysis. In addition, performance metrics can be used to automatically collapse and expand different hierarchical submodels in the initial view. Moreover, based on user-determined thresholds, one can automatically hide/summarize and emphasize/highlight different parts of the model, further reducing the visual complexity. Such thresholds can be relative, absolute, or based on some form of desired service-level value [8] or Apdex-style score [25]. Additionally, long waiting times can be emphasized by modeling such “hidden delays” as special performance-based control-flow elements (e.g., transitions annotated with a time/clock-symbol).

*“The trouble with having an open mind, of course, is that people will insist on coming along and trying to put things in it.”*

— Terry Pratchett, *Diggers*

## 9 | Translations and Traceability

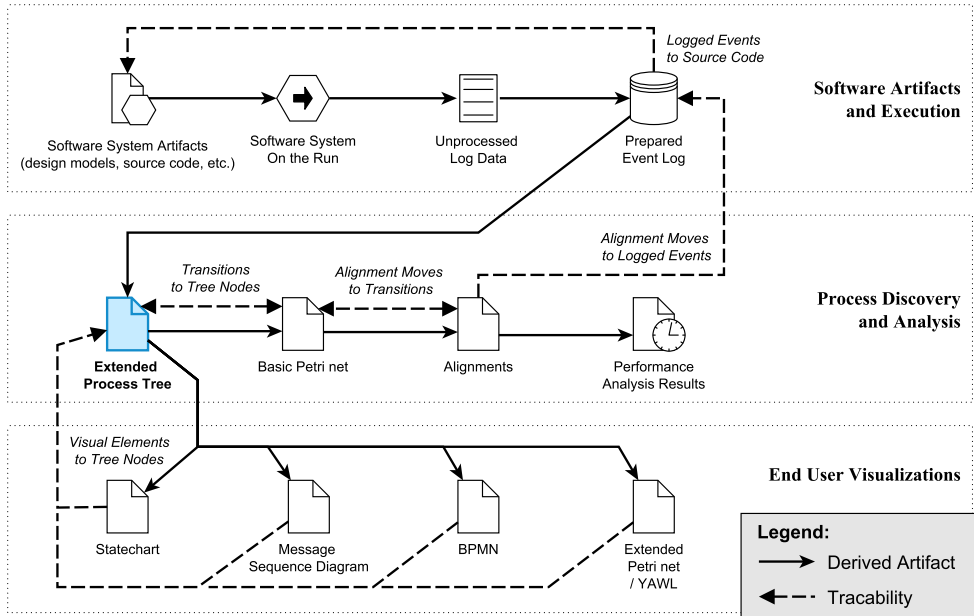
In this chapter, we provide an extensive model translation framework, taking into account the hierarchical, recursive, and cancelation semantics (Contribution 5). In addition, we will show how these translations maintain traceability across models and event logs. Section 9.1 starts with a high-level discussion of our model translation and traceability framework, and the role of extended process trees within this framework. After that, Section 9.2 discusses the translations and tradeoffs for interpreting extended process trees as basic and extended Petri nets. Next, Section 9.3 presents translation schemes from extended process trees to YAWL, BPMN, Statecharts, and Message Sequence Diagrams. Finally, Section 9.4 will conclude this chapter.

### 9.1 The Translations and Traceability Framework

Recall the software process analysis lifecycle from Chapter 1 as depicted in Figure 1.1 on page 4. On one hand we wish to discover various types of models and perform an range of different analyses, and on the other hand we wish to relate these results back to the software system artifacts, thereby closing the loop. The *translation and traceability framework* presented here is a powerful and flexible way to close this analysis loop. This framework uses *translations* (Section 9.1.1) to go from event logs to various models and analysis results, and uses *traceability* (Section 9.1.2) to go from the models back to event logs and the software system artifacts such as the underlying source code. Figure 9.1 summarizes our approach and shows the artifacts and their relations in a software process analysis context.

#### 9.1.1 Translations

As stated in Challenge 5, process mining techniques require a representational bias to make assumptions about the modeled behavior. However, the modeled behavior can be visualized in many different formalisms. In traditional business process mining, visual notations like BPMN, EPC and Petri nets are common. In software engineering, visual notations like UML sequence diagrams, finite state machines, and call graphs are used instead. In practice, a combination



**Figure 9.1:** The *Translations and Traceability Framework*. Shown are the artifacts, their derivation relation, and their traceability in a software process analysis context. The traceability from *visual elements* presented to the end user via the *extended process tree* and *event log* back to the original *software system artifacts* completes the circle in the software process analysis lifecycle, see also Figure 1.1 on page 4.

of different visual notations should be used for different process analysis questions. Ideally, process mining techniques should not be limited by the bias and constraints arising from these visual notations. Rather, they should use an internal representation whose bias is the “greatest common denominator” of the supported visual representations.

In Chapters 6 and 7 we extended the notion of process trees with hierarchical, recursive, and cancelation constructs and semantics. In this way, we extended process trees to be the greatest common denominator of the type of behavior we want to support. By using these *extended process trees as a common representation*, not for visualization purposes but to model the behavior itself, we effectively separate visualization and representation bias. The translations detailed in Sections 9.2 and 9.3 are all defined in terms of extended process trees. This translation approach allows us to integrate with existing techniques like, for example, alignments and allow us to map our results to more user-friendly visualizations. Moreover, we can easily add new visual notations by defining a mapping from extended process trees, and get all the

data and analysis capabilities associated with the underlying tree for free. In addition, this translation approach allows us to define a mapping between any pair of the supported (visual) notations, using the extended process trees as an intermediate format, capturing the intended language or behavior.

### 9.1.2 Traceability

To close the loop in the software process analysis lifecycle, we need a link from *visual elements* presented to the end user back to the original *software system artifacts*. The notion of *traceability* provides these links. We again refer to Figure 9.1 for a summarization of the artifacts and their traceability in a software process analysis context.

We start with the *software artifacts and software execution*. As discussed in Chapter 5, there are various ways to observe and log software behavior and interpret this logged behavior as an event log. At the end, we obtain an event log detailing the observed software behavior. The recorded events provide *traceability back to the source code*. By simply recording which locations in the software artifacts were responsible for generating an event (e.g., a method or a specific line in a source code file, see also Section 5.3.3 on page 105), the first link of traceability can be created.

Given a software event log, we can apply our *process discovery and analysis* techniques. With discovery, we derive an extended process tree, which we will use as a common representation (see also the discussion above). This extended process tree plays a key role in the traceability relations. All of the *end user visualizations* derived from this extended process tree are annotated with traceability, mapping visual elements to specific nodes in the process tree. This enables us to use any of the information available from the extended process tree in all the end user visualizations.

Via the relations build by the alignments technique (see Chapter 8), we obtain a traceability from the extended process tree back to events in the event log. Hence, for any process tree node, we can derive a set of corresponding events, and thus also have traceability back to the source code. In Chapter 10, we show how these traceability relations can be used to allow the user to open the line in the source code corresponding to a selected visual element.

## 9.2 Basic and Extended Petri net Interpretations

Many existing process mining techniques rely on the basic Petri net notation (see Section 2.2.1 on page 25) and do not accept an extended process tree as input. Unfortunately, some of the concepts introduced with extended process trees cannot be fully mapped to these basic Petri nets without loss of semantics.

In practice, a smart basic Petri net interpretation of our extended process trees still allows us to reuse and integrate with existing process mining tech-

**Table 9.1:** From the basic process tree operators to a basic Petri net. Shown are both the atomic Petri net interpretation and the unfolded Petri net interpretation with explicit start and complete transitions.

Process Tree	Basic Petri net	
$\begin{array}{c} \rightarrow \\ \swarrow \quad \searrow \\ a \quad b \end{array}$	<p>Atomic:</p> <p>Unfold:</p>	
$\begin{array}{c} \times \\ \swarrow \quad \searrow \\ a \quad b \end{array}$	<p>Atomic:</p> <p>Unfold:</p>	
$\begin{array}{c} \wedge \\ \swarrow \quad \searrow \\ a \quad b \end{array}$	<p>Atomic:</p> <p>Unfold:</p>	
$\begin{array}{c} \circlearrowleft \\ \swarrow \quad \searrow \\ a \quad b \end{array}$	<p>Atomic:</p> <p>Unfold:</p>	

niques. One example is the integration with the alignments algorithm [21] as was discussed in Section 8.2. Below, we will show how we can construct both an extended Petri net and a smart basic Petri net interpretation which can be used in, for example, calculating alignments.

### 9.2.1 Basic Process Trees to Basic Petri nets

The four basic process tree operators, sequence ( $\rightarrow$ ), exclusive choice ( $\times$ ), concurrency ( $\wedge$ ), and structured loop ( $\odot$ ), are easily translated to a basic Petri net. In Table 9.1 we show two translations of these basic operators. The first translation assumes atomicity, i.e., an activity is completed as soon as it is fired. The second translation unfolds each transition into a start and complete transition. There are two reasons to consider the unfolded version: 1) unfolding allows us to accurately add the notions of named submodels and recursive references later on, and 2) unfolding is explicitly used in the performance analysis techniques in Chapter 8, see also Section 8.2.1 on page 213.

### 9.2.2 Mapping Named Submodels and Recursive References

The basic Petri net notation has no concept of named submodels or recursive references. However, for the purpose of, for example, computing alignments, it is possible to encode these notions using an unfolded basic Petri net.

The first row in Table 9.2 shows a possible translation of the named sub-

**Table 9.2:** From the hierarchical process tree operators to a basic Petri net.

Process Tree	Basic Petri net
$\begin{array}{c} \nabla_x \\   \\ a \end{array}$	
$\begin{array}{c} \nabla_x \\   \\ \times \\ / \quad \backslash \\ \Delta_x \quad a \end{array}$	<p>Inhibitor:</p> <p>Plain:</p>





model operator ( $\nabla_x$ ) introduced in Section 6.3. Observe how the places  $pxs$  and  $pxc$  mark the start and end of the named submodel. The actual submodel boundary transitions  $x+s$  and  $x+c$  use these places to start the named submodel and wait for completion. Hence, the named submodel  $x$  can only be completed (transition  $x+c$ ) after the body (activity  $a$ ) has been executed.

The translation of the recursive reference operator ( $\Delta_x$ ) is more involved. The second row in Table 9.2 shows two possible translations, one with the use of an inhibitor arc<sup>1</sup>, and one with only basic Petri net arcs. To model a recursive reference  $\Delta_x$ , we are going to reuse the places  $pxs$  and  $pxc$  we introduced for the referenced named submodel operator  $\nabla_x$ . The idea is to have a second set of transitions  $x+s$  and  $x+c$ , called the recursion transitions, inside the named submodel. These recursion transitions do two things: 1) they use the places  $pxs$  and  $pxc$  to mark the start and end of a nested named submodel execution, and 2) they use an internal place  $pxr$  to count the number of active recursions. We attach an inhibitor arc from the original named submodel transition  $x+c$  to place  $pxr$  to check whether all recursive executions have finished.

When performing alignments on a basic Petri net without inhibitor arcs, we can use the plain version shown in the second row in Table 9.2. In this case, we still need to model the constraint that all recursive executions are finished before we perform the original named submodel transition  $x+c$ . In practice, we can often rely on the alignment cost function (see Definition 8.2.3) to ensure place  $pxr$  is emptied. After all, the alignment algorithm tries to reach the final marking, and thus needs proper completion without tokens left behind. Hence, there is a high likelihood that the recursion transitions are used in the intended order. However, deviations might cause problems for such alignments.

### 9.2.3 Mapping Sequence Cancellations and Loop Cancellations

Table 9.3 shows two possible translations for both cancellation operators ( $\overset{\star}{\rightarrow}$ ,  $\overset{\star}{\circlearrowleft}$ ) introduced in Section 7.2. One translation uses the cancellation region or reset semantics<sup>2</sup>, the other translation only uses basic Petri net arcs. We model the body of the cancellation operator, the left/first subtree, as normal. Here, we can reuse the atomic or unfolded approaches introduced before. Next, with the resulting model we 1) wrap this body in a cancellation region (places  $p1$  to  $p4$ ) and 2) add an exit  $\tau$ -transition  $te$ , marking the non-cancellation end of this submodel. For each of the places following a cancellation trigger operator ( $\overset{\star}{\rightarrow}_a^C$ ), we add a  $\tau$ -transition  $tc$  firing the cancellation region. We model each of the cancellation paths (non-first subtrees) reusing the atomic or unfolded approach again. The starting place or source for a cancellation path is the output place of the tau-transition  $tc$  firing the cancellation region, i.e., place  $pc$

<sup>1</sup> Recall, inhibitor arcs were explained in Section 2.2.1 on page 31.

<sup>2</sup> Recall, cancellation regions and reset arcs were explained in Section 2.2.1 on page 31.

**Table 9.3:** From the cancellation process tree operators to a basic Petri net. Below, we used the atomic translation for brevity.

Process Tree	Basic Petri net
	<p>Reset:</p> <p>Plain:</p>
	<p>Reset:</p> <p>Plain:</p>

in our examples. For the sequence cancellation operator ( $\rightarrow^*$ ), the final place or sink for a cancellation path is the output place of the exit tau-transition  $te$ , i.e., place  $pe$  in our examples. For the loop cancellation operator ( $\overset{*}{\circ}$ ), the final place or sink for a cancellation path is the starting place of the cancellation body, i.e., place  $p1$  in our examples.

When performing alignments on a basic Petri net without cancellation regions or reset arcs, we can use the plain versions shown in Table 9.3. In this case, we still need to model the constraint that a cancellation path is executed once all tokens inside the cancellation region are consumed, i.e., places  $p1$  to  $p4$  are empty. In practice, we can often rely on the alignment cost function (see Definition 8.2.3) and the grey tau-transitions shown in Table 9.3. As the alignment algorithm tries to reach the final marking without tokens left behind, if possible, the grey  $\tau$ -transitions can and will clean up the tokens inside the cancellation. However, deviations might again cause problems.



### 9.3 Model to Model Translations

As stated before, we use the extended process trees as a common representation. In this section, we present translation schemes from extended process trees to YAWL, BPMN, Statecharts, and Message Sequence Diagrams.

#### 9.3.1 Extended Process Trees to YAWL

Table 9.4 shows the translation from the extended process tree operators to a YAWL model. For the hierarchical operators, we define the named submodel  $\nabla_x$  as a separate model. Using the notion of *composite tasks*, we can reference this model definition from outside (named submodel  $\nabla_x$ ) and within the model (recursive reference  $\Delta_x$ ). For the cancellation operators, we use the native support for *cancellation regions*, yielding a solution comparable to the reset Petri nets in Table 9.3.

#### 9.3.2 Extended Process Trees to BPMN

Table 9.5 shows the translation from the extended process tree operators to a BPMN model. For the hierarchical operators, we define the named submodel  $\nabla_x$  as a separate model. Using the notion of *subprocess tasks*, we can reference this model definition from outside (named submodel  $\nabla_x$ ) and within the model (recursive reference  $\Delta_x$ ).

Translating the cancellation operators is a bit more involved since BPMN has no direct support for cancellation regions. Instead, we rely on boundary events of a subprocess. We wrap the contents of a cancellation region in a *subprocess* and define a boundary *catch cancel event* to mark the start of a cancellation path. Inside the subprocess, we define a *throw cancel event* after each cancellation trigger operator ( $\star_a^C$ ). These throw events are linked to the catch events of the corresponding cancellation region. For the sequence cancellation operator ( $\overset{\star}{\rightarrow}$ ), we connect the end of a cancellation path with a XOR-join gateway just after the cancellation subprocess. For the loop cancellation operator ( $\overset{\star}{\circlearrowleft}$ ), we connect the end of a cancellation path with a XOR-join gateway at the start of the cancellation subprocess.

#### 9.3.3 Extended Process Trees to Statecharts

Table 9.6 shows the translation from the extended process tree operators to a Statechart model. Note the use of *AND super-states* to model parallelism ( $\wedge$ ).

For the hierarchical operators, we use a named *XOR super-state* to model a named submodel  $\nabla_x$ . There are several options to connect states in the presence of a named submodel. The examples in Table 9.6 connect non-super state directly. As an alternative, one can indicate starting states for each super-state, and connect transitions from outside to the named super-state itself. A third possibility is to use pseudo-states representing entering and exiting

named super-states. Each of the above possibilities is equally valid. However, when overlaying a statechart model with detailed performance information, the additional transitions in the second and third option can be useful for visualizing waiting times. For modeling the recursive reference  $\Delta_x$ , we decorate the nested state  $x$ , indicating this state is defined in terms of the corresponding *super-state*. Formally, one would rely on the object-oriented extension defined in [154] to instantiate a nested version of the *XOR super-state object* named  $x$ .

For the cancelation operators ( $\overset{\star}{\rightarrow}$ ,  $\overset{\star}{\odot}$ ) we use a nameless *XOR super-state* to denote the cancelation region. For each cancelation trigger operator ( $\star_a^C$ ), we decorate the nested state  $b$ , indicating this state captures a cancel trigger. For modeling the actual cancelation behavior we have two options: 1) we model a transition from each cancel trigger state to the cancelation paths, or 2) we model only a single transition from the cancelation region XOR super-state to the cancelation paths. The first approach is more accurately modeling the intended semantics, while the second approach typically results in less transitions and thus a visually simpler model. We model the cancelation paths again as normal. For the sequence cancelation operator ( $\overset{\star}{\rightarrow}$ ), we connect the end of a cancelation path with the first state after the cancelation region. For the loop cancelation operator ( $\overset{\star}{\odot}$ ), we connect the end of a cancelation path with the first state inside the cancelation region.

### 9.3.4 Extended Process Trees to Message Sequence Diagrams

Table 9.7 shows the translation from the extended process tree operators to a Message Sequence Diagram (MSD). Fragments are used to model the various control-flow concepts. Observe how the loop semantics are only approximated. To the best of our knowledge, it is not possible to capture the loop-retry semantics of the loop operator ( $\odot$ ) with the standard MSD fragments.

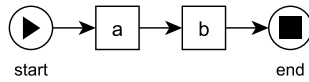
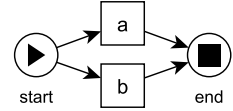
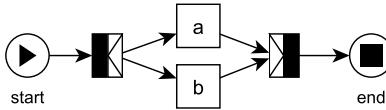
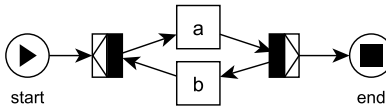
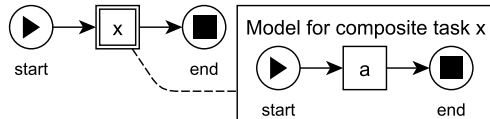
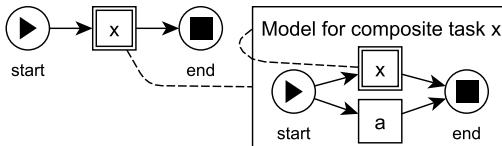
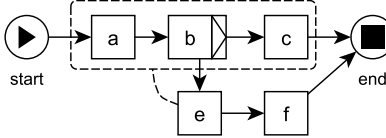
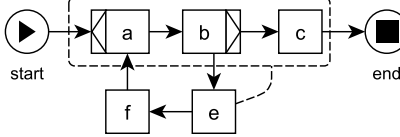
The named subtree operator ( $\nabla_x$ ) can be represented using a nested activity box and a combination of an *activation* and *reply* message. The recursive reference operator ( $\Delta_x$ ) cannot be represented in MSD. As an approximation, we can use another nested activity box plus a recursive call annotation.

The cancelation operators ( $\overset{\star}{\rightarrow}$ ,  $\overset{\star}{\odot}$ ) also do not have a native MSD representation. However, with the introduction of the *try* and *catch* fragment, we can accurately model a sequence cancelation ( $\overset{\star}{\rightarrow}$ ). The loop cancelation ( $\overset{\star}{\odot}$ ) can only be approximated, the cancelation loopback path suffers from the same problems as the loop-retry discussed above. Alternatively, we can introduce a new *retry-catch* fragment.

The real strength of the Message Sequence Diagram is its use of lifelines. In the examples in Table 9.7, we only used one lifeline labeled  $C$ . However, one can use a secondary event log classifier to determine on which lifeline an event “lives”. A typical example is to use method names as the primary


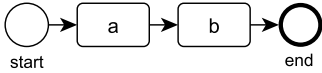

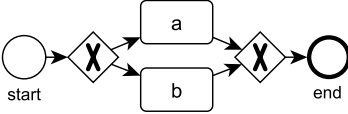

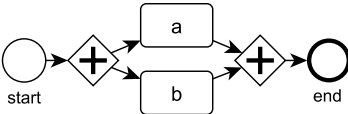

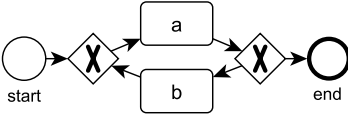

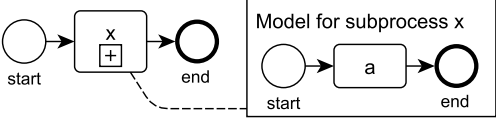
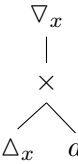
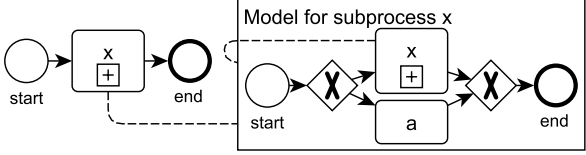
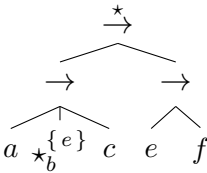
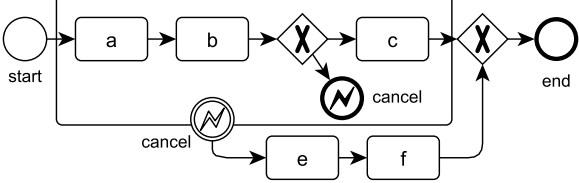
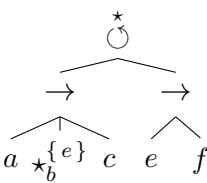
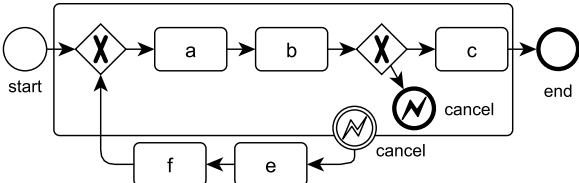
activity classifier, and class names, library packages, or component names as the secondary lifeline classifier. Figure 2.11 on page 37 and Figure 8.17 on page 251 are a good example of this. Another useful secondary classifier is a resource (group) identifier or a data attribute encoding a domain property such as a location or object involved or claimed during the indicated events.

**Table 9.4:** From the extended process tree to a YAWL model.

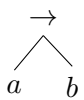
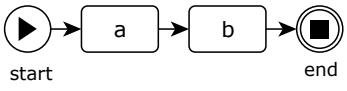
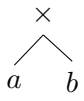
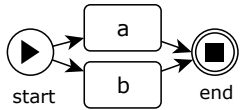
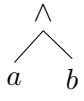
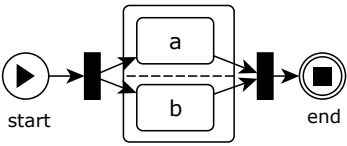
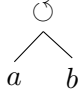
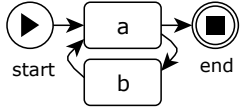
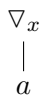
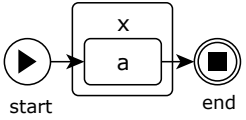
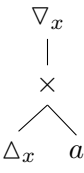
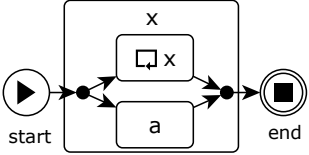
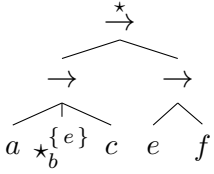
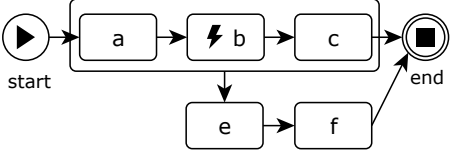
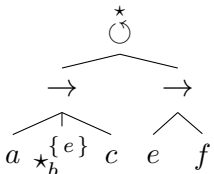
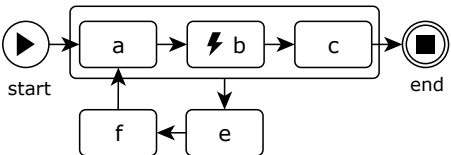
Process Tree	YAWL
$\begin{array}{c} \rightarrow \\ \swarrow \quad \searrow \\ a \quad b \end{array}$	
$\begin{array}{c} \times \\ \swarrow \quad \searrow \\ a \quad b \end{array}$	
$\begin{array}{c} \wedge \\ \swarrow \quad \searrow \\ a \quad b \end{array}$	
$\begin{array}{c} \circlearrowleft \\ \swarrow \quad \searrow \\ a \quad b \end{array}$	
$\begin{array}{c} \nabla x \\   \\ a \end{array}$	
$\begin{array}{c} \nabla x \\   \\ \times \\ \swarrow \quad \searrow \\ \Delta x \quad a \end{array}$	
$\begin{array}{c} \overset{*}{\rightarrow} \\ \swarrow \quad \searrow \\ \rightarrow \quad \rightarrow \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ a \quad \star_b \{e\} \quad c \quad e \quad f \end{array}$	
$\begin{array}{c} \overset{\circlearrowleft}{\rightarrow} \\ \swarrow \quad \searrow \\ \rightarrow \quad \rightarrow \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ a \quad \star_b \{e\} \quad c \quad e \quad f \end{array}$	



**Table 9.5:** From the extended process tree to a BPMN model.

Process Tree	BPMN
	
	
	
	
	
	
	
	

**Table 9.6:** From the extended process tree to a Statechart.

Process Tree	Statechart
	
	
	
	
	
	
	
	





**Table 9.7:** From the extended process tree to a Message Sequence Diagram.

Process Tree	MSD	Process Tree	MSD

## 9.4 Conclusion and Open Challenges

In this chapter, we provided an extensive model translation framework, taking into account the hierarchical, recursive, and cancelation semantics (Contribution 5). In addition, we showed how these translations maintain traceability across models and event logs. In Chapter 10, we will show how this traceability can be leveraged to close the loop in the software process analysis lifecycle from Chapter 1 (see Figure 1.1 on page 4). In Chapter 12, we will show how the various visual notations covered by Section 9.3 can be used and combined for comprehensive software analyses.

With the model translation and traceability framework presented in this chapter, there are several interesting research directions for future work.

■ **Future Work 9.1 — Extend Common Representation.** As we argued in Section 9.1.1, there is a need for a common representation whose bias is the “greatest common denominator” of the supported visual representations. With the extended process trees we made a first attempt at such a *common representation*. However, there are various types of behavior we have not considered yet. Some examples include: workflow patterns like the *milestone* [15], multi-instance patterns, various environment triggers like the BPMN *events* or MSD *external messages*, and the statechart *history* and *communication* concepts. Additionally, one should consider extending such a common representation with a *data perspective* (variables, guards, etc.) and *data flow* relations.

■ **Future Work 9.2 — Extend Transformations.** The current framework includes extended process trees, Petri nets, YAWL, BPMN, Statecharts, and Message Sequence Diagrams. However, there are various other useful notations, both for visualization and for tool integration, which one might consider. Obvious examples include automata, labeled transition systems, and UML flow diagrams. However, one might also consider derived models focussing on specific perspectives, such as resource allocation schemas, call graphs, class diagrams, and network graphs. The trick is to maintain traceability across all models, linking and combining results between the various notations. For example, control-flow models and performance analysis results should allow one to trace bottlenecks from these models to derived class diagrams and network graphs.

■ **Future Work 9.3 — Integrate Existing Static Artifacts as a First-Class Citizen.** The recorded events provide *traceability back to the source code*. However, these software artifacts are currently not fully integrated in this framework. For example, it can be desirable to add relations between event from an event log and existing architecture artifacts. This would enable, for example, a traceability link between the performance metrics calculated over a discovered model and a class diagram, showing which classes and packages contribute the most to the overall running time.



# IV | Applications

<b>10</b>	<b>Tool Implementations</b> .....	<b>277</b>
10.1	Introduction	
10.2	The Statechart Workbench	
10.3	The Instrumentation Agent	
10.4	The SAW Eclipse Plugin	
10.5	User Experience Evaluation	
10.6	Conclusion	
<b>11</b>	<b>The Software Process Analysis Methodology</b>	<b>303</b>
11.1	Introduction and Positioning	
11.2	The Software Process Analysis Methodology	
11.3	Practical Concerns	
11.4	Conclusion	
<b>12</b>	<b>Case Studies</b> .....	<b>323</b>
12.1	Introduction	
12.2	Open Source Software Case Study – The JUnit 4.12 Library	
12.3	Industrial Software Case Study – The Wafer Handling Process	
12.4	Conclusion	

---

## I Introduction

Chapter 1  
Overview

Chapter 2  
Preliminaries

Chapter 3  
Related Work

Chapter 4  
A Process Mining  
Foundation

---

## II Hierarchical Process Discovery

Chapter 5  
On Software Data  
and Behavior

Chapter 6  
Hierarchical and Recursion  
Aware Discovery

Chapter 7  
Cancellation Discovery

---

## III Beyond Model Discovery

Chapter 8  
Hierarchical Performance  
Analysis

Chapter 9  
Translations  
and Tracability

---

## IV Applications

Chapter 10  
Tool Implementations

Chapter 11  
The Software Process  
Analysis Methodology

Chapter 12  
Case Studies

---

## V Closure

Chapter 13  
Conclusion

Appendix A  
Proofs

In Part IV, we discuss applications of the techniques and algorithms presented, and discuss the way they are supported by our tools.

**Chapter 10** presents the implemented tools, their interactions, and how our tools integrate with existing software artifacts.

**Chapter 11** presents a methodology for obtaining and analyzing software event log data in a structured way.

**Chapter 12** presents various case studies, showing how our techniques can be used in practice.

“Taking time to be totally, gloriously,  
proudly unproductive will ultimately  
make you better at your job.”

— Michael Guttridge

## 10 | Tool Implementations

In the previous chapters, we presented various techniques, heuristics, and algorithms. In this chapter, we present three tools that implement and consolidate the work discussed (Contribution 6). All of the tools are open source and publicly available. In addition, we provided a tool screencast showing the tools in action, scan the QR code or use the link in Figure 10.1 to the right. Section 10.1 gives a global introduction to the proposed tools, and puts the tools into context. Afterwards, Sections 10.2 to 10.4 discuss each of the tools. Section 10.5 presents a small user experience evaluation for the presented tools. Finally, Section 10.6 concludes this chapter.



<https://youtu.be/xR4XfU3E5mk>

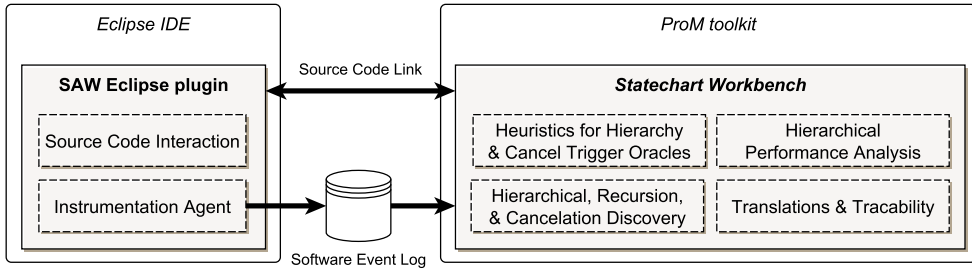
**Figure 10.1:** Scan the above QR code or use the shown url to see the Statechart Workbench and SAW tools in action.

### 10.1 Introduction

In this chapter, we present three tools that implement and consolidate the discussed work: the *Statechart Workbench*, the *Instrumentation Agent*, and the *SAW Eclipse plugin*. Figure 10.2 schematically shows the implemented tools, their context, and the interaction between the tools.

The core of our work is the *Statechart Workbench* tool, a novel software process discovery, analysis, and exploration tool. The Statechart Workbench implements the techniques from Chapters 6 to 9, and provides a rich, mature, and interactive integration of these techniques. With the Statechart Workbench, one can use both business and software event logs to analyse hierarchical behavior, performance (timings), frequency (usage), conformance and reliability in the context of various formal models.

Next to the Workbench, we implemented the *Instrumentation Agent* tool, a Java agent capable of producing a ready-to-use event log when loaded with a Java program. This Instrumentation Agent consolidates some of the tech-



**Figure 10.2:** The implemented tools, their context, and the interaction between tools. The *Statechart Workbench* implements the techniques from Chapters 6 to 9. The *Instrumentation Agent* produces ready-to-use event logs when loaded with a Java JAR. The *SAW Eclipse plugin* provides a user interface for both the *Instrumentation Agent* and the source code traceability link.

niques and structures discussed in Chapter 5. The provided *Instrumentation Agent* is a reference implementation providing logging for the Java programming language. However, note that the ideas from this *Instrumentation Agent* can easily be transferred to other programming languages, as discussed in Section 5.1.1 on page 93.

Finally, we provide the *SAW Eclipse plugin*. This Eclipse plugin provides a user interface for the *Instrumentation Agent*, allowing users to get started right away with their own software by providing a user interface for software logging in the Eclipse IDE. In addition, the *SAW Eclipse plugin* enables the user to interactively link the results from the *Statechart Workbench* back to the source code locations inside the software system (Chapter 9).

**Tool Availability** – The *Statechart Workbench* is available via ProM [159]. In addition, the source code, readme file [114], user manual [116], and screencast [115] for the *Statechart Workbench* are also publicly available. The *Instrumentation Agent* and *SAW Eclipse plugin* are available via [113].

## 10.2 The Statechart Workbench

As indicated in the introduction, the *Statechart Workbench* tool represents the core of our work. In this section, we will first give an overview of the tool and some of the key design decisions (Section 10.2.1) and overall architecture (Section 10.2.2). After that, we present a walkthrough of the tool, showing the various capabilities and user interface choices (Section 10.2.3).

### 10.2.1 Overview and Design Decisions

With the *Statechart Workbench*, we implemented and combined the techniques from Chapters 6 to 9. In order to lower the threshold for users to use the pro-

posed techniques, we designed the Statechart Workbench around the following guiding principle:

*The user should be able to easily, quickly and seamlessly explore the behavior in an event log using an array of chained algorithms. Moreover, the user should be able to interact with the produced models and results and inspect the effects of different parameter settings in a near real-time fashion.*

Many of the more mature process mining tools follow this guiding principle. Examples include but are not limited to: the Inductive Visual Miner [132], the Interactive Data-aware Heuristic Miner [138], Disco [76], and Celonis [51].

Following this guiding principle, the Statechart Workbench is designed as a process exploration workflow tool. We put the discovered model central in the tool and we allow users to interactively change various parameters in near real-time, thus encouraging the exploration of the parameter space and observed behavior. We use an automated chain of algorithms, i.e., a workflow, to limit the number of user interactions needed. Furthermore, to quickly provide the user with initial results, we provided configuration presets for common use cases and tuned certain parameter defaults to reduce initial computation times.

The Statechart Workbench is available as a plugin for the Process Mining Toolkit ProM [187], leveraging existing work inside ProM and allowing users to use our work together with the various other process mining techniques. In the architecture description below, we will highlight key examples of the above design decisions.

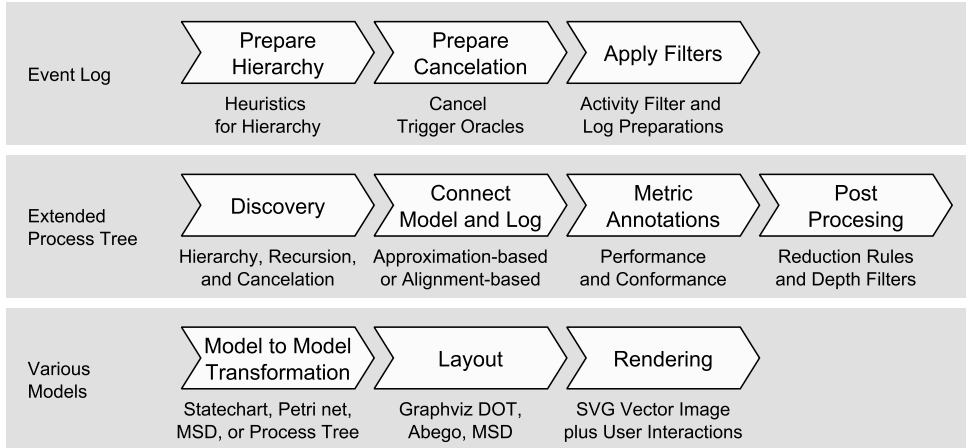
### 10.2.2 Software Architecture

At the core of the Statechart Workbench is the *Workbench Workflow*, as depicted in Figure 10.3. This workbench workflow implements the automated chain of algorithms and tasks. Each of the depicted tasks (the chevrons in the figure) has some parameters. To encourage exploration, a user can change any parameter at any time. Whenever a user changes one of the inputs, automatic background computations are triggered such that as few tasks as possible are recomputed.

Each step in the Workbench Workflow has a well-defined functional interface based on its input-output models. This setup allows us to easily switch individual algorithms and extend the workbench with new features. Note how the tasks compare to the artifacts in the *Translations and Traceability Framework* in Figure 9.1 on page 260. Below, we will discuss each step in turn.

**Prepare Hierarchy** – In this step, we allow the user to select a *heuristic for hierarchy* for the hierarchical and recursion aware discovery. Based on the discussion in Section 6.2.4 on page 116, we implemented the heuristics and options listed below.





**Figure 10.3:** The Statechart Workbench workflow. The chevrons indicate algorithms and tasks chained into the workflow. Beneath the chevrons, descriptions regarding the user input or used algorithms are displayed. The tasks are grouped based on the main type of model involved. Whenever an input changes, only the minimal number of tasks are recomputed.

- The *Nested Calls* heuristic is a preconfigured variant of the nested intervals, which by default looks for nested method calls based on lifecycle information, a common pattern in software logs.
- The *Structured Names* heuristic uses a split symbol approach over activity names. By default, this heuristic is configured on the dot (.) symbol to split package-class names like `org.processmining.Main.main()` into a hierarchy over the static package structure or “architecture”.
- The *Pattern Names* heuristic matches an exact pattern, using regular expressions, over activity names to form a hierarchy.
- The *Multi Attributes* heuristic uses a sequence of classifiers or attribute names to infer a hierarchical event label sequence. This heuristic is very effective where events have been annotated with external information such as source code information, domain knowledge, or patterns inferred by domain experts or other (ProM) tools.
- The *Single Classifier* option essentially allows one to bypass or disable the hierarchical discovery, by inferring a single level of hierarchy.
- The *Existing List Classifier* option assumes an existing atomic, list-based structure in the event log, encoding hierarchical information much like Definition 6.2.2 on page 114.

In addition, we provided two presets to allow users to quickly bypass the heuristics configurations and just get an initial model discovered.

- The *Normal Log* preset assumes regular, flat (business) event logs. In this preset, we use a default single classifier setup without any cancelation trigger oracle (see next step). This preset allows users to just get started with most (non-software) event logs.
- The *Software Log* preset assumes a typical software event log focussing on internal software behavior, such as the logs produced by our Instrumentation Agent. In this preset, we use a default nested calls setup with the default software cancelation trigger oracle (see next step). This preset allows users to just get started with typical software event logs.

**Prepare Cancelation** – In this step, we allow the user to select a *trigger oracle* for the cancelation discovery. Based on the heuristics discussion in Section 7.3 on page 173, we implemented the options listed below.

- The *Handle Exception* heuristic is designed to work together with the *Nested Calls* hierarchy heuristic, utilizing software domain knowledge. We assume that the start of an exception catch region is logged via a so-called handle event and detectable via a special lifecycle transition. The *Nested Calls* hierarchy heuristic detects and prepares such events, and the *Handle Exception* heuristic extracts these events to populate the cancelation trigger oracle. This heuristic covers typical software event log cases, such as the ones produced by our Instrumentation Agent.
- The *Manual* option allows users to select specific activities for the trigger oracle, using, for example, naming semantics or their domain knowledge.

**Apply Filters** – After the heuristics are applied, we apply a simple activity frequency filter and prepare the event log data-structures for the adapted Inductive Miner discovery framework.

**Discovery** – In the discovery step, we can either execute the Naive Hierarchical Discovery (Section 6.4) or Recursion Aware Discovery algorithm (Section 6.5), and we can optionally mix in the Cancelation Discovery (Section 7.4) and existing Inductive Miner extensions like infrequent discovery (Section 4.4). Note that we used the two hierarchical discovery algorithms as a basis, and mix in the cancelation as an option. This is because the hierarchical discovery is, implementation wise, much more involved, especially when considering the delayed discovery from Algorithm 6.2. For the hierarchy-cancelation mix approach, we used the approach suggested in Table 7.5. The result is an extended process tree with both hierarchical and cancelation features.

**Connect Model and Log** – After discovery, we establish a relation between the model and log. For this step, we provided two implementations: *approximation-based* and *alignment-based* connections. The alignment-based approach uses the normal alignments algorithm using a Petri net view of the discovered extended process tree, as discussed in Section 8.2. After an alignment has been computed, the results are projected back onto the process tree by precomputing several data-structures, see also the discussion in Section 8.5.2.

Since these alignments computations can be quite expensive, we also provide an alternative. The approximation-based approach is a first attempt at the concept discussed in Future Work 8.4 (see page 257). We use the suggested event-model mappings provided by the log splitting inside the Inductive Miner framework. The advantage is that these computations are very fast, hence it is the default setting when first discovering a model. The disadvantage is that we have no guarantee that the resulting mapping and derived metrics are accurate and reliable.

**Metric Annotations** – Based on the user selected performance/conformance metrics, several data-structures are preconfigured for the right metrics to be computed. We use a delayed computation implementation for the actual metric computation itself. This way, metrics are only computed when used in the visualization. The main reason for this design choice is the post-processing in the next step: if parts of the model are hidden, then we do not have to compute the metrics for the hidden parts. Note that when a different metric is selected, most of the data-structures can be reused without recomputations.

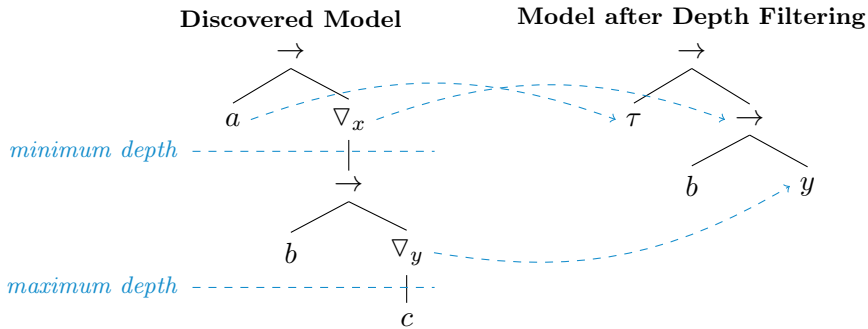
**Post-Processing** – During post-processing, we apply two transformations to the discovered extended process tree. First of all, we *apply the reduction rules* from Tables 2.1 and 7.3. This helps in providing a visually simpler model. Next, we apply user-chosen *hierarchical depth filters*. In Section 10.2.3, we will discuss a coarse-grained and a fine-grained way for setting up the hierarchical depth filters. For now, the idea of depth filtering is illustrated in Figure 10.4. A minimum depth filter trims parts near the root of the process tree. This can be useful when interesting (software) behavior is obfuscated by initial abstraction layers such as `main()` and `run()` functions. A maximum depth filter hides details of named subprocesses. This can be useful when one initially starts exploring a model by hiding unnecessary details.

**Model to Model Transformations** – After the post-processing, we can start visualizing the discovered and annotated model. The first step is a model-to-model transformation, to show the results in a model of the users choice. This step implements the transformations from Section 9.3. By default, the Statechart visualization is selected, hence the name Statechart Workbench<sup>1</sup>.

**Layout** – For the graph-based models (Statecharts, Petri nets) we use Graphviz DOT [32] for computing a layout, which was already embedded inside the ProM framework. For the Message Sequence Diagrams, we implemented a simple layout algorithm that uses the structure in the process tree to order and arrange MSD messages vertically. For the Process Tree visualization, we used the compact tree layout algorithm provided by Abego TreeLayout [19].

---

<sup>1</sup> The original reason for choosing the Statechart visualization as default was the earlier research ideas resulting from a visit by David Harel. Although no real collaboration resulted from these earlier ideas, we stuck to the Statechart idea as it was a simple, hierarchical visualization with which we could easily perform various experiments.



**Figure 10.4:** Illustration of hierarchical depth filtering. Left is the model before filtering, and right is the model after filtering. In this example, we hide everything above  $x$  and below  $y$ . The dashed arrows relate the changed tree nodes between both models.

**Rendering** – After obtaining layout coordinates, the model is rendered as a SVG vector image. This allows us to provide seamless zooming without loss of quality. Furthermore, during rendering, we use various decorators to determine the labels and colors for the visual elements. Note that these decorators use the prepared metric annotations, kicking off the delayed metric computations. Using the SVG objects, various user interactions are linked to the shown model, such as info popups, selections, etc. For more details, see Section 10.2.3 below.

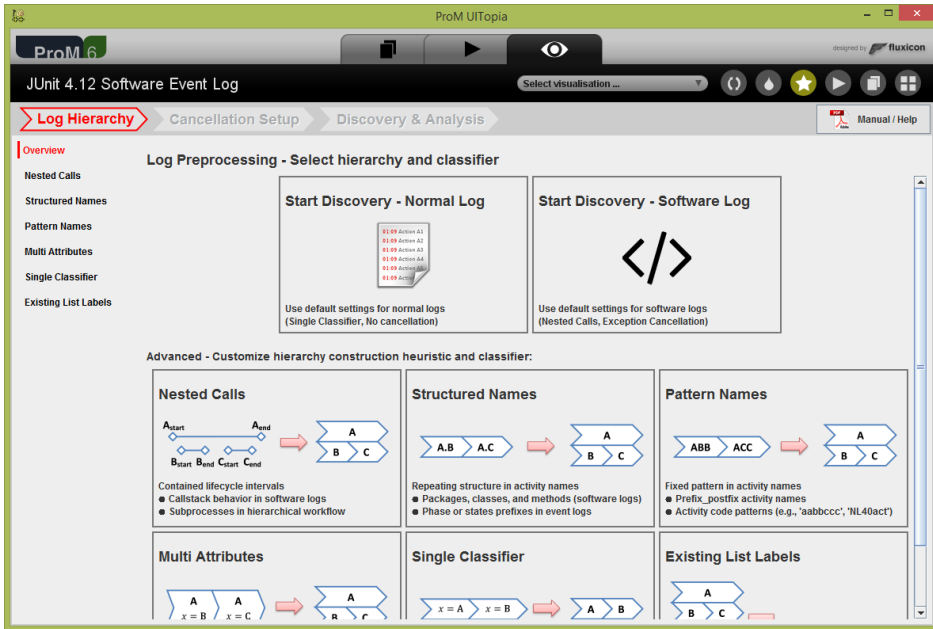
### 10.2.3 Tool Walkthrough

In the previous sections, we discussed the technical side of the Statechart Workbench. In this section, we will discuss the user interface, showing how everything connects and can be used by end users. We will be using the JUnit software event log [108] as an example in the provided screenshots.

#### Starting the Statechart Workbench

To open the Statechart Workbench, load an XES event log, and then either launch the *Statechart Workbench* log visualizer or start the *Discover using the Statechart Workbench* plugin action. See the user manual for more detailed steps on how to do this in ProM [116].

When the user opens the Statechart Workbench, the user first has to inform the tool how to interpret their data. The initial screen, as shown in Figure 10.5, corresponds to the *Prepare Hierarchy* and *Prepare Cancellation* tasks in the Statechart Workbench workflow. In the user interface, the tool provides the user with both *presets to skip all settings and quickly get to the model* as well as a documented and illustrated wizard to set up the details of the heuristics discussed in Section 10.2.2. Shown on top are the two quick presets to get



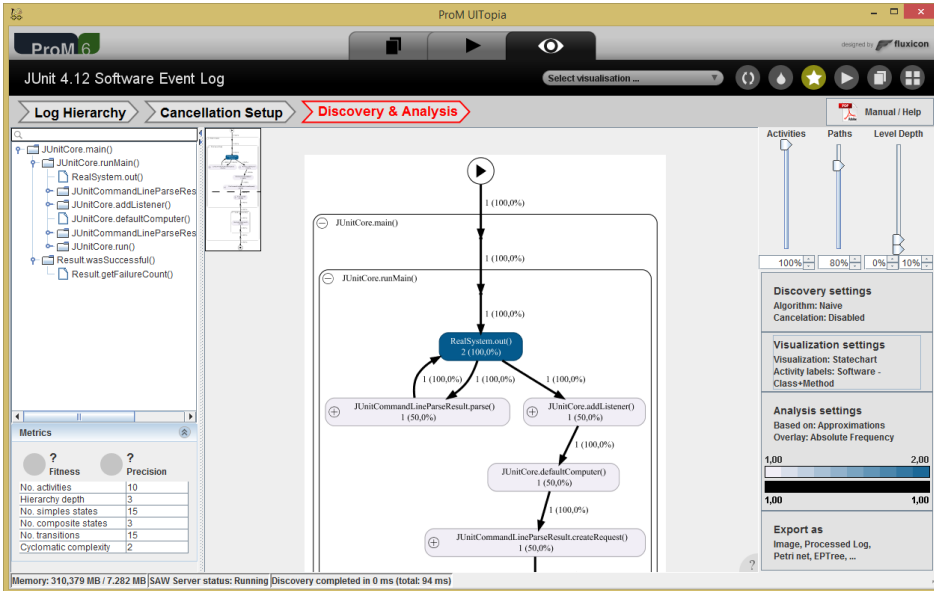
**Figure 10.5:** The initial Statechart Workbench for setting up the heuristics. Shown is the list of available heuristics for hierarchy. Presets allows the user to skip all settings and quickly get to the model.

started with most event logs. If the user chooses to configure the heuristics manually, the user gets the option to select a hierarchy heuristic, and afterwards a cancellation heuristic. Note that the activity classifier used for discovery is set in the hierarchy heuristic, as an activity classifier is already needed for log preprocessing. Upon selecting a preset or setting up the heuristics, a model is discovered and the tool switches to the *Discovery & Analysis* screen.

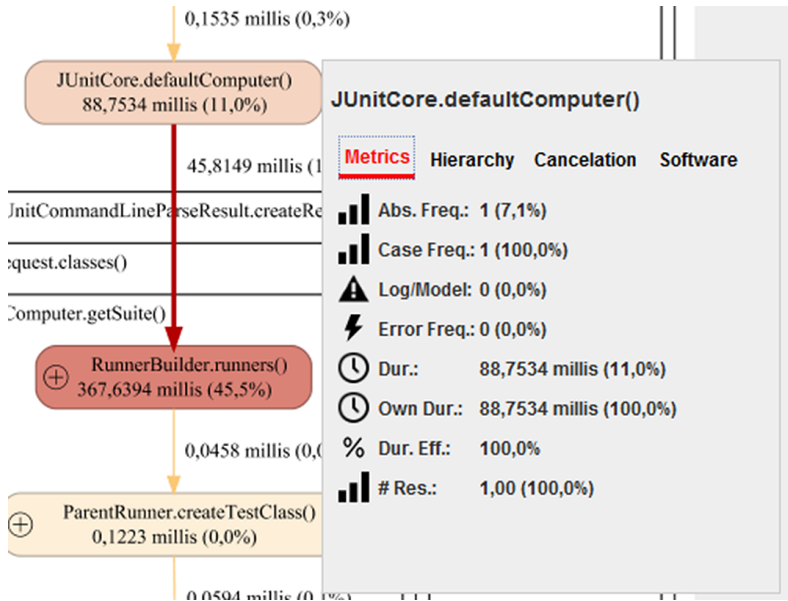
### Discovery and Analysis

After the initial heuristics setup, the user is presented with the *Discovery & Analysis* screen as shown in Figure 10.6. In the center of this screen, the discovered model is shown. On the left, a list of all the activities is shown in a hierarchical tree view. And on the right, various options and settings are shown. Using the three big chevrons at the top, the user can go back to select different hierarchy and cancellation heuristics.

**Center panel: Discovered Model** – The center panel shows the discovered model. By dragging the screen (left-click and hold on the model), the user can move through the model. By using the scroll wheel, the user can zoom in and out. Thanks to the underlying vector image, the user can seamlessly zoom in and out (see also Section 10.2.2). We provide the user with a minimap of the shown model to help navigation in larger models.



**Figure 10.6:** The Discovery & Analysis screen in the Statechart Workbench. To the left, global metrics and a searchable activity list are given. In the middle, an interactive, zoomable model with minimap is given. The model and list views are linked. To the right, the user can change various settings.



**Figure 10.7:** Info popup screen in the Statechart Workbench. By clicking on a model element, the user can access an info popup with several tabs showing detailed information.

In the case of hierarchical models, the tool presents plus and minus icons on named submodels. Via these icons, the user can expand and collapse a subprocess/hierarchy level. This allows the user to intuitively influence the hierarchical depth filters in a fine-grained manner.

By left clicking on a node, the user can open up a detailed popup with more information (Figure 10.7). Here, various details from the selected model element and the underlying events are shown. The popup includes metrics, information about the modeled hierarchy and cancelation, and the traceability information linking back to the source code.

As we will discuss in more detail in Section 10.4, we provided a link between the Statechart Workbench and the Eclipse IDE. By using the detailed popup or by double clicking on a model element, the user can open the corresponding source code line in Eclipse. This action simply uses the available traceability information as discussed in Section 9.1.2.

**Left panel: Activity overview and selection** – The panel on the left shows all the activities in a hierarchical tree view. The search bar at the top allows the user to lookup specific activities: it filters the shown list. By selecting activities (left click), the corresponding parts in the center model will be outlined in red. With control-click, the user can select and deselect multiple activities at the same time.

In the case of hierarchical models, the user can expand and collapse a subprocess/hierarchy level by clicking on the tree icons left of the activity names. These actions again update the hierarchical depth filters accordingly. Furthermore, the activity list on the left and model in the center are linked: expanding or collapsing elements in one view updates the other, and vice versa.

Below the activity overview, various model metrics are given. When the model and log are aligned, the *fitness* and *precision* of the discovered model are shown. In addition, some simple model statistics are given, such as the number of activities, the discovered hierarchy depth, and the cyclomatic complexity of the visualized graph.

**Right panel: Settings** – The right panel shows all the options to manipulate the discovery settings and analysis overlays. These settings help the user understand the logged behavior by providing near real-time interactions for filtering, reducing the visible complexity, and adjusting the model quality.

The three big sliders on top manipulate the *activity*, *path*, and *level depth* filters for discovery. The *activities* slider controls the fraction of activities that is included in the log and corresponds to the *Apply Filters* step in the Workbench Workflow. The *paths* slider controls the amount of noise filtering applied and corresponds to the infrequent discovery extension in the *Discovery* step in the Workbench Workflow. The *level depth* slider controls the amount of hierarchy levels shown/collapsed in the model. When exploring hierarchical models, the level depth slider allows for a coarse-grained depth filtering.

Below the sliders are several setting buttons. These buttons show the current settings for discovery and analysis. When you click on these buttons, a popout allows you to set detailed settings. There are four such settings buttons: *Discovery*, *Visualization*, *Analysis*, and *Export*.

**Discovery settings** – In the *discovery settings*, the user can switch between the Naive Hierarchical Discovery and Recursion Aware Discovery discovery algorithms. In addition, the user can change the cancelation trigger oracle from here. This way, the user can interactively experiment with the trigger oracle when needed. Furthermore, the cancelation extension as well as the process tree rewriting can be enabled/disabled from this menu.

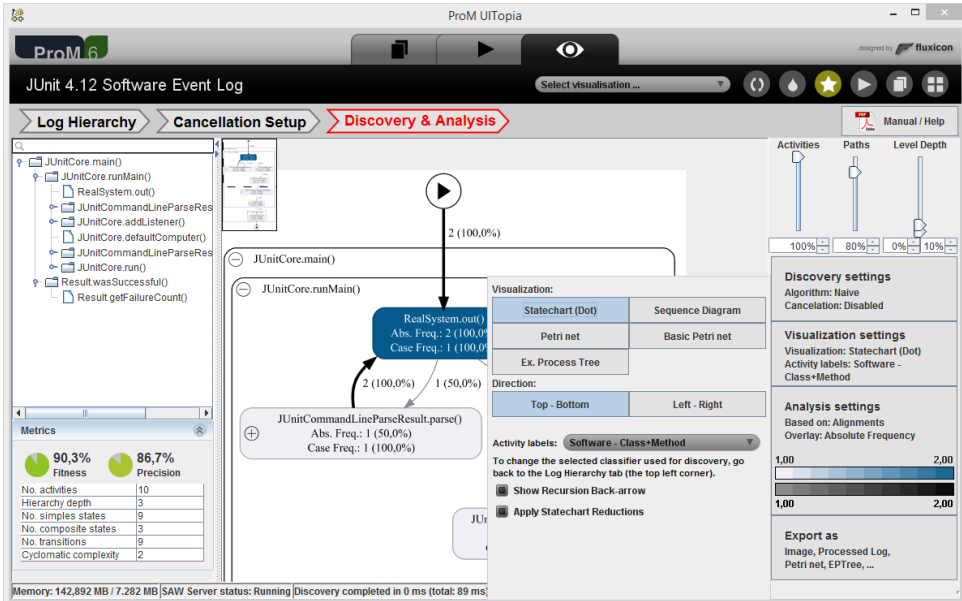
**Visualization settings** – In the *visualization settings*, the user chooses the type of model to model transformation to be used. Figure 10.8a shows the visualization settings panel and the available options. In Figures 10.6 and 10.8a, the statechart visualization is shown in action. In Figure 10.8b, the process tree visualization is shown in action. In addition, the user can choose the model layout orientation. Furthermore, the user can toggle the use of statechart rewrite rules (similar to the process tree rewrite rules), as well as a visual aid for recursions.

For software logs, several heuristics are available for reducing the size of the depicted activity labels/names without changing the discovered model, such as removing the package name from `package.class.method()` labels. Such reduced activity names can help simplify the model when using long labels as classifiers. This setting only changes the displayed/rendered labels via the label decorator in the *Rendering* Workbench Workflow step. The underlying model discovery still uses the canonical names provided by the original classifiers to distinguish, for example, similarly named classes in different packages.

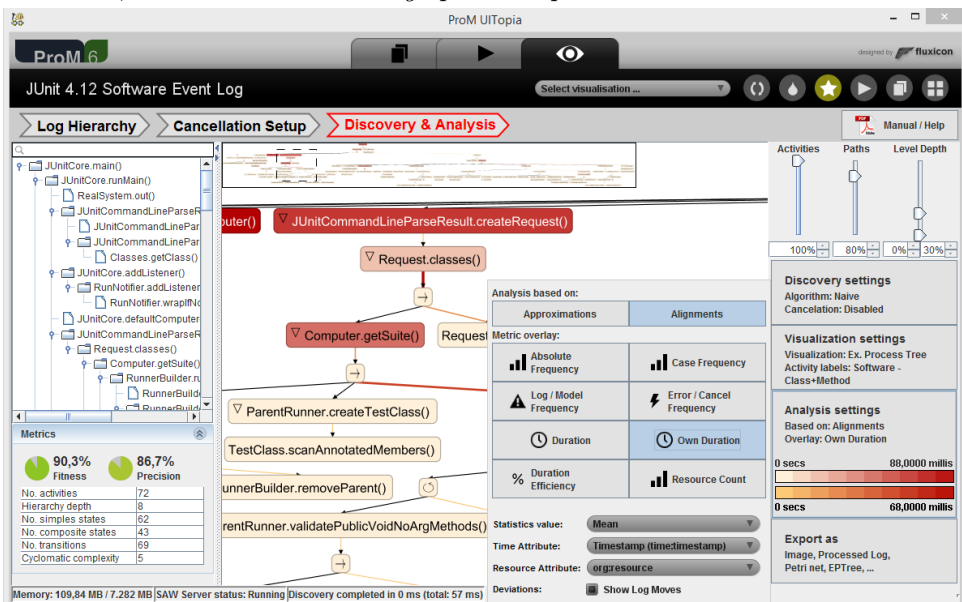
**Analysis settings** – In the *analysis settings*, the user can set up the metric overlays displayed on the model. Figure 10.8b shows the analysis settings panel and the available options. Different metrics can be calculated using either the fast approximation algorithm or the accurate alignment-based algorithms. The following metrics are provided in the Statechart Workbench:

- **Absolute Frequency** How often did an activity or method call occur in the entire event log? Implements Metric 8.1.
- **Case Frequency** In how many traces/cases did an activity or method call occur? Implements Metric 8.2.
- **Log/Model Frequency** How many deviations from the discovered model are present in the log? Implements Metric 8.3.
- **Error/Cancel Frequency** How many times did a cancel or exception trigger occur after a given activity? Implements Metric 8.5, configured to show how often an activity is followed by a cancelation region trigger.
- **Duration** What was the total duration of an activity or method call? Uses Metric 8.6 on the nodes, and uses Metric 8.7 on the arcs.





(a) Shown is the *Statechart* model visualization with the *Frequency* metric overlay. In addition, the *Visualization settings* panel is opened.



(b) Shown is the *Process tree* model visualization with the *Own Duration* metric overlay. In addition, the *Analysis settings* panel is opened.

**Figure 10.8:** Various model visualizations and settings panels available in the Discovery & Analysis screen in the Statechart Workbench.

- **Own Duration** What was the duration of an activity or method call, minus the time spent in lower method calls or submodels? Implements Metric 8.10.
- **Duration Efficiency** How much work was performed divided by how much time? This can give an efficiency indication in case of multithreaded code. Implements Metric 8.11.
- **Resource Count** How many resources or threads were used to perform the activity or method call? Implements Metric 8.4.

For the above metrics, the user can choose which event log attributes are to be used for resources and timestamps, allowing the use of, for example, the predefined *nanotime* in our XES extensions from Section 5.3. In addition, one can choose how to interpret multiset metrics (e.g., min, max, mean, etc.). Based on the chosen metric setup, the model will be overlaid with a heatmap via the color decorator in the *Rendering* workbench workflow step.

To complement the model move frequencies overlay, the user can toggle whether log moves should be rendered in the model. Based on the underlying Petri net places and markings, reusing the work from [21, 130], log moves are positioned inbetween the discovered control-flow elements.

**Export as** – In the *Export as* panel, the user can export the model and/or processed event log either as an image, or as a ProM object for further analysis.

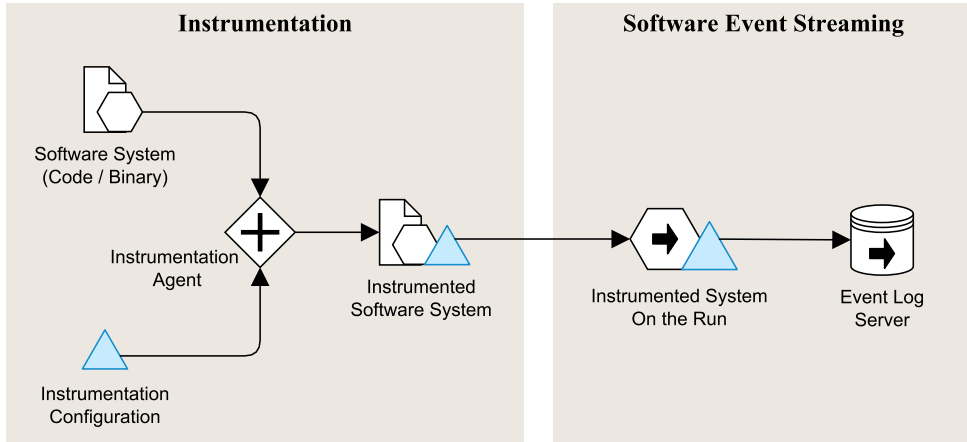
Finally, we included the user manual within the tool itself. See the Manual/Help button with pdf icon in the top right corner in Figures 10.5 through 10.8.

## 10.3 The Instrumentation Agent

The *Instrumentation Agent* tool provides a way to instrument Java programs, and log their execution to XES event logs. This Agent is a reference implementation to demonstrate how software event logging can be added to existing programs, and the ideas can easily be transferred to other programming languages. In Section 10.3.1, we will look at the Agent in more detail, explaining how the tool instruments a software system and generates events. In Section 10.3.2, we will briefly look at the accompanied *Event Log Server*. This server records the generated software events into an XES event log, which is directly usable inside the Statechart Workbench and other process mining tools. Figure 10.9 provides a schematic overview of the instrumentation process and shows the *Instrumentation Agent* and *Event Log Server* in context.

### 10.3.1 Inside the Instrumentation Agent

The Instrumentation Agent is a so-called *Java Agent*, a special Java binary designed to hook into the Java Virtual Machine (JVM). Via this hook, the Agent can observe and manipulate classes as they are loaded. The user can configure which classes should be manipulated to generate log events via a



**Figure 10.9:** The *Instrumentation Agent* process. Schematically is shown how the Agent is attached to a software system, also known as instrumentation. Next, when running the instrumented system, software events are generated and streamed to an event log server for processing.

configuration file. In this way, the Agent uses concepts from Aspect-Oriented Programming (AOP) to add software event logging to existing programs without modifying the code itself [65]. The generated log events are streamed to an external program to limit the amount of overhead on the instrumented/observed program. To attach the Instrumentation Agent binary `log_agent.jar` to a program binary called `myapp.jar`, one uses a command like the following:

```
java -javaagent:log_agent.jar -jar myapp.jar
```

Now, the Instrumentation Agent can manipulate the bytecode of the classes to be loaded for `myapp.jar`. The Javassist [52, 53] library is used to ease the manipulation of class bytecode. Via a special *Instrumentation Configuration* file, we can inform the Agent how the observed classes should be manipulated via instrumentation rules. Below, we discuss some of these AOP-inspired rules that can be used in the Instrumentation Configuration file. In these rules, we will reuse the terminology and event types first introduced in Section 5.3.1 on page 102. In addition, we will refer to Listing 10.1 for examples.

### Method Pointcut

The *Method Pointcut*<sup>2</sup> searches for specific method definitions, and instruments these methods such that the beginning/start and end/complete of these

<sup>2</sup> In Aspect-Oriented Programming terminology, a *pointcut* is a set of joinpoints or locations in a program where behavior or code such as event generation can be inserted.

**Listing 10.1** Example code snippet showing when which event types are generated where. This example was first introduced in Section 5.3.1 on page 102

```

1  class A {
2      void f(int y) { - - - - - call A.f
3          try {
4              - - - - - calling B.g
5                  int r = b.g(12, y);
6              - - - - - returning B.g
7          } catch (Exception e) { - - - - - handle in A.f
8              ...
9          }
10     } - - - - - return/throws A.f
11 }
12 class B {
13     int g(int x, int y) { - - - - - call B.g
14         return x / y;
15     } - - - - - return/throws B.g
16 }

```

methods generate events. The start, before the first line in the method definition, generates a **call** event. The end, right before a method returns or throws an exception, generates a **return** or **throws** event respectively. In the Method Pointcut, the user can configure specific name patterns for inclusion, optionally using wildcards. These name patterns are matched against the canonical `package.class.method(parameters)` name. Hence, the Method Pointcut can be used to instrument entire classes and (sub)packages. For example, in Listing 10.1, using `*` as a wildcard, the pattern `A.*` matches the method `A.f(int)` but not `B.g(int,int)`. In this example, the instrumentation would generate the events **call A.f** and **returns/throws A.f**, but not **call B.g** or **returns/throws B.g**.

Optionally, the user can indicate whether constructor methods and method parameters should be included in the event generation. In addition, the user can indicate if the start of `catch(Exception)` blocks should generate **handle** events, for example for cancellation discovery (Chapter 7). See for example `handle` in `A.f` on line 7 in Listing 10.1.

### Method-Call Pointcut

The *Method-Call Pointcut* searches for specific method calls in specific contexts. In contrast to the Method Pointcut above, a Method-Call Pointcut event is generated at the place where a method `f` calls a method `g`. The Method-Call Pointcut instruments a method `f` such that it generates **calling** and **returning** events for called/invoked methods `g`. In the Method-Call Pointcut, the user configures two name patterns for inclusion, optionally using wildcards. The *include* pattern searches for context methods `f` to consider for instrumenta-

tion. The *call* pattern searches for called/invoked methods *g* inside the body of method *f* for generating software events. Like before, the patterns are matched against the canonical `package.class.method(parameters)` name. For example, in Listing 10.1, using `*` as a wildcard, the *include* pattern `A.*` matches the method `A.f(int)` and the *call* pattern `B.*` matches the method `B.g(int,int)`. In this example, the instrumentation would generate the events **calling** `B.g` and **returning** `B.g` inside method `A.f(int)`.

There is a subtle difference between the Method and Method-Call approaches. Both trace the execution of called or invoked methods. However, the Method-Call Pointcut approach observes from the place where the method is being called, while the Method Pointcut approach observes from inside the called method. Recall the software location information from Section 5.3.3 on page 105. The Method-Call approach can track the *callee* and *caller* location, whereas the Method approach can only track the *callee* location. In addition, the Method approach resolves the location to the actual method being executed, whereas the Method-Call approach only records the referenced method definition, without any dynamic effects like overloading being resolved. For example, suppose that variable `b` on line 5 in Listing 10.1 is declared as type `class B` but points to an instance of `class C extends B`. Using the Method-Call approach, we would generate a **calling** `B.g` event, containing information from *callee* `B.g` and *caller* `A.f`. Next, using the Method approach, we would generate a **call** `C.g` event, containing only information from *callee* `C.g`.

### Interface Pointcut

The *Interface Pointcut* searches for implementations of specific interface methods. The instrumentation works exactly the same like the Method Pointcut and Method-Call Pointcut above. However, instead of matching on canonical method names, we specify an include pattern over canonical interface names `package.interface`. Whenever we consider a class `A` for instrumentation, we check all the interfaces `I` implemented by that class. Whenever an interface `I` matches the specified include pattern, we retrieve all the methods declared in that interface. For each of the interface methods retrieved this way, we instrument the corresponding method implementation in class `A` as if it were matched by a Method Pointcut or Method-Call Pointcut. Hence, this Pointcut can be used to instrument specific interfaces or APIs.

### Endpoint Pointcut

The *Endpoint Pointcut* searches for specific endpoint communication classes such as Sockets and classes implementing the Servlet interface. By using knowledge of the corresponding interfaces, this Pointcut generates **call** and **return** events with explicit communication details such as the local and remote IP address and port numbers used in the observed communications. This way, this

special Pointcut allows us to log enough information to track and correlate software events across application boundaries. We can leverage this additional information to, for example, infer *business transactions*, as discussed in Section 5.1.3 on page 95. See also the work in [119].

### 10.3.2 Logging Software Events

In the previous section, we discussed how the Instrumentation Agent can generate software events for specific locations in a program. When the instrumented software is running, a stream of software events is being broadcasted. One possible use of this software event stream is to apply online profiling analytics (e.g., like gprof [73] or Yourkit Profiler [136]), or applying online/streaming process mining techniques [8, 203, 205]. Another approach is to simply record the generated software events into an XES event log. The *Event Log Server* does exactly this, providing a XES log file which is directly usable inside the Statechart Workbench and other (traditional) process mining tools.

As simple as the concept of the Event Log Server sounds, there are various design decisions and challenges to account for. Although the details are out of scope for this thesis, we will sketch some of the challenges below.

First of all, one has to consider how the Instrumentation Agent and the Event Log Server communicate. Which technique and protocols should be used? What is the incurred overhead, and what effect does this have on (the performance of) the observed software system? How can the chosen technique and protocol be used to simultaneously track and log multiple concurrent execution threads? For example, our (naïve) Event Log Server implementation simply uses TCP or UDP sockets to listen to broadcasted software events. However, (Java) TCP connections are thread unsafe, and require a synchronization mechanism, introducing (performance) overhead on the side of the observed software.

Secondly, one has to consider how the Event Log Server tracks cases/traces. Do we allow the server to receive and observe software events from multiple software instances at the same time? Such features can be useful when we observe multiple users simultaneously interacting with our observed software systems. If we allow the tracking of multiple cases/traces, how do we buffer the software events and write the resulting (completed) cases/traces to disk? What are the limitations of the chosen approach in terms of the number of simultaneous cases/traces and maximum trace length we can track? Consider the size characteristics of software logs discussed in Section 5.2 on page 96.

With the Event Log Server, one can experiment with implementations of buffered and non-buffered XES loggers using both TCP and UDP implementations. As we will further discuss in Chapter 11, more research is needed on the scalability and tradeoffs of such (software) logging methods.

## 10.4 The SAW Eclipse Plugin

The SAW Eclipse plugin<sup>3</sup> enables the user to interactively link the results from the Statechart Workbench back to the source code of the system. Moreover, this Eclipse plugin integrates the Instrumentation Agent tool into the Eclipse IDE. In Section 10.4.1, we will briefly discuss the Instrumentation Agent integration provided by SAW. In Section 10.4.2, we will briefly discuss the Statechart Workbench integration provided by SAW.

### 10.4.1 The SAW Instrumentation Agent Integration

The SAW Eclipse plugin embeds the Instrumentation Agent and Event Log Server tools discussed in Section 10.3. Via convenient Eclipse *Run As* UI elements named *Java Application – Instrumented* and *JUnit Test – Instrumented*, the SAW plugin takes care of properly attaching the Instrumentation Agent binary to a selected program or unit test run respectively. In addition, the SAW plugin can automatically start and manage the accompanied Event Log Server in the background. This way, users can run their program or unit test as they normally would from their Eclipse IDE. The corresponding XES event log will simply appear in the Eclipse project folder upon termination, and can directly be imported into ProM.

The points in the software that will be instrumented and observed by the Agent is managed by the *Instrumentation Configuration* file as normal. The SAW Eclipse plugin can provide the user with visual aids, showing which locations in the source code will be instrumented upon run based on the actual Instrumentation Configuration file. Figure 10.10 shows these instrumentation visual aids in the various Eclipse views, such as the source code editor, the package explorer and the code outline.

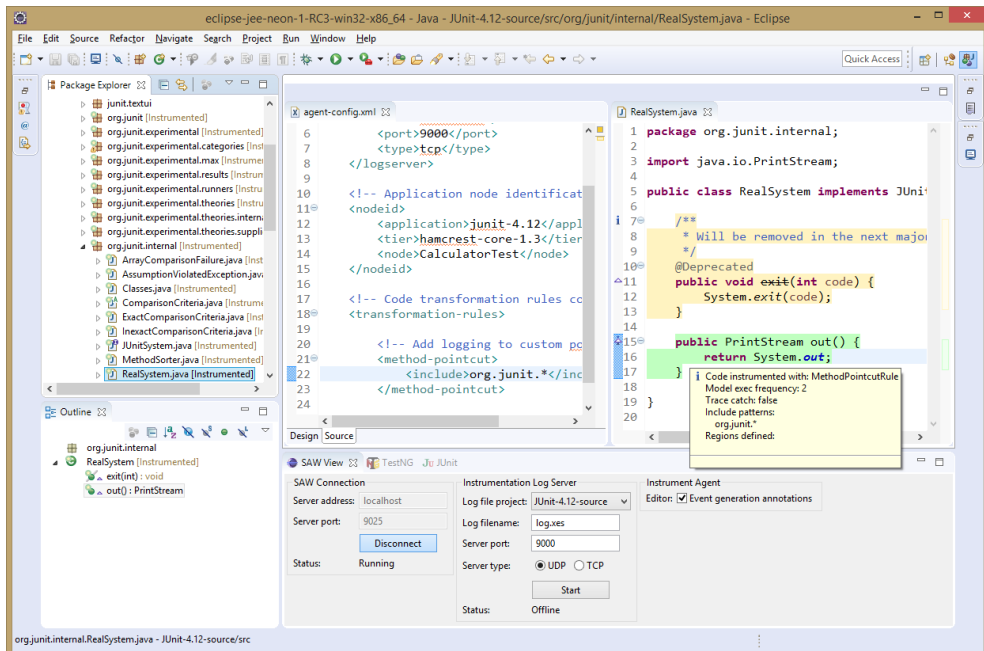
### 10.4.2 The ProM-Eclipse Link

The SAW Eclipse plugin also provides a link to the Statechart Workbench in ProM. Via a simple TCP connection, dubbed the ProM-Eclipse link, the SAW plugin can receive information and requests from the Statechart Workbench. Below, we will discuss two use cases using this ProM-Eclipse link.

First of all, the link allows the Statechart Workbench to request a specific source code line to be brought into focus. Upon receiving such a request, the SAW Eclipse plugin can locate the corresponding source code resource, and open an editor at the indicated line number. This way, the user can open specific locations in the source code corresponding to model elements in ProM using the available traceability information as discussed in Section 9.1.2. This

---

<sup>3</sup> SAW stands for Software Analysis Workbench. It reflects the vision to incorporate the Statechart Workbench tool into the Eclipse IDE, providing an integrated analysis solution.



**Figure 10.10:** The SAW Eclipse plugin. The SAW View panel at the bottom provides settings for the ProM-Eclipse link (SAW Connection) and the Instrumentation Agent and Log Server integration. The annotations in the Package Explorer and Outline to the left as well as the source code annotations to the right show the user which points/locations in the source code will be instrumented when executed. The background color and information tooltips provide additional details for each source code location, including information from the Statechart Workbench in ProM such as execution frequency.

allows users to, for example, inspect the source code corresponding to interesting control-flow elements, particular performance hotspots such as bottlenecks, or deviations such as outliers.

In addition, the ProM-Eclipse link allows the SAW plugin to receive information from the Statechart Workbench, and project it onto the source code resources. For example, Figure 10.10 shows how the frequency metrics calculated in the Statechart Workbench can be projected over source code locations using background colors and annotations with detailed tooltips.

Finally, the ProM-Eclipse link also allows information to be sent from Eclipse back to ProM. Although not used in the current implementations, this link allows, for example, to highlight parts in the process model with high complexity (for some source code based complexity metric), indicating where in the control-flow errors or mistakes are more likely. See also the discussion in Future Work 8.5 on page 257.

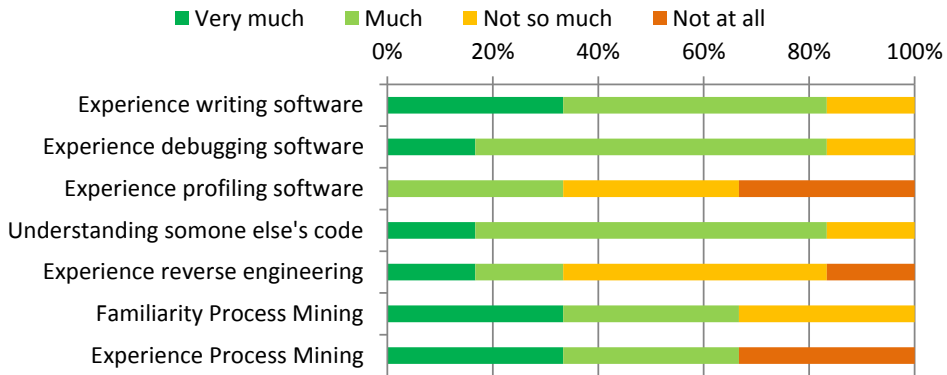


## 10.5 User Experience Evaluation

In this section, we present a preliminary user study of our tool and the user feedback obtained during this user experience evaluation. Rather than a large tool or user experience evaluation, this evaluation focusses on a small-scale assessment of the tools with the goal of gaining initial insight into the usefulness of and possible improvements for the proposed approaches and implementations. Below, we first present our methodology and setup (Section 10.5.1), after which we discuss the evaluation results and initial insights (Section 10.5.2). We conclude with a brief discussion on the threats to validity (Section 10.5.3).

### 10.5.1 Methodology

In this user study, we observed six participants using our tools and provided them with a little questionnaire before and afterwards. The participants were selected to have at least a master's degree in computer science, and to have several years of experience in developing software. At the start of the user study, we asked participants to rate their prior knowledge on a four-point Likert scale and list their expectations prior to seeing or using the tools. Figure 10.11 shows the obtained background knowledge distribution of the participants.



**Figure 10.11:** Background knowledge distribution of the participants.

After the initial questionnaire, we provided the participants with a pre-installed setup of our tool, the JUnit dataset [108], a list of instructions, and a short tutorial manual similar to the user manual [116]. Recall, the JUnit dataset consists of 946 events describing 182 unique activities (method call/return) with a discovered hierarchy depth of 25 nested submodels (i.e., nested method calls). The list of instructions explained the general purpose of the tools, the context, and lists several tips and questions designed to encourage participants to use and explore the tool. The provided tutorial manual explains how to access the plugin in ProM, explains the main screens, and describes the tool features to help users get started.

During the user study, we observed the participant's interaction with our tool and recorded what they were doing at which time. For these observations, we focussed on what the participants tried to do, what worked, what participants found confusing, and what they were missing.

After completing the tool tasks, we asked the participants to give feedback on the tool. We used this open-answer type of feedback to check hypotheses formed during observations, and obtain a view of how users experienced the tools. In addition, we asked the participants to fill out a user experience questionnaire using the methodology from [106]. In this questionnaire, user experience is rated via 26 items on a seven-stage scale, each represented by two terms with opposite meanings. These items are weighted and combined to rate the user experience on six scales:

- **Attractiveness** – What is the overall user impression of the tool? Do users like or dislike the tool?
- **Perspicuity** – Is it easy to get familiar with the tool? Is it easy to learn how to use the tool? Is there a (steep) learning curve?
- **Efficiency** – Can users solve their tasks with the tool without unnecessary effort? Is the tool perceived as responsive and practical?
- **Dependability** – Does the user feel in control of the interaction? Is the tool predictable and supportive?
- **Stimulation** – Is it exciting and motivating to use the tool?
- **Novelty** – Is the tool innovative and creative? Does the tool catch the interest of users?

To evaluate the overall user experience, we use the benchmark from [106], comparing our tool on the six scales mentioned above against a dataset from 246 product evaluations across 9905 participants<sup>4</sup>. This benchmark comparison gives us a better picture on the quality of our tool and offers a first indicator for what is perceived as good and what types of improvements can be made to improve the user experience.

### 10.5.2 Evaluation Results

Below, we first discuss the observations and the feedback obtained during the user study (Section 10.5.2). After that, we evaluate the user experience evaluation, and relate these results to the made observations (Section 10.5.2).

#### Observations and Feedback

Based on our observations, we could divide the participant's efforts into clear tasks. Participants approached each question in the list of instructions by a serie of tasks, divisible into initial tasks and later tasks. We noted that *the*

---

<sup>4</sup> Users form expectations during interactions with various software products. The question whether a product's user experience is sufficient can be estimated by comparing it to many other commonly used products [106].

*initial tasks are more explorative*: before diving into specific questions and details, the participants first needed to get their bearings and get an overview of the behavior and performance aspects. *Later tasks are more focussed*: the participants used their obtained overview to effectively zoom in to specific parts of the model. Here, we noted that for the initial tasks participants switched often between different types of model visualizations (e.g., statecharts, process trees, etc.). We concluded that for different kind of analysis efforts and questions, participants preferred different types of model visualizations. In later tasks, participants inspected the model visualizations in more detail. The named submodel expansion and collapsing and metric heatmaps were extensively used to manage and navigate the process complexity. Occasionally, participants switched and compared different model visualizations and the corresponding source code locations to confirm observations. That is, in the later tasks, participants build hypotheses based on one visualization, and check these hypotheses across different visualizations and between models and the underlying source code (in both directions). When asking participants about their favorite visualization settings, we were surprised by the popularity of the process tree visualization. Especially when annotated with various metric overlays, participants found this visualization very insightful. We expected to use the process tree visualization only internally for tool debugging purposes, but the participants informed us this tree visualization closely represents the software call graphs they were familiar with.

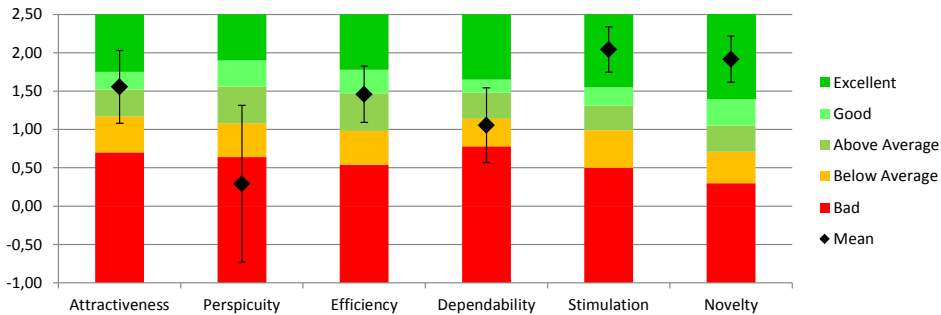
During the feedback session, the participants recognized and confirmed the *initial and later task* approach sketched above. Furthermore, the participants commented that the search, highlight and level depth filter (both sliders and the named submodel expand/collapse) aids were of great use. However, the participants also commented that these model exploration and filtering aids should be further improved. In particular, the search functionality should not only highlight parts of the model, but actively pan and zoom areas of interest into view focus. Furthermore, participants advised to include model-level filters, going beyond simple named submodel expand/collapse functionality and allowing them to hide and dismiss uninteresting areas in the control-flow model. Moreover, participants noted that such model-level filters could help them explore local performance metrics by excluding known or uninteresting bottlenecks in other parts of the model and rescaling performance heatmaps.

The participants noted that, although the basics of the tool were not perceived as difficult, many of the more advanced features and settings could benefit from more explanation, for example via legends, examples, and tooltips inside the UI. In addition, participants advised to improve the graph layout algorithm (Graphviz DOT [32]) to provide more consistent and predictable visualizations, especially for named submodel expand/collapse interactions. Furthermore, participants found the source code link to Eclipse really useful,

and suggested to improve and expand this type of integration between models and source code artifacts. Lastly, participants provided several suggestions for visualization improvements and different performance metrics. The cumulative duration metric (Metric 8.9) is one example result from this feedback session.

### User Experience Evaluation

Figure 10.12 shows the user experience questionnaire (UEQ) results for the tool user study. The comparison of the results for the tool with the data in the benchmark allows us to derive conclusions about the relative quality of the tool compared to other tools [106].



**Figure 10.12:** User experience questionnaire (UEQ) results for the tool user study. The black diamonds indicate the UEQ results with 95% confidence intervals. The background colors provide a comparison against the benchmark from [106], indicating the relative quality of the tool compared to other tools.

Overall, we got very positive feedback from our participants, as confirmed by the UEQ results in Figure 10.12. As we also observed, the participants really explored the data and rated our tool as very attractive, efficient, stimulating and novel. In particular, participants rated the multiple, linked visualizations and source code integration, as well as the interactive hierarchy folding and unfolding as indispensable, novel, and stimulating. This confirms the benefits of our guiding principle on page 279: it is worth the effort to provide a seamless integration and near real-time interaction with various models and an array of algorithms and techniques.

In the UEQ results, we noted a slightly lower dependability score, around the above/below average division. Based on the feedback results, we attribute this lower dependability score to a combination of several factors. The perceived learning curve and the need for more explanation (e.g., legends, examples, tooltips, etc.) cause the user to perceive the tool as less predictable and supportive. The non-determinism of the used graph layout algorithm (Graphviz DOT [32]) as well as the subtlety of the discovery parameters cause the user to feel less in control of the interactions.

Lastly, we noted a relatively low perspicuity score, with a relatively large confidence interval. This suggests a large variety in the perceived learning curve amongst participants. Participants noted two reasons for the perceived learning curve: 1) the extensiveness and complexity of the tool itself, and 2) the complexity of the discovered process models. We already commented on reason 1 with the need for more in-tool explanations. Reason 2 explains the large confidence interval. Participants have different expectations regarding how software should be visualized based on their prior experiences and backgrounds. When the visualizations did not match their expectations, they reported the presented process model as more complex and difficult, and thus a higher learning curve and lower perspicuity score. Although we accounted for these expectations with the various model visualizations provided, the UEQ results stressed how much these visualizations influence the usability of the tool. Despite these results, participants still indicated that they considered the tool to be more accessible than average (academic) tools and ProM plugins.

### 10.5.3 Threats to Validity

This section discusses the validity threats in our experiment and the manners in which we have addressed them.

**Internal Validity** – There exist two major validity threats with the above case study: 1) the objectivity of the users, and 2) the number of participants.

Naturally, due to our involvement and the use of open-answer type of questions at the end of the study, we risk affecting the objectivity of the users. We mitigated this threat by taking into the account the more unbiased responses users gave on the a priori expectations questions (written form) and the responses to the user experience questionnaire (filled in before the open-answer type of questions).

Secondly, the small sample of participants (six) also threatens the validity of the above results. Recall that the goal of this user study is to gain initial insight into the usefulness of and possible improvements for the proposed approaches and implementations. Hence, to (partly) mitigate this threat, we performed a quick confidence interval-based analysis, as provided by the user experience questionnaire framework [106]. Based on this analysis, we can conclude that only for the perspicuity scale we would need more participants for an accurate result; all other scales show a sufficient high agreement amongst participants. The low accuracy/large confidence interval of the perspicuity scale was already taken into account in the discussed above.

**External Validity** – The major external validity threat is the selection of the participants, i.e., to which degree can the conclusions from our user study be generalized? We partly addressed this threat by diversifying our participants on background knowledge, as discussed in Section 10.5.1 (see Fig-

ure 10.11). However, we acknowledge that with a larger follow-up user study, special care should be taken to select participants with more diversification in terms of background knowledge, experience, and demographics.

## 10.6 Conclusion

In this chapter, we presented three tools that implement and consolidate the discussed work (Contribution 6). The *Statechart Workbench* tool is a novel software process discovery, analysis, and exploration tool and implements the techniques from Chapters 6 to 9. The *Instrumentation Agent* tool is a Java agent capable of producing a ready-to-use XES event log when loaded with a Java program. The *SAW Eclipse plugin* provides a user interface for the Instrumentation Agent and enables the user to interactively link the results from the Statechart Workbench back to the source code locations inside the software system. A preliminary *user experience evaluation* showed the strengths and weaknesses of the proposed tools, and showed how the ideas from Chapters 6 to 9 are relevant and useful in practice. All of the above tools are open source and publicly available. In addition, documentation, readmes, and a tool screencast are provided.

With the tools and evaluation results presented in this chapter, there are several interesting research directions for future work.

### ■ Future Work 10.1 — Investigate Log Architecture Scalability and Tradeoffs.

As suggested in Section 10.3.2, there is a tradeoff between completeness and accuracy in logging software. To improve the scalability of logging, certain companies have employed a distributed logging infrastructure to mitigate some of the logging overhead. Typically, such distributed logging infrastructures are used to collect and store so-called clickstreams, i.e., streams of events capturing how a large number of users interact with a software system such as an online webservice. In such setups, an observed software system generating events distributes these events over multiple logging systems/servers, thus reducing the impact of high-velocity event generation [26]. However, to the best of our knowledge, no investigations have been made in employing such techniques for fine-grained software sampling and tracing. In short, how can such scalable logging architectures help in capturing more fine-grained software behavior with minimal impact on the observed performance? And what tradeoffs exist between scaling the logging infrastructure and the accuracy and completeness of observed events?

■ **Future Work 10.2 — Model-Level Filters.** With the discovery techniques and tools in this thesis, larger and more complex process models can be discovered and investigated. These large and complex process models call for novel techniques and aids in assisting the user to navigate the modeled behavior and

performance characteristics. As noted in Section 10.5.2, tool users found there is a need to apply model-level filters. That is, filters on parts of the model that help users explore local behavior and performance metrics, by excluding known or uninteresting behavior in other parts of the model, and thereby rescaling performance heatmaps. Such model-level filters should hide parts of the model indicated by the user, and not remove the corresponding event data from the analysis workflow. Ideally, the user is visually reminded of the hidden model elements in a fashion similar to the collapsed named submodels.

■ **Future Work 10.3 — Custom Traceability Annotations.** In Future Work 8.5, we already suggested to include traceability to the source code and other static artifacts. In the evaluation in this chapter, we observed the added benefit of source code traceability from the model to the code artifacts. In this case, the traceability was directly derived from available location information in the logged software events. However, system analysts and domain experts also have other types of traceability information available, for example from API descriptions, code generators, and domain-specific models. Hence, we propose to allow the software analysis tools to accept such notions via custom traceability annotations, possibly using an oracle or dictionary-like approach. This way, the discovered process models can be further enriched with links to other (static) artifacts and domain concept annotations.

■ **Future Work 10.4 — Linked Views.** In the evaluation in this chapter, we observed the added benefit of viewing the same behavior via different types of model visualization. In addition, we observed how users can benefit from the linking between the activity list on the left and model in the center of the Statechart Workbench (see page 286). Hence, we propose to extend these ideas to more advanced linked views, by allowing users to see different types of model visualization side by side, synchronizing focus, selections, highlights, and more across the different views.

*“The important thing is not to stop questioning.  
Curiosity has its own reason for existing.”*

— Albert Einstein

# 11 | The Software Process Analysis Methodology

In this chapter, we present a *methodology* for obtaining and analyzing software event log data in a structured way (Contribution 1). Section 11.1 motivates the need for a software process analysis methodology. After that, Section 11.2 introduces and describes the proposed methodology. Next, Section 11.3 discusses some practical concerns associated with the software process analysis methodology. Finally, Section 11.4 wraps up this chapter.

## 11.1 Introduction and Positioning

In this section, we introduce and position the need for a methodology for analysing software processes. Below, Section 11.1.1 starts by motivating the need for such a methodology. After that, Section 11.1.2 presents a small discussion of related work methodologies.

### 11.1.1 Why we Need a Software Process Analysis Methodology

Analyzing software starts with the actual, runnable software system itself. From this running software system, one needs to extract event data before any process mining can be applied. Extracting such event logs from running software systems is far from trivial (Challenge 6). Moreover, extracted software behavior and event data differ from business process event logs in several ways. Software event data has different properties and contains different patterns and structures not commonly found in business process event logs. In addition, the transformation of software logging data to event log traces is far from trivial. In practice, one has to decide on what constitutes a software event, and how events are grouped into cases/traces in a software-analysis setting. Chapter 5 already discussed some of these challenges. In addition, for a true round-trip analysis, any insights gained from the process mining analysis need to be related back to the software system domain. Hence, there is a need for tracing process mining results back to software artifacts like the actual source code that generated the event data (Challenge 1).



In industry, like with data mining some years ago [198], there is the expectation that process mining is a push-button technology. This is not true: process mining is complex and requires various tools, steps, and people. Moreover, as discussed above, when applying process mining for analyzing software, the process becomes even more complex. A methodology can help to understand, structure, and manage the interactions along such a complex process.

### 11.1.2 Related Methodologies

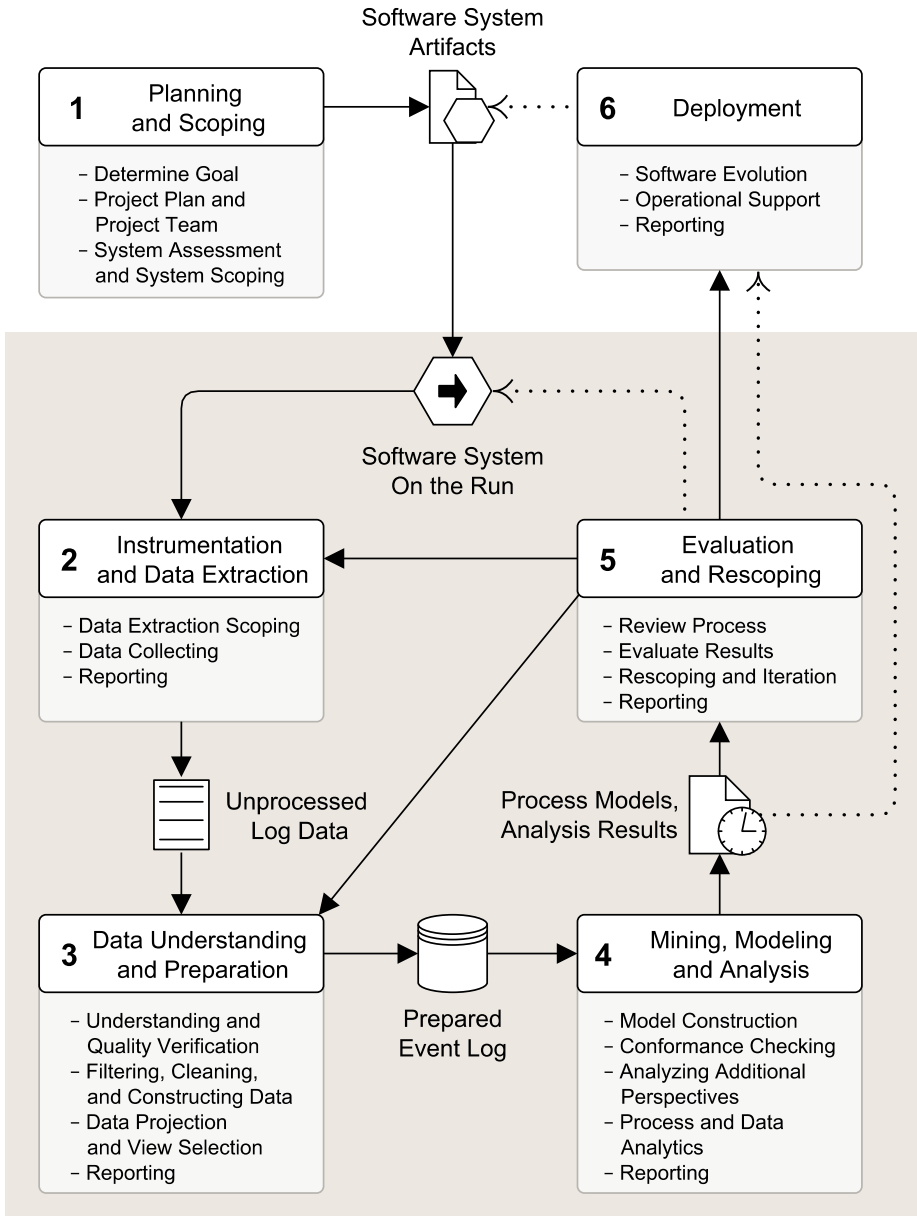
Within the fields of data and process mining, efforts have been made to establish methodologies to support practitioners, guide planning and execution, and save time and costs. Two widely used data mining methodologies are CRISP-DM [198] and SEMMA [102]. However, these methodologies are very high-level and provide little guidance for software or process mining specific activities [8]. Well-known process mining methodologies are the Process Diagnostics Method (PDM) [41], the L\* life-cycle model [8], and PM<sup>2</sup> [64]. The scope of PDM is limited to smaller projects; only a small number of techniques is covered. L\* is mainly designed for analyzing structured processes, but covers more techniques than PDM. To the best of our knowledge, PM<sup>2</sup> is the first process mining methodology to explicitly encourage iterative analysis, which is vital in any large and complex analysis project.

When analyzing large and complex software system processes, an iterative approach is vital. In contrast to traditional data and process mining projects, the software system under study can and should be included in the iterative analysis methodology. For large software systems, it is often infeasible and too costly to log everything and a scoping selection has to be made. By including the system under analysis, we can quickly obtain new data where needed, and investigate the system in different configurations and at different abstraction levels. *To the best of our knowledge, no existing methodology includes and leverages data generation (i.e., the software on the run) together with explicit knowledge about the software system (e.g., the source code) in the analysis.*

## 11.2 The Software Process Analysis Methodology

The *Software Process Analysis Methodology* is described in terms of six phases, and is summarized in Figure 11.1. For each phase, we collected a list of concrete tasks, and provide some pointers and common questions to consider. Observe that, in contrast to existing methodologies discussed above, the software system and the data extraction phase are *included* in the analysis cycle. By including the instrumentation of the system in the analysis cycle, we can quickly obtain new data where needed, and investigate the system in different configurations and at different abstraction levels. This way, the methodology allows for a fast feedback loop. The remainder of this section discusses each phase in detail.

### Software Processes Analysis Methodology



**Figure 11.1:** The Software Process Analysis Methodology. Shown are the six phases and main artifacts involved. The solid arrows indicate only the most important and frequent dependencies between phases. The dashed arrows suggest how the analysis results are related back to the software artifacts.

### 11.2.1 Phase 1 – Planning and Scoping

This initial phase focusses on understanding the business and project problem and goals as well as understanding the involved domain, software system, and the scope of the analysis. At the end of this phase, one has defined: a) a project plan, and b) a scoping of the software system included in the analysis. We recognize the following concrete tasks.

**Determine Goal** – First of all, one should *determine the goal* of the software analysis project. That is, one should agree upon an analysis direction, and select the right research questions. With every project, there are various stakeholders with different *business goals* and *business objectives*. Why does one want a software process analysis? What are the expectations? What outcomes and results are expected, and is there a *deployment vision*? Are we reengineering and documenting a legacy system? Are we seeking a formal model for some model-driven engineering efforts? Are we analyzing performance aspects for maintenance? Etc.

After a clear picture of the high-level goals and objectives is obtained, one needs to consider how these goals and objectives can be translated to *process mining goals*. Which software processes are we going to analyze? Are certain types of models expected to be discovered (e.g., behavioral process models, prediction models, etc.)? Are certain performance insights, behavioral pattern insights, or deviation insights expected? Together with establishing the process mining goals, one should agree upon an analysis direction and formulate a set of *research questions*. These research questions should be formulated such that they are related to the selected processes can be answered based on event data. That is, what are we looking for in the data? On which process or software elements should we focus? Etc.

**Project Plan and Project Team** – Before starting the analysis project, also a preliminary *project plan and initial assessment* should be made, taking into account all stakeholders. Based on the goals and objectives we determined above, one should make an *inventory of the available resources*. In addition, one should assess if there are additional *requirements* for the analysis, and which *constraints* need to be taken into account. For example, for our analysis, do we need access to certain systems? Are there special hardware platforms, machines, or computing time involved? Do we need time with specific domain experts, and do we have an associated cost, budget, or time constraint? Note that the process analysts are often not the domain experts for the system under study [41, 64]. Hence, the analysis should be performed in a *project team*, involving stakeholders and domain experts. In such a team, the domain expert does not need to completely understand the specific software being analyzed in detail. Instead, the expert could save time and effort in setting up the basic infrastructure and providing context for the analysis results. Based

on the above observations, we also need to investigate the associated *costs and benefits* as well as the involved *risks* and develop *contingency plans* where needed. E.g., what will we do if we cannot access the required domain expertise, tools, machines, or software platforms?

**System Assessment and System Scoping** – For a software process analysis, one should also perform a *system assessment and scoping* to get familiar with the *domain terminology and knowledge* as well as the *system architecture and design*. Not only should the analyst be able to translate analysis results to domain concepts to which the involved experts can relate. The analyst should also be able to make informed (analysis) decisions taking into account the software and system context. In addition, one should investigate the means by *how data can be retrieved*, i.e., a data retrieval strategy should be chosen [119]. Some examples of possible data sources were listed in Section 5.1.1 on page 91. Based on the above assessments, we need to perform the associated *initial tool assessments*. For example, are there certain software stacks or platforms we need to work with? Is there a specific environment needed for triggering the intended behavior? Is there a production, acceptance, or development environment that can be used? Is there a simulation or an adequate test suite that can be used? See also Section 5.1.2 on page 93. Note that a domain expert could save time and effort in setting up the basic observation infrastructure and data retrieval.

When selecting a data retrieval strategy, one should note the characteristics and limits of the obtained data. That is, an assessment of the *quality of the logging infrastructure and the retrieved data* should be made. For example, what is observed and logged, and what is not observed? Is the logging technique lossy, i.e., is it dropping events? Are there specific components that cannot be observed? Are specific hidden I/O channels influencing the data or control-flow, such as databases, files, etc.? How accurate is the obtained timing information, and which logging overhead is introduced? Is there a tradeoff between completeness and accuracy, e.g., to which degree is the granularity of logging affecting the observed timings? In a distributed system, are there local clocks and clock/timing synchronization issues to take into account? In Section 11.3 we will elaborate further on the above issues.

### 11.2.2 Phase 2 – Instrumentation and Data Extraction

Phase 2 is the first phase of the analysis cycle. This phase focusses on data generation: how and which log data is collected? At the end of this phase, one has a reproducible and documented dataset to be used in the upcoming data and process mining tasks. We recognize the following concrete tasks.

**Data Extraction Scoping** – First of all, a *data extraction scoping* should be determined. In the overall goal and system assessment of Phase 1, a spe-

cific software process and data retrieval strategy was selected. This process can be observed from many different angles at different levels of granularity. Furthermore, the exact scoping can differ per research question. That is, not all research questions can be answered with the same scoping and resulting data set. For example, which parts of the software, which components, and which interfaces should be logged? Are there different software versions and configurations to consider? When observing a software component, are we logging the internals of the component or only its external interfaces? Which data aspects do we include, e.g., interface parameters, data structures, etc.? Recall that there might be a tradeoff between completeness and accuracy. For broader comprehension and control-flow questions, a detailed scoping might be necessary, while for specific performance questions, a less intrusive scoping and sampling can be useful.

**Data Collection** – Next, *data collection* can start by running and observing the system. Care should be taken to verify if the correct data is logged. For example, when using logging with rotating logs, where log files are periodically archived or deleted, one should check no crucial information is missing or the wrong information is included. A good practice in such situations is to clear and reset the logging setup to a known state. When several logging sources are used, one should check how event data can be merged and combined. Maintaining the correct order of events when merging data sources is not trivial, especially when dealing with possible clock synchronization issues.

When collecting data about user or customer behavior, one should consider how long the data collection should last. That is, what is a good time window to observe? When collecting data about internal software behavior, one should consider if certain setups (e.g., test suites, simulations, etc.) should be executed multiple times. Especially when collecting data for performance research questions and when dealing with multi-threaded software, a statistically stable dataset should be created. In addition, one should be aware of possible sources of noise or deviating event data, and account for accordingly. For example, are there any background or scheduled tasks? What is the current network, cpu, and memory load on the used computing platform? Should we take warmup rounds into account (e.g., dynamic CPU clocking, binary and class loading, virtual machine optimizations, etc.)?

**Reporting** – For reproducibility and evaluation purposes, the actual instrumentation and data extraction process used, as well as the data generated, should be *reported*. One should record which software stack was used, which software versions, and in which configuration. In addition, one should record which deployment setup was used. Did one use any (API) mockups or simulated components? On which platform, with which hardware, was the software deployed? What environment was set up, including external libraries, databases, etc.? How was the data collected, and what choices were made?

Finally, one should also provide a short description and statistics of the obtained dataset. How many events, traces, or lines are there in the dataset? At which granularity was the data logged, and for which research questions was this data intended? By reporting these choices and properties during data extraction, one can clearly explain decisions and assumptions at a later stage to the involved domain experts.

### 11.2.3 Phase 3 – Data Understanding and Preparation

After obtaining the raw log data, one has to understand and preprocess the data. By selecting a case and event class/activity identification, one projects the data onto event logs according to different views. Note that the tasks in this phase are crucial for the success of the mining and analysis efforts. These tasks are often very time-consuming: we are dealing with large and complex data, and data preparation is an error-prone process. After this phase, one has various reproducible and documented event logs. We recognize the following concrete tasks.

**Understanding and Quality Verification** – First, one has to *understand and verify the quality* of the obtained data. Does the data make sense, considering the software system and setup used for extracting the data? Are common log size statistics within the expected range (e.g., number of events, traces/lines, labels/activities, etc.)? What patterns are there in the dataset? Are there any data quality problems? And what could be interesting projections and views for event logs? A good approach is to start with performing a quick spreadsheet analysis of the data to get a feeling for the log size statistics, and inspect a few sample data points. After that, a quick dotted chart analysis [172] using various interpretations can help to get a feeling of the overall dataset. The goal at this point is not to perform a detailed analysis, but rather to perform a quick sanity check to catch data extraction mistakes, determine useful data preparations, and save time later on.

**Filtering, Cleaning, and Constructing Data** – Usually, one has to *filter and clean* the dataset, and *construct additional data* [8, 64, 119]. This task can be performed both on the unprocessed data and on the obtained event logs. When a dataset was obtained from an existing logging or tracing infrastructure, one usually first needs to parse and reformat the data, and optionally add typing and semantical information. This is especially true when analysing an industrial software system, where unprocessed data is likely stored in a proprietary logging format. A simple model-to-model transformation or import script usually suffices for this task.

Filtering and selecting data is a frequently used data preparation task for reducing the logged complexity or focus the analysis on specific parts of the dataset. As one considers various research questions, one often needs to return

to the filtering to adjust the focus or try a different selection. In this task, events, traces, and/or attributes are removed based on various selection criteria and filters. For this task, many different ready-to-use filtering tools are implemented in ProM. One commonly used approach is borrowed from the process cubes approach [5]: *slice and dice operations*. These operations are based on attribute filtering. A *slice* produces a “slice” of the dataset by focussing on a specific value range for selected data attributes. In contrast, a *dice* divides a dataset into multiple smaller datasets by discriminating specific values for the selected data attributes. For example, one may slice a software dataset by focussing on specific interface names, and dice the dataset into different logs based on specific configurations, software versions, or product types. Another commonly used approach is to group similar traces, also known as *variants*, through techniques like clustering. In addition, certain rules can be used to select and filter event data, such as *compliance rules* and the *windowing and protocol run techniques* introduced in Section 5.1.3 on page 95.

In some cases, not all data is logged, and *missing data has to be reconstructed or completed* based on the rest of the dataset. Usually, some merging and aggregation of data has to be done. Events are sometimes logged at a too low abstraction level, or crucial parameters and data attributes are distributed over multiple events. For example, a sequence of interface calls might be about a particular product id, while only the first call (e.g., a define function, a constructor, etc.) explicitly provides the involved product id. In such cases, data can be inferred from the context and propagated accordingly. Lastly, one may also need to enrich the log and construct data. Newly added data can be derived from information in the log itself, or it can be added based on *external datasets, external models, or domain knowledge*. For example, an architectural description model could divide interface functions into conceptual phases or protocols. Such data is usually not (directly) logged, but can be easily added afterwards. On the one hand, such data could help in transferring insights between the process analyst and domain expert. And on the other hand, such data can help in the actual analysis phase by providing additional structure and information. In this example, such architectural phases or protocols can be used together with the hierarchical discovery techniques from Chapter 6 to discover more structured process models. For a practical example, see the industrial case study in Section 12.3.4 on page 343.

**Data Projection and View Selection** – Next, one applies a *data projection and view selection*. Depending on the research questions to be answered, obtained datasets can be projected onto multiple event logs, each focusing on a different view. A key question here is how to construct a case notion. Section 5.1.3 on page 94 already listed three common approaches for determining a case notion: *instance identifiers, business transactions, and windowing/protocol runs*. In addition, during projection, one defines event class identifiers,

i.e., what are the activity attributes? In a software event log, one typically can identify function names or process phases as an event class, but any other combination of attributes is also valid. Depending on the chosen projection and view, one can annotate the obtained cases with various meta attributes. For example, is a particular case referencing a product id, a particular batch, or a specific user? Is this case representing a particular configuration or simulation setup, and are there other context indicators?

**Reporting** – For reproducibility and evaluation purposes, the actual preprocessing and projection used, as well as the data produced, should be *reported*. From which datasets did we start? What is the quality of this dataset, and which fixes, if any, were applied? Which filterings and projections were used to narrow down the dataset? Which view was adopted and for which research questions was this event log intended? By reporting these choices and properties during data preparation, one can clearly explain decisions and assumptions at a later stage to the involved domain experts.

#### 11.2.4 Phase 4 – Mining, Modeling, and Analysis

After obtaining event logs, the actual data and process analysis can take place. In this phase, various mining, modeling, and analysis techniques are applied. The different models and analysis results produced yield new insights and new questions. After this phase, one has various reproducible and documented results. We recognize the following concrete tasks.

**Model Construction** – One usually starts with *model construction*, since most process mining techniques use a control-flow model as a basis. The model can either be *discovered* automatically or a reference model can be *constructed*. Reference models can originate from existing documentation and static artifacts, domain knowledge, be a result of process repairment, or a manual (reconstruction) effort. Ready-to-use discovery tools like the discovery techniques in this thesis, various ProM plugins (e.g., the Inductive Miner), or tools like Disco [76] are often very useful for obtaining a model of software behavior.

Note that, after discovering a software process model, one usually can perform a quick sanity check to verify if the preprocessing and discovery were performed correctly. For example, when a windowing or protocol run technique was used, one can verify whether the expected patterns are rediscovered. In addition, if certain patterns or constraints are described in existing (data) models, one can check if the discovered model complies with such patterns. For example, if one would observe an I/O software interface, the discovered model should require a constructor or a method named `open()` before a method named `close()` or `release()` is allowed. In short, parts of the observed software behavior is often known or can be inferred, and the the discovered model should a) make sense, b) verify the known parts, and c) elaborate on the missing details.



**Conformance Checking** – One can perform *conformance checking* to “confront” process models with real-life behavior captured in event logs. By aligning observed behavior with modeled behavior, one can gain detailed insight into commonalities and deviations between the logged and modeled behavior [21]. In addition, the model quality can be assessed (e.g., fitness and precision), and compliance rules can be automatically checked. In business processes, deviations or infrequent behavior often indicate certain situations where humans deviated from the prescribed process. In software processes, deviations or infrequent behavior is still generated by the underlying process and could indicate interesting edge cases. For example, if one would discover an 80/20 model to describe the mainstream (80%) behavior using a simple (20%) model, one could align the complete event log to spot where infrequent edge cases occur. Hence, one could argue that, in a software context, infrequent behavior can be more interesting than the mainstream behavior.

Finally, when using conformance techniques on a preconstructed model, deviation analysis can be used to perform automated model repair. In model repairment, the model is adapted to fit the logged behavior better. For example, in the context of software evolution, one could use model repairment to update existing documentation and models, incorporating the observed changes in the implementation. In contrast to discovery, one would keep the knowledge already contained in the documentation and improve where needed.

**Analyzing Additional Perspectives** – By aligning a control-flow model and event log, *additional perspectives* can be *analyzed*. Generally, it is advisable to first check if the model is as intended and correct, and only afterwards start the more costly alignment-based analyses. In such an alignment-based analysis, one is advised to first perform a quick sanity check using frequency analysis to see if the numbers for the main parts in the model make sense. After that, one can move on to an analysis addressing specific research questions.

*Frequency, usage, and probability statistics* can be projected onto various parts of the model. Such frequency analyses on software processes can reveal which control-flow paths are executed frequently. By using the additional location information such as package and class names, one can analyze frequency properties of various artifacts such as classes, see for example the Message Sequence Diagram approach from Section 9.3.4 on page 267. By using the traceability information available in the event log, one can analyze code execution coverage and frequency properties like code branches.

Next, also *performance characteristics* such as throughput times, durations, and other timing properties can be investigated in the context of the model. For a fine-grained software process analysis, we advise using the hierarchical performance analysis techniques from Chapter 8. That is, to cope with the complexity of the underlying software process model, one can use the hierarchical performance breakdown of, for example, the duration in the con-

text of both the call relation (hierarchy) and the control flow (choices, loops, recursions, etc.). This way, one starts with a coarse-grained overview of the performance, with most of the control-flow hidden in the hierarchy. Upon finding points of interest, one can unfold various parts of the hierarchy to perform a fine-grained performance analysis and investigate root causes for performance issues. For a practical example of such an analysis, see the JUnit case study in Section 12.2.3 on page 327.

In addition, using the recorded event attributes, *decision (guard-condition) mining* can be performed on the various modeled branches. Note that a modeled branch does not have to correspond to an if-else statement in the source code. Sometimes, a choice construct can emerge from, for example, a design pattern, where the different branches emerge from different implementation subclasses. For example, see the Composite design pattern in the JUnit case study in Section 12.2.4 on page 333. In other cases, a modeled branch can indicate differences in implementations across different software versions, configurations, or different types of input.

Next, based on recorded resource involvement, *resource behavior and usage can be derived and analyzed*. Depending on the meaning of recorded resources, this enables various types of analyses. For example, when using organizational mining techniques like social network mining [17], one can investigate how resources collaborate. In a software event log where the resources represent computing nodes (i.e., deployment resources), such a “social” network could explain how a collection of programs, servers, databases, and more are linked together and perform, much like the approach used in application performance management (APM) suites. By tracing the social network nodes via the underlying event log back to a control-flow model, one can relate deployment resources to a control-flow perspective. Alternatively, in a software event log where the resources represent threads or processes, such a social network could explain how multi-threaded software interacts, and complement notions such as duration efficiency (see Section 8.4.2 on page 230).

Finally, the *reliability* of various parts of the modeled behavior can be assessed by exploiting domain knowledge. For example, consider a software event log that recorded triggered and caught exceptions. By using a combination of cancelation discovery (Chapter 7) and a cancelation-trigger frequency metric (Followed-by Frequency, Section 8.4.2), one can get a breakdown across hierarchies of where in the software most of the exceptions occur, in terms of control-flow, packages, classes, etc.

**Process and Data Analytics** – In addition to the process mining tasks described above, various *process and data analytics* may prove useful. Typically, simple statistics such as averages, data mining techniques like clustering, and visual analytics such as scatter plots and histograms can quickly answer many simple research questions and provide a lot of insight. For example, as

noted already in Phase 3 above, a dotted chart analysis [172] is a simple and effective visual analytics approach to investigate various patterns.

**Reporting** – Again, for reproducibility and evaluation purposes, the actual mining and analysis process, as well as the models and results obtained should be *reported*. From which event logs did we start? Which log and which settings were used for model discover? Which logs were used in the various alignment-based analyses? When using the analysis tools, which assumptions were made? Based on the data, how can the various research questions be answered, and which hypotheses can be formulated? Based on the analysis results, which unexpected results were discovered? Which results should be discussed with domain experts? And which issues should be investigated in more detail using a different dataset? By reporting these choices and observations during the analysis, one has a good starting point for the evaluations in the next phase.

### 11.2.5 Phase 5 – Evaluation and Rescoping

At this phase in the project, one has produced various models, insights and analysis results. Before proceeding, one has to evaluate and discuss these results as well as the used analysis process. New insights and questions may yield new deployment efforts (Phase 6) and new data extraction and preparation efforts through rescoping (Phases 2 and 3). At the end of this phase, a decision should be reached on the deployment and the next steps to be taken. We recognize the following concrete tasks.

**Review Process** – Before proceeding, one has to *review the analysis process*. Do domain experts agree with the system scoping, the data extraction scoping, and the method of data collection? Were the right software versions and configuration used? Do the domain experts have any concerns based on the used approach? Are there known implementation or deployment quirks experts often deal with? Are there known software communication or timing issues experts point out? Furthermore, assumptions made during preparation and analysis should be verified. Are the used projections, views, filters and cleanup steps correct? And are there known issues with the logging infrastructure used which might have been overlooked? Often, such issues pop up in hindsight when reviewing the process and assumptions made. However, by leveraging the documentation from previous phases, and by performing multiple smaller analysis iterations, such issues can be detected during the process review and quickly be dealt with in a future iteration.

**Evaluate Results** – Next, it is important to *evaluate and discuss the analysis results*. Are the analysis results sound and explainable, i.e., do they make sense? Are the produced models and insights logical and do they make sense? One simple check is to compare the model sizes to metrics from the

underlying source code. For example, do the number of discovered activities match the expected number of software function calls? Although existing documentation might be absent or not up-to-date with their actual realization, domain experts can often point out if certain domain rules or expectations are violated in the discovered models. For example, consider the I/O software interface mentioned above in the *model construction* task of Phase 4. One should discuss, check and verify these results with various stakeholders and domain experts. In addition, often, *the discussions based on the produced models and results are more valuable than the actual models and results themselves*. Confronting domain experts with the analysis results can improve the domain expert's understanding of the system. In addition, the obtained results can result in key follow-up research questions or the need for verification on different settings and datasets. Such evaluation conclusions can lead to the rescoping and iteration efforts discussed below.

In addition to rescoping and iterating, a result evaluation can result into deployment efforts. Based on the obtained analysis results and answers to the research questions, which follow up actions can be formulated? That is, how can the original business objectives and deployment vision formulated in Phase 1 be realized using the obtained analysis results? We will elaborate further on possible follow-up actions in Phase 6 below.

**Rescoping and Iteration** – Based on the above results, evaluations, and discussions, new questions might arise and new *rescoping and iteration* may be in order. New research questions may result in new data projection and preparation efforts, focusing on different views. For example, consider the analysis of a dataset from an industrial software process where the case id was formed based on a product id and certain windowing rules. After discussion with domain experts, the question may arise what the discovered process looks like from a product batch perspective, or when using different windowing rules (i.e., using different assumptions).

Additionally, analysis results may prompt to observe and log the software at different abstraction levels, *selecting different parts of the software for the data extraction scoping*. For example, after analyzing a particular software interface or API, one might want to know what happens on a lower level during each of the observed interface functions. In such a situation, the corresponding lower-level data might not have been recorded upfront, and a new dataset has to be collected. As another example, after discussing the results of a particular software process, domain experts might be interested to see if the derived conclusions are specific for the chosen configuration or hold in general. In such a case, the analyst has to return to the software system, select a different software configuration, and repeat the analysis.

Performance investigations might call for *different environment enactments*, yielding *new data collection setups*. As was noted in Chapter 5, observing a

system changes the system [168]. Hence, it might be desirable to first discover a process model from a detailed tracing of a software system to obtain a highly accurate control-flow model. In a second iteration, one might want to use a less intrusive observation method such as sampling to obtain more reliable performance characteristics. The second, sampled log might not fit the original control-flow model perfectly, but gives a better indication of where the actual performance bottlenecks occur.

**Reporting** – At the end of this phase, the evaluation should be reviewed and *reported*, and a documented decision should be reached on the deployment and the next steps to be taken. Such reporting can be as simple as meeting minutes highlighting decision outcomes and listing the agreed upon action points plus responsibilities. In addition, one could also formulate common mistakes and lessons learned document to speed up future analysis efforts.

### 11.2.6 Phase 6 – Deployment

The models and insights obtained are typically the starting point for changes, and therefore not the end of the project. Usually, the knowledge gained needs to be presented and adopted. In addition to final reporting, software evolution projects (e.g., redesigning or adaptation) and operational support (e.g., monitoring or armoring) may need to be deployed. In many cases it will be the user, not the analyst, who will carry out the deployment steps [198]. We recognize the following concrete tasks.

**Software Evolution** – The analysis results can support and guide *software evolution*. For one, the discovered models can *support redesigning and refactoring efforts*. For example, suppose a control-flow and performance analysis revealed certain bottlenecks or inefficiencies in communication or multi-threading. The source code traceability can be used to quickly identify the related lines in source code artifacts, and thus where redesigning or refactoring is needed. Observe that, in contrast to profilers, process mining results can reveal that certain classes and methods are only problematic in certain states of the overall process, under specific control-flow states, conditions, or inputs.

Additional, *analysis results can be used and adapted for future model-driven engineering efforts*. Discovered control-flow models could be adapted for use in model-driven toolchains. Existing models can be adapted or repaired using the analysis results. Mined resource-oriented (social) networks can be used to improve deployment models.

Detailed analysis of test environment event logs and comparisons with production environment event logs can guide *test suite adaptations*. For example, analysis results based on operational, user, or factory conditions can be used to generate more complete test suites. Additionally, analysis of error logs can be used to create new bug-reproducing test code.

Lastly, other insights can be used for *future investigations*. For example, if an analysis was focussed on component X, and that component is somehow related to component Y, then the analysis could reveal potential problems, issues, or unexpected results about component Y. Hence, by analyzing component X, one might conclude that future investigations into component Y could be useful.

**Operational Support** – The models, results, and insights can be deployed in *operational support*. For one, identified problematic patterns, observed errors or other irregularities can be formulated for detection at runtime using *online monitoring*. For example, suppose that analysis reveals that in 10% of the cases, a software interface was used incorrectly or not as intended. In such a case, one could decide to setup a monitoring rule to detect the incorrect usage. Upon detection, one could log the events leading up to the problem, create a memory dump, signal an administrator, etc.

Alternatively, one could also deploy dedicated *conformance observer components* based on the analysis results and identified patterns. Such a component could actively check for correct interface or API usage, and halt the software when certain conformance rules are violated. Using such a setup, one could run various test suites, scenarios, and simulations to search for non-conforming software components.

Another usage is to use *future prediction and recommended actions* to help optimize production deployments. For example, suppose that, based on historical observation logs, after computation A happens, also computation B is likely to happen, e.g., based on common user patterns, external process elements, etc. When detecting computation A, one could also predict that computation B is likely to be requested, and hence we can preemptively compute the result for B where possible. Alternatively, suppose that historical API usage logs show common errors or mistakes made by API users. Then the API provider could use the historical data and analysis results to deploy a recommender that detects erroneous API usage, and recommend actions to solve the problems.

Finally, modeled behavior can be used in *component armoring*, i.e., to isolate software component faults such that components are protected from other, less-trusted components. Such an armoring can be designed based on both the intended behavior (possibly discovered from event logs) and the observed erroneous behavior (thus armoring against known issues).

**Reporting** – Knowledge and insights gained need to be *organized, presented and reported* in a way that the user or customer can use it. Note that, in many cases, it will be the user/domain expert, and not the analyst, who will carry out the above deployment steps [198]. This reporting can be as simple as a final presentation, or it can be an experience report, updated software documentation, or a deployment plan.

## 11.3 Practical Concerns

In the previous section, we detailed the different phases and tasks in the software process analysis methodology. As was already indicated via the questions mentioned throughout the methodology description, there are various issues to consider when analyzing software processes. In this section, we will discuss some of the more practical concerns in more detail. Section 11.3.1 will address problems related to incompleteness when observing software systems. Section 11.3.2 elaborates further on observing timings and performance in a software process context. Section 11.3.3 provides practical tips on including the software system and the data extraction phase in the analysis cycle. Finally, Section 11.3.4 suggests the use of visual workflows for reproducible and (partially) self-documenting (pre)processing.

### 11.3.1 Completeness of Software Observations

As discussed before, there are various ways to observe and log software behavior. When using an existing logging or tracing infrastructure, it is important to know not only what is observed, but also *what is not observed*. That is, which behavior and points in the software are not recorded? When using a custom or dedicated sampling or instrumentation technique, the same questions arise: what can we observe, and what is excluded? Moreover, especially when using sampling techniques, we need to take into account that some events are dropped and not recorded. That is, we need to be aware if the logging technique is lossy, and take appropriate actions. On the one hand, this may imply that we need to collect more data and run the software system multiple times, to minimize the risk of missing key observations. Simple metrics like code coverage can help in assessing the completeness of the obtained datasets. On the other hand, we need to take this type of noisiness into account when performing our preprocessing and process analysis. For example, when we know some events are dropped during logging, we should not aim for discovering a perfectly fitting process model. Otherwise, we would obtain a model that overfits to a noisy data source.

Elaborating on the notion of unobserved behavior sketched above, when observing software processes, one needs to be aware of possible *hidden influences*. For example, are there specific hidden (I/O) channels influencing the data or control-flow? One common hidden influence is the actions triggered by data in a database system or the signals resulting from updates in such a database system. Another common hidden influence is the exchange of data and action control signals across software processes via shared files. In addition, within a software process, shared data-structures are often used to influence the data and control-flow inside a process. For example, consider the queue data structure used in a typical producer-consumer setup to queue and trigger

computations. Besides being tricky to observe, such hidden channels create a type of indirectness in the control flow that is difficult to infer and analyze.

Another consideration when observing (distributed) software is that *the logged behavior may be over-complete*. For example, suppose a client invokes a remote procedure call, but due to some error (e.g., network communication failure), an error is returned and the remote procedure is not actually invoked. In such a case, the logged behavior may have recorded a start activity corresponding to invoking the remote procedure, possibly followed by an abort event or a complete event with a complementary error attribute to indicate that the remote call failed. When analyzing such over-complete event logs, one needs to choose how such failed actions are taken into account when discovering a model. One solution is to drop any failed remote procedure call event, including the associated incomplete start event. Note that such a solution is not always valid. Another solution could be to drop traces containing failed calls in order to mine the good weather behavior from completely successful traces only. In other cases, one might want to explicitly analyse such failures.

### 11.3.2 Timings and Clock Considerations

Another consideration is the accuracy of the observed timings. On the one hand, we need to consider how much the granularity of logging is affecting the observed timings. After all, observing, tracking, and recording behavior takes time and resources. In addition, in a distributed system, local clocks (i.e. the time on different platforms) can be different. In some situations, solutions like distributed clocks or a so-called heartbeat signal send over a common channel can mitigate timing issues. In other cases, one needs to (a-priori) adjust local clocks using, for example, a technique like the Network Time Protocol (NTP), as described in RFC 5905 [142].

When investigating performance issues, for example in a performance experiment setup, various other sources of inaccurate timings need to be considered. For example, are there any background or scheduled tasks? And what is the current network, cpu, and memory load on the computing platform used? When analysing logged timing data, we need to make sure the observed timings are due to the observed software system, and not some external tasks. In addition, one should consider if warmup rounds need to be taken into account. That is, is the used setup, programming language, and platform subjected to things like dynamic cpu clocking, binary and class loading, virtual machine optimizations (e.g., as in the Java JVM), etc. One possible solution is to close background tasks and bring the environment in rest, and run a performance experiment multiple times to get stable results. Another solution might be to run the performance experiment on multiple comparable (virtual) machines to eliminate machine-local factors. Note that, by using an event log as a basis



for performance analysis, we can combine the results from multiple traces and multiple software runs, and apply appropriate statistical analyses. Hence, we should make sure we collect a large enough dataset to draw statistically valid conclusions. One way to verify the statistical stability is by simply applying common outlier and distribution analyses on the obtained dataset before any more advanced (process mining) techniques are applied.

### 11.3.3 Iterate and Iterate Often

In practice, the road from system and data extraction scoping to actual process and data analysis is complicated. Usually, there is no straightforward way to select the right scoping, projection, and view, even though these decisions are critical for a successful process and data analysis. An analyst first has to get to know the system being analysed, and build up the knowledge required to achieve a high-quality analysis. In addition, as we already observed in Chapter 5, software data tend to be large and complex, which further complicates the job of the software analyst. Therefore, we advise to perform a so-called *dry-test analysis* first. That is, to start with a highly-controllable environment, such as a development environment, and relatively small dataset. This way, one can get a sample of the type of problems that need to be overcome, and it is easier to perform the various sanity-checks mentioned in the methodology above. The purpose of such a dry-test analysis is not to (directly) answer the formulated research questions, but rather to build trust in the analysis and constructed assumptions, preprocessing, and analysis setups. Following up on a dry-test analysis, we recommend to use multiple smaller iterations rather than a few large big-bang iterations. Especially in software process analysis, the size and complexity of the obtained datasets can easily be overwhelming. Do not attempt to log everything the first time, but build trust in the performed analysis steps, and go back to the software system to generate and collect more data only when needed. In addition, it is important to remember that *there usually is no single golden model and event log that can clearly and accurately explain the whole software behavior*. In a software process analysis, we often can and should return to the software system on the run to collect different datasets, discover different models, and focus on various views. That is, we often can and should iterate, and we should iterate often in small increments.

### 11.3.4 Visual Analysis Workflow

In the methodology above, we noted the need for proper reporting for reproducibility and evaluation purposes as well as documenting decisions and assumptions. In addition, as discussed above, we noted that in practice, many iterations of data extraction, data preparation, and analysis will be needed. Visual workflow tools such as KNIME [99], RapidMiner [161], and Rapid-

ProM [160] allow for reproducible and (partially) self-documenting (pre)processing. Such tools provide a block-based visual workflow where common tasks and algorithms can visually be chained together. This way, such tools provide an easy way of keeping track of all the applied data preparation and analysis algorithms. In addition, such block-based approaches often allow for templating or the construction of custom, domain-specific building blocks. This way, domain-specific tasks can be isolated, defined, templated, and documented.

## 11.4 Conclusion

In this chapter, we presented a methodology for obtaining and analyzing software event log data in a structured way (Contribution 1). In contrast to existing methodologies, the *software process analysis methodology* actively includes the system under analysis. In addition, this chapter addressed various practical concerns such as timing issues in software systems, log completeness, and the notion of *dry-test analysis*.

With the methodology and approaches presented in this chapter, there are several interesting research directions for future work.

■ **Future Work 11.1 — Investigate Logging Scalability and Tradeoffs.** As suggested in Phase 1 of the above methodology, there is a tradeoff between completeness and accuracy in logging software. That is, the granularity of logging can affect the observed behavior and the observed timings. In Future Work 10.1, we already suggested to investigate possible techniques for improving the logging tools. However, tool improvements are only part of the solution. How is a software analyst to know how much can be observed and logged, and what the effect is of fine-grained software sampling and tracing on the observed behavior and performance? What scalability concerns should a software analyst be aware of? What guidelines can be established in aiding a software analyst in selecting the right tradeoffs? To the best of our knowledge, no investigations have been made in such scalability concerns and tradeoffs.

■ **Future Work 11.2 — Investigate Process-Oriented Code Coverage Notions.** As noted in Section 11.3.1, when sampling software behavior, we can collect more data and run the software system multiple times, to minimize the risk of missing key observations. We suggested to employ simple metrics like code coverage to help in assessing the completeness of the obtained datasets. However, traditional code coverage metrics, such as line and statement coverage, are focussed on the source code artifacts. As we have discussed in this thesis, when analyzing the processes, there exist various other notions of completeness. See for example the directly-follows completeness presented in Definition 4.3.1 on page 85. Therefore, we suggest that future work should look into completeness or (code) coverage metrics that strike a balance between a source code ori-

ented view and a process oriented view. That is, how do we define the notion of completeness of a software event log, possible with respect to the underlying source code, for the purpose of software process analysis?

*“Wisdom comes from experience.  
Experience is often a result  
of lack of wisdom.”*

— Terry Pratchett

## 12 | Case Studies

In this chapter, we present two case studies, showing how our techniques can be used in practice. Section 12.1 provides a short introduction to the case study and evaluation setup in general. After that, Section 12.2 presents an open source software case study on the JUnit 4.12 library. Section 12.3 presents an industrial software case study on the wafer handling process inside ASML’s lithography machines. Finally, Section 12.4 wraps up this chapter.

### 12.1 Introduction

In this chapter, we discuss two software process analysis case studies. The first case study (Section 12.2) analyzes the JUnit 4.12 library, which is publicly available and open source. The second case study (Section 12.3) analyzes part of the wafer handling process inside ASML’s lithography machines. This industrial case study was conducted by the author in collaboration with a team of ASML software engineers. Both case studies are typical software process analyses: in accordance with the discussion in Section 5.2 on page 96, these analyses illustrate how the observed behavior complexity and log size characteristics in software process analysis differs from traditional business process analysis. In these case studies, we show how the ideas from this thesis can be used to perform software process analyses on real-life open source and industrial software systems. In the case studies below, we focus on the validation and applicability of the proposed techniques, tools, and methodology.

The methodology discussed in Chapter 11 will be used to structure the case studies. Note that the methodology described in the previous chapter is actually a result from the experiences obtained during these case studies.

### 12.2 Open Source Software Case Study – The JUnit 4.12 Library

In this case study, we analyzed the inner workings of the JUnit 4.12 library. This case study is related to the setup used in the user experience evaluation from Chapter 10. Below, Section 12.2.1 gives an introduction to the case study. Section 12.2.2 describes how the initial data extraction and understanding was

performed. Section 12.2.3 presents the initial high-level process exploration and understanding. Section 12.2.4 presents some of the detailed findings in the context of well-known software design patterns. Finally, Section 12.2.5 briefly discusses the threats to validity.

### 12.2.1 Case Description

JUnit is an open source unit testing framework for the Java programming language. A programmer writes and annotates special unit test methods alongside a Java program. In order to run these unit tests, the JUnit library is linked to a software project at compile time and inspects the software at runtime for the annotated test methods to execute. After running the unit tests, the JUnit Library outputs the test results: which unit tests succeeded, which failed, and optionally the exception stack trace for a failed unit test. In JUnit, test methods are organized into test classes, and several test classes can be combined into test suites.

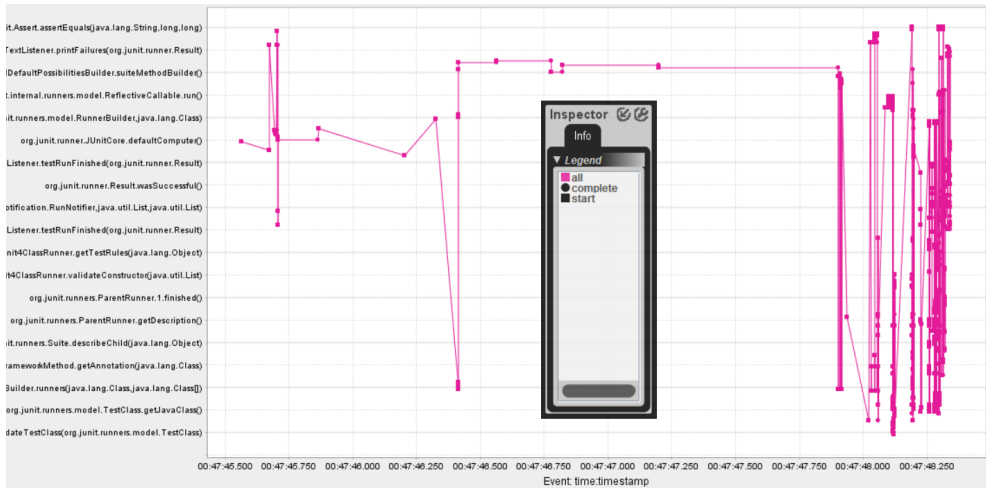
In this case study, we looked at the inner workings of the JUnit 4.12 library. During the case study, we leveraged the access and traceability to the underlying source code by using the ProM-Eclipse source code link in our tools. In terms of existing documentation, we only relied on general descriptions like the one in the previous paragraph.

For this case study, we aimed to perform an exploratory analysis. The main research questions in this case study are: *What is the global process of the observed software? And what parts of the software are the main contributors to the total runtime duration?* In addition, during a more detailed exploration, we tried to uncover *how various aspects of the JUnit behavior is implemented, and if specific design patterns are used to realize this behavior.*

### 12.2.2 Initial Data Extraction and Understanding

As stated above, we looked at the inner workings of the JUnit 4.12 library, available via [67]. We used the example input found at [40] to trigger actual unit test behavior. Based on a quick source code package inspection, we configured the Instrumentation Agent tool to record a XES event log capturing method-call level events for all classes inside the `org.junit.*` package and its subpackages. Since we are focussing on the overall behavior in this first analysis iteration, we tried to trace all behavior for a small input, and we take any performance overhead incurred by the tracing for granted.

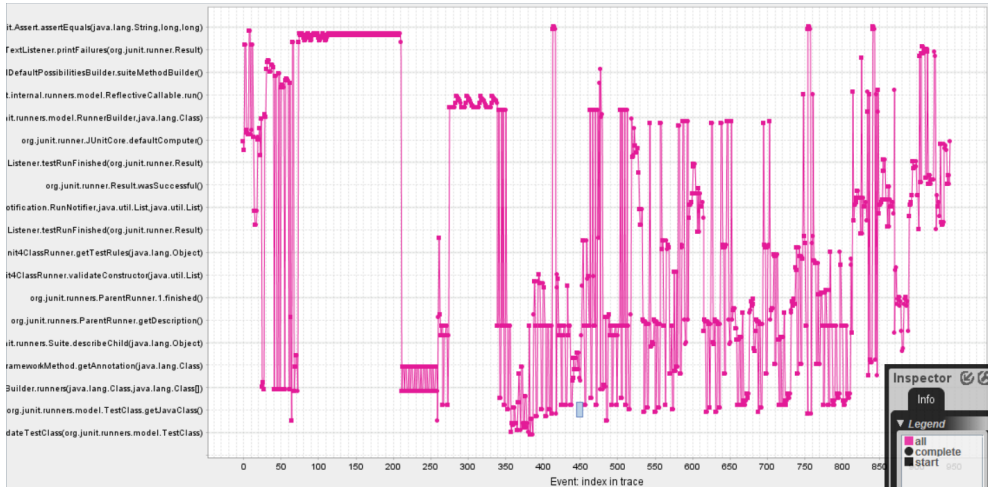
The obtained XES event log contains 1 trace with 946 events covering 182 unique methods/activities. A quick scan of the traced events and the recorded activity and lifecycle information using the ProM *Log Visualizer* suggest that a nested call pattern was recorded as expected from the tracing setup, and that no software exceptions were recorded.



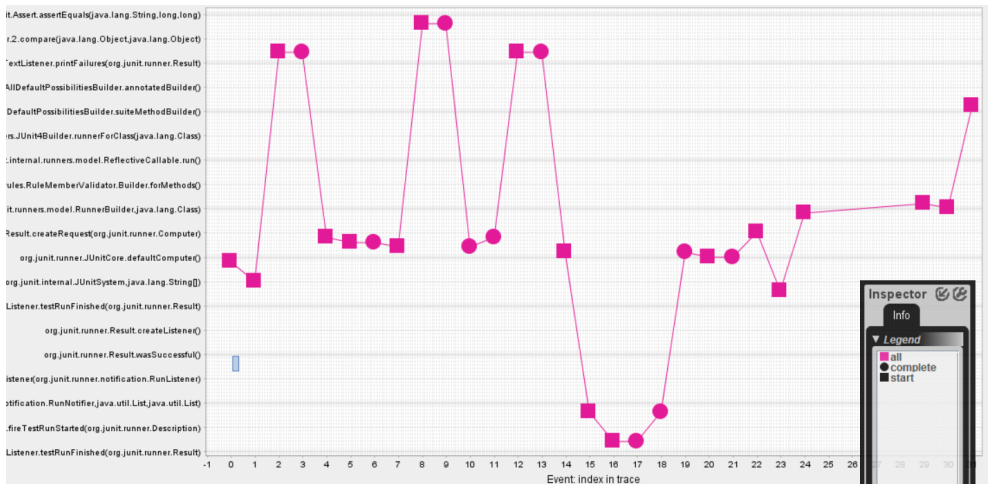
**Figure 12.1:** The time-based Dotted Chart [172] view of the JUnit event log. Each dot in this scatterplot is an event, with the timestamps of events on the x-axis and activities on the y-axis. The shapes the lifecycle-transition, and the lines indicate the directly-follows relation.

In order to get a feeling for the overall event log and to perform a quick sanity check, as suggested by the methodology from Chapter 11, we performed an initial Dotted Chart analysis. Since we wanted to get a feeling for the type of events recorded in the single software run, we projected the method activity names onto the y-axis. We used the default timestamp on the x-axis. To further inspect the nested call pattern, we used shapes for the method call/start and return/complete. Figure 12.1 shows the resulting Dotted Chart scatterplot. Time-wise, we can draw a vertical division at approximately three-quarters on the x-axis. To the left of this division, there are only a couple of method calls, and to the right of the division, we see a dense collection of method calls. This indicates most of the time is spent on the first few method calls, i.e., the startup phase of the observed JUnit software process. With the tracing setup used for obtaining this event log, it is likely that the initial overhead we observe here is caused by the class loading or initial tracing performance overhead.

In the current Dotted Chart, the x-axis does not allow us to inspect individual software events using time: events happen within milliseconds after another and are drawn on the same x-location, visually overlapping each other. By switching the x-axis to the index of events in the trace, we obtain the scatterplots shown in Figure 12.2. These plots show the rather structured passing of control between the various methods. After zooming in, Figure 12.2b seems to confirm a nested call pattern in the data: various methods are being started and completed between the start (square) and complete (circle) of a specific



(a) Complete overview of the event log.



(b) Zoomed in view on top left part.

**Figure 12.2:** The index-based Dotted Chart [172] view of the JUnit event log. Each dot in this scatterplot is an event, with the index of events in the trace on the x-axis and activities on the y-axis. The shapes the lifecycle-transition, and the lines indicate the directly-follows relation.

method. For example, see the start-start-complete-complete pattern at the bottom in Figure 12.2b, near  $x = 15$  and  $x = 18$ .

Based on the above observations, we have gained an idea of the quality and overall characteristics of the obtained event log. We conclude that the software behavior is recorded as intended, although the recorded performance information is probably less reliable. Based on the observed nested call patterns and number of unique activities, we predict that traditional process discovery algorithms will have difficulties discovering a meaningful process model.<sup>1</sup> However, a hierarchical discovery approach is likely to be more successful. At this stage, we have found no reason to apply custom filtering or preprocessing steps on this event log, and proceed with process discovery.

### 12.2.3 High-Level Process Understanding

To obtain an initial, hierarchical process model, we loaded the JUnit event log into our Statechart Workbench tool and used the *Software Log* preset. We shortened the shown activity labels by removing the package names and obtained the initial model shown in Figure 12.3. We immediately observe that, without much effort, we get a sensible model. The Statechart model depicted is well-structured, and contains a nice hierarchy based on the method call relations: there are no large self-loops or parallel compositions, which are often indicative for non-block-structured or non-fitting behavior.<sup>2</sup> Hence, we hypothesize that the obtained model is a well-structured representation of the obtained event log, and thus also fits the underlying software process well.

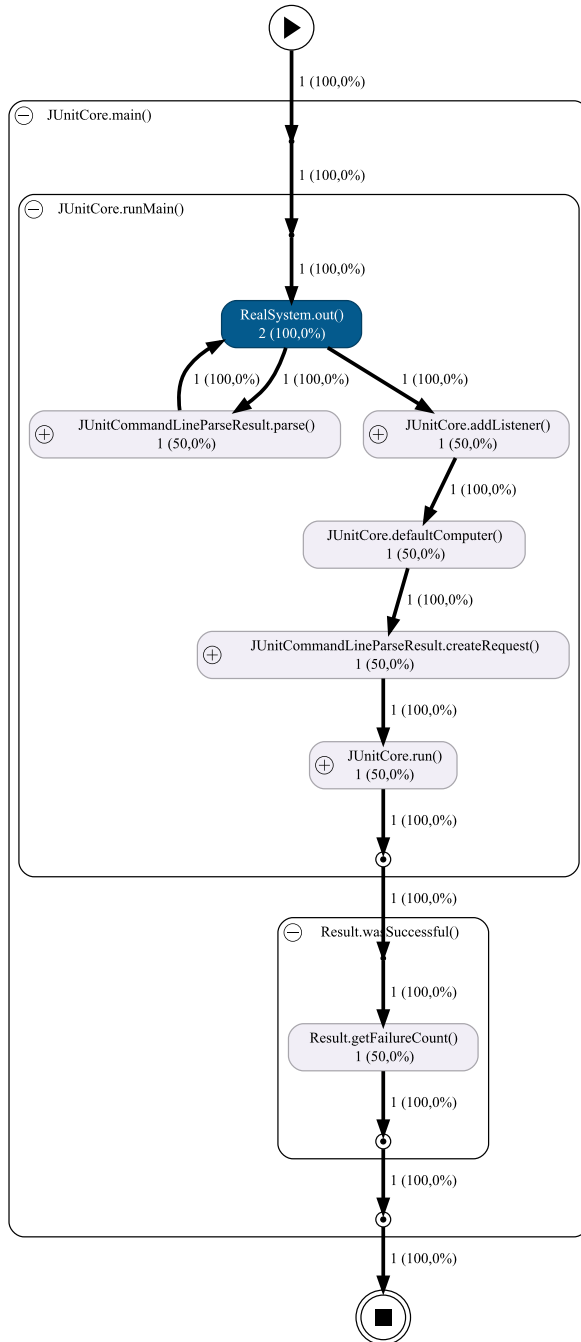
Based on this initial model, we can address one of our main research questions. The global process of the software starts with the `JUnitCore.main()` method invoking the `JUnitCore.runMain()` method. During this `runMain()`, four actions are performed in order. First, the command-line input is parsed via the method `JUnitCommandLineParseResult.parse()`. Next, the JUnit internals are set up and configured via `JUnitCore.addListenser()` and `JUnitCore.defaultComputer()`. After that, a test request is prepared by invoking the method `JUnitCommandLineParseResult.createRequest()` and runned via `JUnitCore.run()`. After `runMain()` returned, the overall unit test result is checked and reported via `Result.wasSuccessful()`. Using the expand/collapse functionality for named submodels, we can inspect each of the above methods in further detail. For example, by inspecting the method `JUnitCommandLineParseResult.createRequest()` in detail, we can clearly see how, during request creation, the test classes are scanned for annotated test methods to run as unit tests, e.g., see `TestClass.scanAnnotatedMembers()` in Figure 12.4.

---

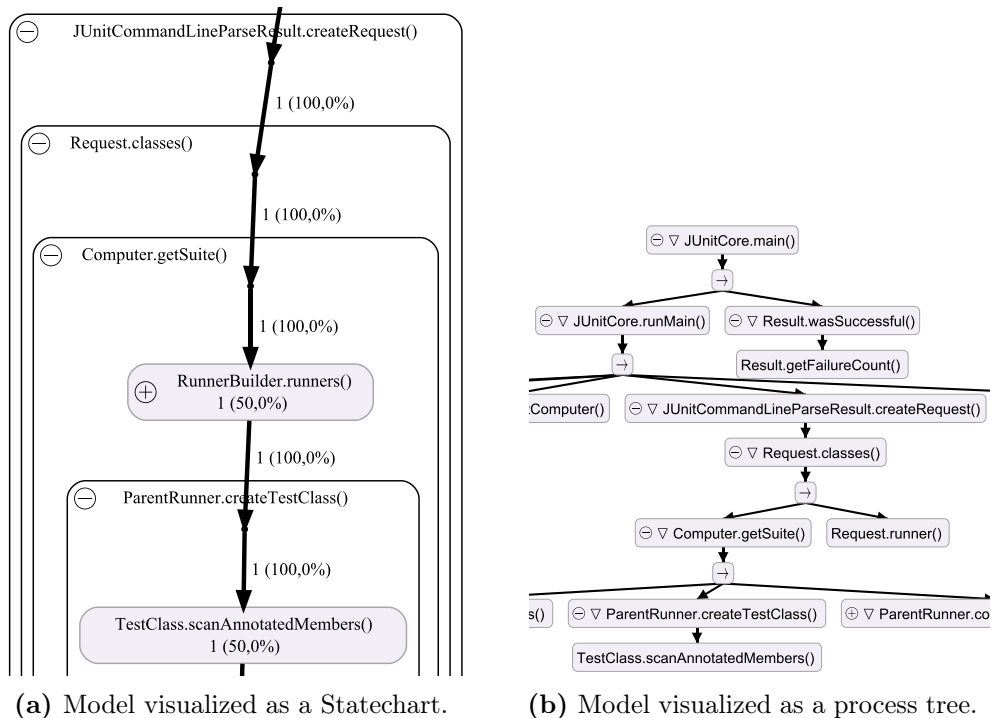
<sup>1</sup> Recall the observations and lessons learned regarding nested interval patterns from the sythetic hierarchical discovery evaluation in Section 6.7.1 on page 138.

<sup>2</sup> Large self-loops and parallel compositions are often introduced by the Fallback cases from Section 4.2.5 on page 81, and thus indicate non-block-structured or non-fitting behavior.





**Figure 12.3:** The initial model for the JUnit log obtained via the Statechart Workbench. The *Software Log* preset and default parameter settings are used. The activity labels depicted are shortened by removing the package names.



**Figure 12.4:** Part of the JUnit model, showing when annotated methods are scanned. The *Software Log* preset and default parameter settings are used. The activity labels shown are shortened by removing the package names. The absolute frequency metric overlay is computed via approximations.

Using this initial model as a basis, we turned on the alignment-based duration metric overlay to investigate the overall duration. Based on the alignments, we obtained a fitness score of 89,7% and a precision score of 86,7%. These scores confirm the hypothesis that the model shown is a well-structured representation of the obtained event log, i.e., the model represents the behavior in the event log well and in a structured manner.

Using the process tree visualization with the duration overlay, we expanded parts of the hierarchy based on the named submodels that are highlighted red, indicating a relatively high duration. Figure 12.5 shows part of the process tree obtained in this way. In this model, we can clearly see a red vertical path across the tree, showing a root cause breakdown for the main contributor to the total runtime duration. During `runMain()`, the test request preparation in the method `JUnitCommandLineParseResult.createRequest()` accounts for most of the duration. Exploring the submodels for request preparation reveals that most time is spent in the `RunnerBuilder` and `AllDefaultPossibilitiesBuilder` classes. A

quick code inspection of these classes using the ProM-Eclipse link reveals that during these process steps, data-structures are allocated and filled with information based on reflection operations used on the unit test classes. From the Java documentation for reflection, we know that reflection operations incur a significant performance overhead [155]. Based on this information, it is likely that the main causes for the majority of the runtime duration is a combination of object creation and reflection operations used to scan test classes for annotated unit test methods. However, recall from the Dotted Chart observations that the overhead we observe here could be caused by the initial class loading or tracing performance overhead. Hence, to confirm this hypothesis, we would need to collect additional software event data using a sampling setup with lower performance overhead and repeat this analysis. Note that the new event log would be less detailed in terms of control flow. Obtaining a high-quality performance-oriented event log is future work.

So far, we have shown how our hierarchical discovery and performance analysis techniques can be applied to overcome some of the challenges encountered when dealing with software event logs. In addition, based on the performance observations, we saw why the software system and the data extraction phase should be *included* in the analysis cycle, as argued in the methodology discussed in Chapter 11. In the next section, we will highlight some of the detailed findings in the context of well-known software design patterns.

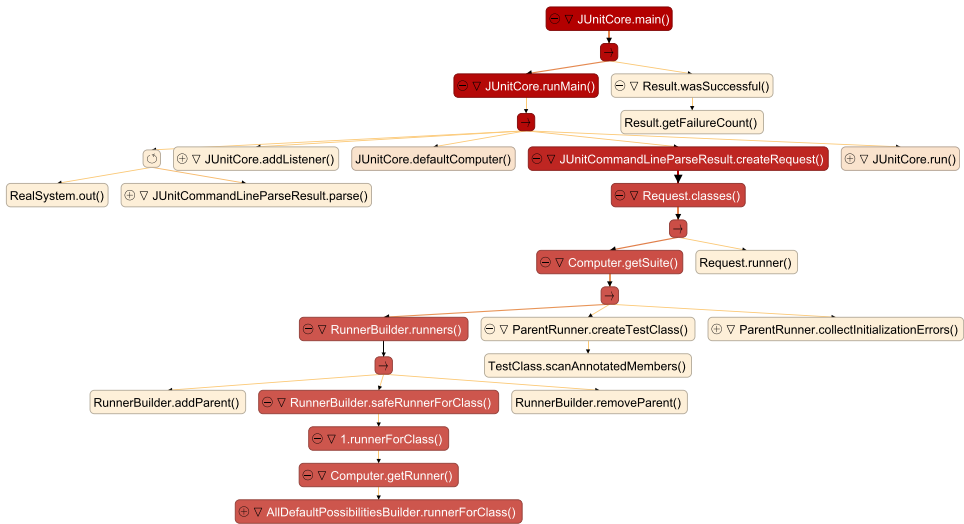
#### 12.2.4 Detailed Exploration and Design Patterns

Below, we will discuss two detailed explorations performed on the JUnit event log, and relate the obtained findings to well-known design patterns.

##### **Test Result Reporting and the Observer Design Pattern**

When running a suite of unit tests using JUnit, we observe that, instead of producing a large result report at termination, test results are being reported as soon as unit tests are finished. Based on these observations and the global process depicted in Figure 12.3, we hypothesize that these intermediate test result reports are generated during `JUnitCore.run()`. In addition, the `JUnitCore.addListener()` in the global process model hints that the Observer design pattern [68] is implemented. In the Observer design pattern, a subject or notifier sends state change notification to all registered listeners or observers. Hence, this pattern is a likely candidate for reporting intermediate test result reports. A quick scan of the activities in the event log indeed reveals various `Notifier` and `Listener` classes, confirming the presence of the Observer design pattern.

In order to better investigate the Observer pattern in isolation, we used a projection of the event log, focussing on the listeners/observers and notifiers/subjects part of the Observer pattern. We used the *Filter Log by Event Attributes* ProM plugin to retain only the events dealing with a `*Notifier` or



**Figure 12.5:** The Statechart Workbench process tree model for the JUnit log, annotated with the duration metric. The *Software Log* preset and default parameter settings are used. The activity labels depicted have been shortened by removing the package names. The duration metric is computed using alignments.

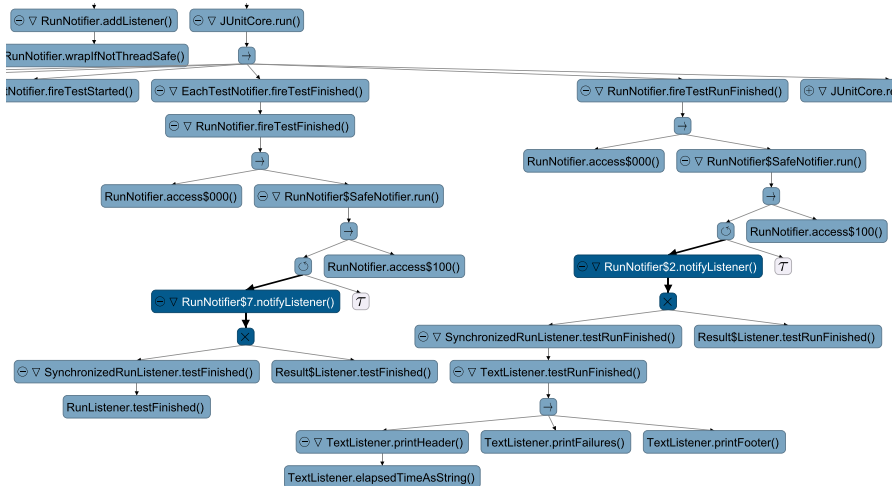
\*Listener class (where \* is a wildcard) and we retained the `JUnitCore` methods `addListener()`, `removeListener()`, and `run()` to provide context. Using this filtered event log, we discovered the hierarchical model shown in Figure 12.6.

In the high-level Message Sequence Diagram (Figure 12.6a), we can clearly see how the Observer design pattern is used to signal unit test progress. First, listeners are registered via `addListener()` and `addFirstListener()`. Next, various updates are propagated via the `*Notifier.fire*`() methods. Finally, a cleanup is performed via `removeListener()`. The above process is a textbook example of the Observer design pattern in action. Zooming in, the `*Notifier.fire*`() notifier updates nicely follow a unit test execution pattern: a test run is started, then tests are started and finished, and after that a test run is finished.

By expanding the various notifier update submodels, we gain more insight in what happens in the various listeners/observers. Figure 12.6b shows the detailed process tree corresponding to the `EachTestNotifier.fireTestFinished()` and `RunNotifier.fireTestRunFinished()` notifications. The first thing we notice is the `RunNotifier` and `SafeNotifier` wrappings, with the latter method calling `notifyListener()` in a loop, probably for all the registered listeners. Based on this part of the control flow, and by quickly inspecting the corresponding source code using the ProM-Eclipse link, we conclude that these wrappings general-



(a) High-level Message Sequence Diagram, providing a global overview.



(b) Part of a more detailed process tree, showing the TextListener in context.

**Figure 12.6:** Part of the projected JUnit model, focussing on the Observer design pattern. The underlying event log is filtered on the listeners and notifiers part of the Observer pattern. The *Software Log* preset and default parameter settings are used, together with an absolute frequency overlay.

ize the Observer pattern, allowing code reuse for the various `*Notifier.fire*()` notifier updates. Going down in the hierarchy, we see the `TextListener` printing unit test failure results during `RunNotifier.fireTestRunFinished()`. Hence, indeed the Observer design pattern is used for printing intermediate unit test result reports as soon as the unit test runs are finished.

### Unit Test Hierarchy and the Composite Design Pattern

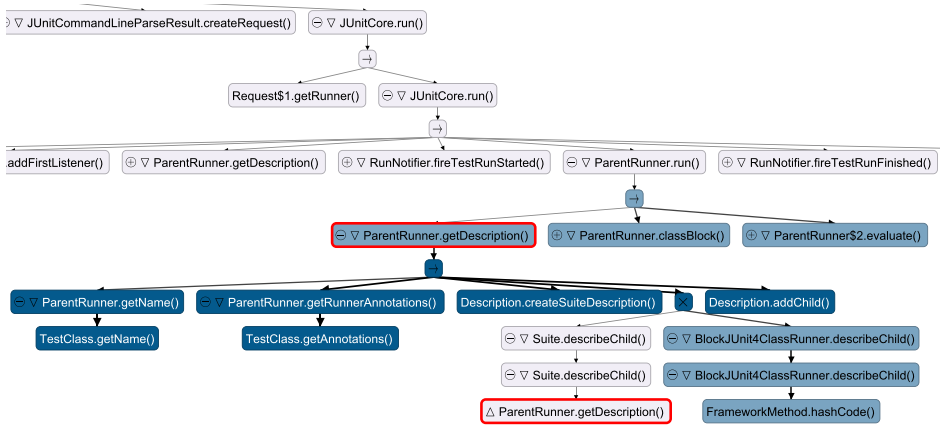
Going back to the original, unfiltered event log, we explored the model from Figure 12.3 in more detail. We observed that during `JUnitCore.run()` and in between the methods `EachTestNotifier.fireTestRunStarted()` and `EachTestNotifier.fireTestRunFinished()`, the method `ParentRunner.run()` is used to actually execute unit tests. While further exploring the process during `ParentRunner.run()`, we noticed that `ParentRunner.getDescription()` calls itself.

Due to the above observations, we switched to the Recursion Aware Discovery algorithm and obtained the model in Figure 12.7. In Figure 12.7a, we see the context and subprocess definition of `ParentRunner.run()`. Notice how the method `ParentRunner.getDescription()` refers to itself in the corresponding subtree, confirming a recursive control-flow relation. Figure 12.7b, shows this recursive behavior in more detail, with the dashed arrow making the recursive control-flow relation explicit. Observe that we see two branches in an XOR split configuration after the method `Description.createSuiteDescription()`: the left branch uses the `BlockJUnit4ClassRunner` class, while the right branch uses the `suite` class. The left branch represents a recursion base case, while the right branch recurses on the `ParentRunner.getDescription()` method. A quick source code inspection on the `BlockJUnit4ClassRunner` and `suite` model elements using the ProM-Eclipse link reveals that both classes are subclasses of `ParentRunner`.

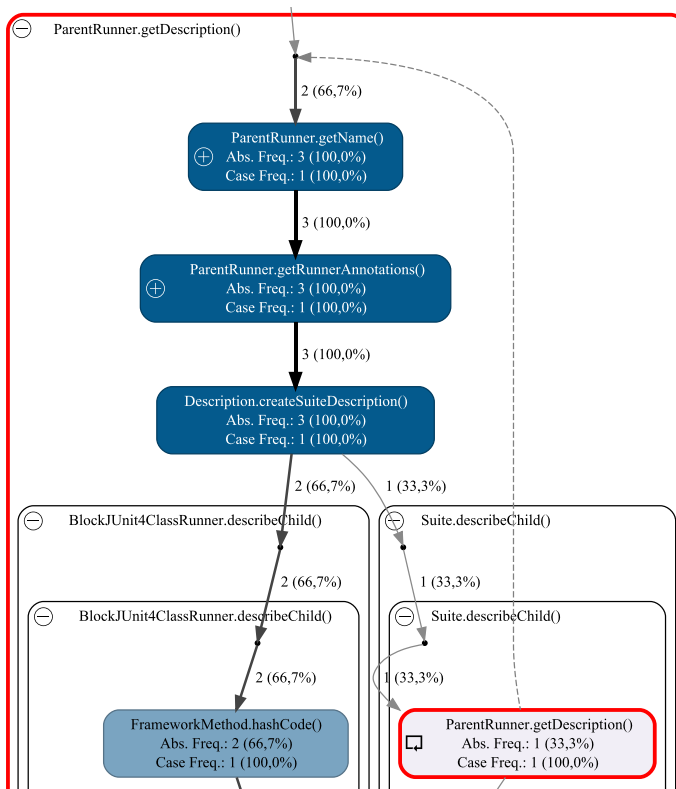
Recall from Section 12.2.1 that test methods are organized into test classes, and several test classes can be combined into test suites. In the above-mentioned `ParentRunner` subclasses, we see this unit test organization structure reflected as well. In fact, based on the observed recursive control-flow and `ParentRunner` class name, we can conclude that there is a tree structure of these so-called unit test runners. Hence, we are dealing with an implementation of the Composite design pattern [68]: the `ParentRunner` and its subclasses compose unit test runners into a tree structure. This tree structure represents a part-whole hierarchy and allowing other code to treat individual test runners and compositions of runners/test suites uniformly. The discovered recursive behavior nicely captures this recursive design pattern in action.

#### 12.2.5 Threats to Validity

At the end of Section 12.2.3, we already discussed how the used instrumentation and tracing overhead threaten the validity of the performance analysis and how we can mitigate this threat. Another threat to validity is that the used example



(a) Part of the high-level process tree, providing a global overview.



(b) Part of the Statechart model, showing the recursion control-flow explicitly.

**Figure 12.7:** Part of the JUnit model, focussing on recursive behavior. The unfiltered event log is used together with the Recursion Aware Discovery algorithm. The red bordered `ParentRunner.getDescription()` methods model the recursive control-flow behavior resulting from the Composite design pattern.

input might be too simple and too small, i.e., it might not trigger all interesting behavior. However, in the above analysis, we already could identify various structured loops and recursion patterns used for processing the input, thus (partly) mitigating this thread. We do acknowledge that a larger follow-up study is needed to confirm this.

## 12.3 Industrial Software Case Study – The Wafer Handling Process

In the previous case study, we investigated the JUnit 4.12 library in a controlled setup, showing how the techniques, tools, and methodology proposed in this thesis can be used in practice on open source software. In the next case study, we will look at the wafer handling process inside ASML's lithography machines, an industrial case study driven by the author in collaboration with a team of ASML software engineers. We will show how we can deal with more complex (industrial) behavior in less controlled setups. Below, Section 12.3.1 gives an introduction to the case study. Section 12.3.2 describes how the initial data extraction and understanding was performed. Section 12.3.3 presents the basic process discovery and evaluation results. After that, Section 12.3.4 presents a more detailed follow-up process analysis. Section 12.3.5 presents some follow up explorations, broadening the scope and suggesting possible future case study efforts. Finally, Section 12.3.6 briefly discusses the threats to validity.

### 12.3.1 Case Description

Since 2015, we performed a multitude of process mining case studies at ASML, a large high-tech company. In the case study discussed below, we analyzed the control-flow of the *wafer handling process* in ASML's *lithography machines*. This case study is just a representative example: it covers complex software behavior with multiple abstraction levels and perspectives, contains normal and error handling behavior as well as asynchronous behavior.

Lithography machines are used in the production process of integrated circuits or chips. These machines process silicon *wafers* (the product) by exposing patterns on them using a mask and a light source with nano-scale precision at very high throughput. Figure 12.8 provides a schematic view of a typical lithography machine. Wafers are processed in batches, called *lots*, and flow through multiple physical positions inside the machine as they are processed. In a typical wafer processing lifecycle, wafers enter via the *wafer handler*, go through the *wafer stage* where they are exposed, and exit via the *wafer handler* again, see Figure 12.8. This *wafer handler* system implements the *wafer handling process*, the focus of this case study.

The wafer handler system is a large, complex piece of software, which has evolved over time as new requirements, constraints, and hardware became available [194]. For this system, existing documentation was written for differ-



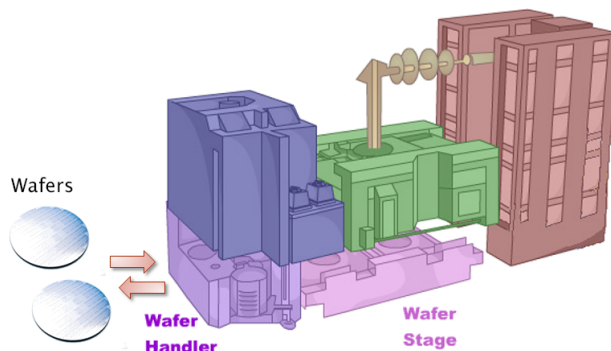
ent versions (older and newer) of different machine types (product variants A, B, and C) from different perspectives (interface provider or user). The documentation is written using natural language (English), which sometimes leaves room for multiple interpretations. The text is supported by (UML) diagrams, for which the (dynamic) semantics is not always precise enough either. As a result, sufficient, up-to-date, and detailed knowledge about the actual implemented behavior for specific versions and configurations of this ever-evolving system is not readily available. Hence, we are dealing with *legacy software*. Therefore, we needed to investigate the implemented software and observe and analyze the actual software behavior. Hence, our main research question is: *what is the actual wafer handling process currently implemented and how does it compare against documented behavior?*

### 12.3.2 Initial Data Extraction and Understanding

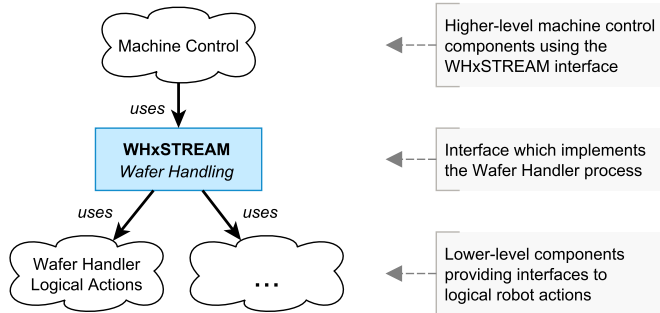
In contrast to the JUnit case study presented before, obtaining and understanding the software event data in this industrial case study proved much more challenging. Below, we discuss the first two phases of understanding we went through while analyzing the implemented wafer handling process.

#### System Scoping and Data Extraction

As stated above, we investigated the wafer handling process. Due to the size and complexity of the complete software system, and the focus of this case study, we cannot simply log all observable behavior. Hence, we needed to investigate the software architecture and select software interfaces specifically dealing with this wafer handling process. At the start of the case study, we



**Figure 12.8:** Schematic view of the lithography machine in the ASML case study, illustrating the position of the wafer handling and wafer stage. Wafers (the product) enter via the wafer handler, go through the wafer stage, and exit via the wafer handler again. Figure courtesy of ASML.

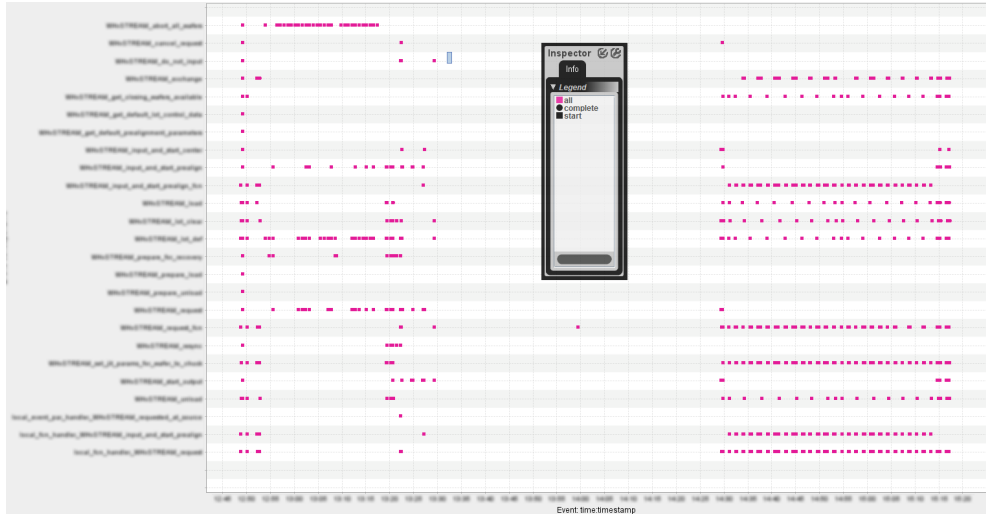


**Figure 12.9:** The analyzed interface in the ASML case study in the context of the larger software system. The WHxSTREAM interface is used by the higher-level machine control components and trigger hardware instructions via the so-called logical actions.

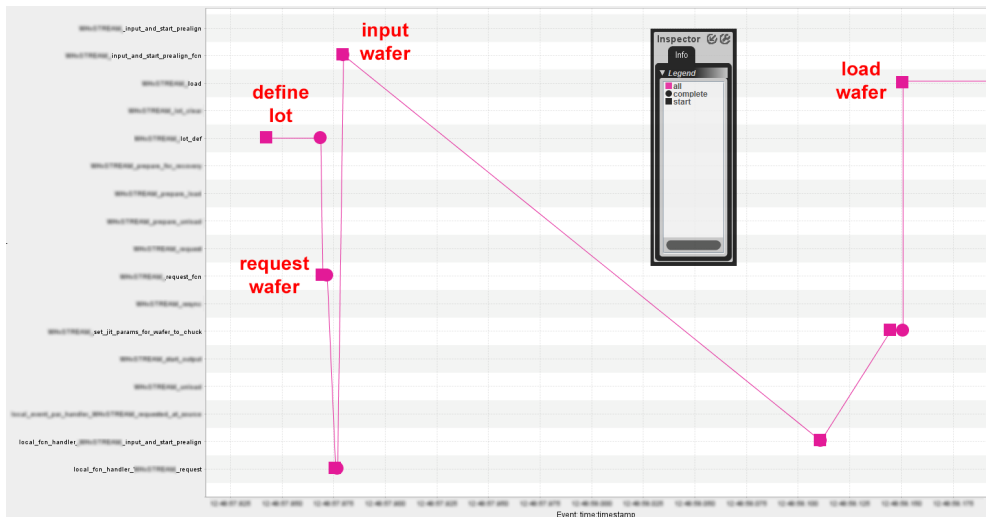
collaborated with a team of software engineers at ASML to select appropriate interfaces (i.e., a system scoping) and determine how to observe and log the software behavior (i.e., a data retrieval strategy). In collaboration, we selected the *WHxSTREAM* software interface (*WH* stands for Wafer Handler); Figure 12.9 illustrates this interface in context. We leveraged the existing tracing/logging infrastructure at ASML to obtain interface-level events detailing the call and return of interface functions. This existing infrastructure was originally developed in-house for, amongst others, debugging purposes, and the resulting logs were not used in formal model-based process analysis. Due to time and cost constraints, we could not observe the software in an industrial production environment: it was too expensive to run the software on the actual lithography hardware. In collaboration, we chose to use a well-established simulation environment used internally for software development and debugging. In addition, we chose a software configuration setup used for a frequently deployed release (a so-called NXT machine type). We used scenario-based regression test suite to trigger typical wafer batch processing behavior.

### First Iteration – Global Dataset Inspection

After executing the above setup, we obtained raw interface function call event data in a proprietary format, containing 2,044 event lines describing 25 different interface functions/activities. After a quick investigation of the structuring and format of event data, we wrote a custom parsing and conversion script to allow inspection of the event data in ProM. Note that, at this point, we only aimed to get a feeling for the overall event dataset and to perform a quick sanity check, as suggested by the methodology from Chapter 11. Thus, at this point, we did not apply any data preprocessing or interpretation. Moreover, no case notion was given: all events are simply grouped into one trace.



(a) Complete overview of the event data.



(b) Zoomed in view on the left part.

**Figure 12.10:** The time-based Dotted Chart [172] view of the ASML event data. Each dot in this scatterplot is an event, with the timestamps of events on the x-axis and activities on the y-axis. The shapes correspond to the lifecycle-transitions, and the lines indicate the directly-follows relation.

We started by inspecting the data in the ProM *Log Visualizer* by looking at various events in the dataset. We note that for each interface function call, a start and a complete was recorded, as well as a `result` attribute. In case the result attribute recorded a negative value, a second function call was recorded afterwards. In addition, various function call events recorded lot and wafer id parameters under various names (e.g. `wafer_number`, `input_wafer_id`, etc.).

In order to get an overall feeling of the recorded behavior, we performed a preliminary Dotted Chart analysis. We projected the method activity names onto the y-axis, and used the default timestamp on the x-axis. We used shapes for the method call/start and return/complete. Figure 12.10 shows the resulting Dotted Chart scatterplot. In the overview from Figure 12.10a, we immediately see a lot of repeating patterns. These patterns, together with the parameter observations discussed before, suggest that there are multiple instances recorded one after another. After zooming in, Figure 12.10b shows us various details of the repeating patterns. Based on these observations, we infer that an example run starts with a call to the *define lot* interface function to initialize a batch of wafers, and ends with a call to *clear lot* to finish a batch of wafers. In between there are several function calls to handle the individual wafers in a batch, such as *request wafer*. We captured the above observations in so-called windowing rules, grouping events and creating traces by indicating the start and end activities of an example run of the software interface usage. Recall that in Section 5.1.3 on page 94 we already discussed how such *windowing* or *protocol run* rules can be used for case identification.

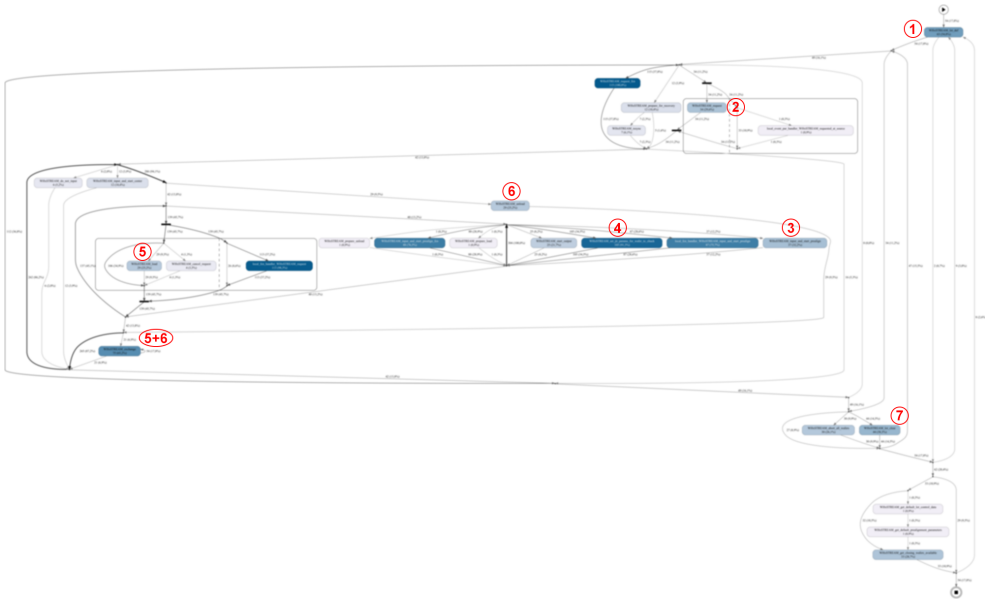
### 12.3.3 Basic Process Understanding

With the above observations, we progressed into getting a basic understanding of the wafer handling process. Below, we report the preliminary processes we discovered, and briefly discuss the accompanied result evaluation.

#### Second Iteration – Preliminary Process Discovery

Using the above observations, we adapted the conversion script to clean up the event data (e.g., taking into account the `result` value), and create cases/traces using the windowing rules. In addition, we used the standard XES extensions to identify key attributes such as activity names, timestamps, and lifecycle transitions (i.e., start/complete or call/return). After reapplying the improved conversion script, we obtained an event log consisting of 54 cases, 1,906 events, and 25 activities describing interface functions.

We loaded the revised event log with windowed events into our Statechart Workbench tool and used the *Normal Log* preset. Note that, based on the previous log inspections and dotted chart analysis, we did not find nested call behavior: we observed only one “slice” of the software stack, the WHxSTREAM interface. When setting the path slider at 100% (no infrequent behavior filter),

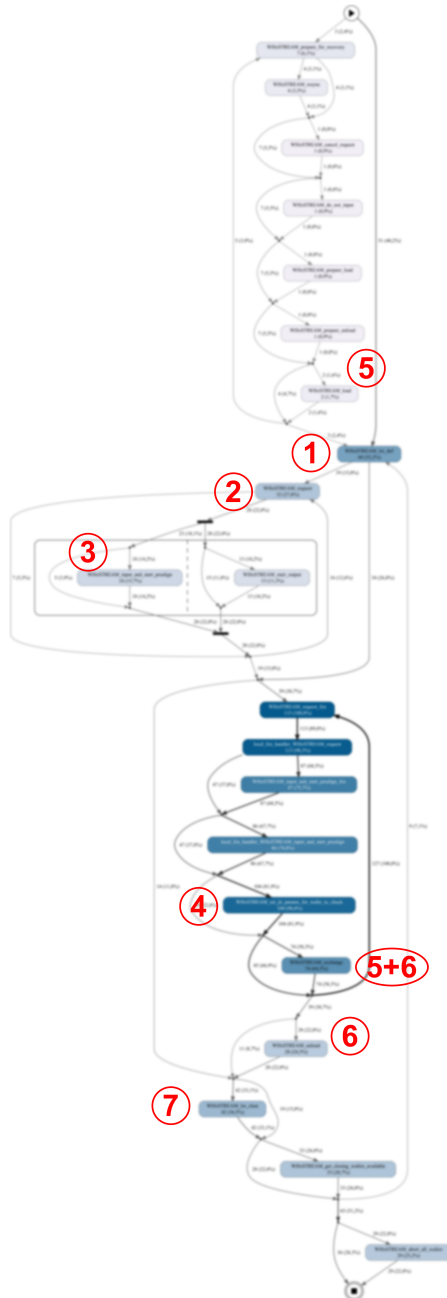


**Figure 12.11:** The initial model for the ASML log obtained via the Statechart Workbench. The *Normal Log* preset is used with the path slider at 100%. **Legend:** 1) define lot, 2) request, 3) input and start prealign, 4) set jit parameters, 5) load, 6) unload, 7) clear lot, 5+6) exchange.

we obtained the model shown in Figure 12.11. Clearly, this model shows large self-loops, indicative of non-block-structured behavior.<sup>3</sup> This apparent lack of structure in the discovered process model is unexpected. After all, we are dealing with software behavior, which should follow the implicit process in the source code. We do not expect the presence of, for example, noise introduced by deviations commonly found human-originated business logs.

In an attempt to find more structure, we set the path slider at 80% and obtained the model shown in Figure 12.12. This model already shows more structured behavior. There are still more loops and skips in the model than expected, but this 80/20 model does reveal more process structures. We identified a couple of interface function calls which explicitly refer to the processing of a lot (i.e., a batch) or an individual wafer, based on their name and their data parameters. These interface functions, labeled 1 through 7, together describe the main sequence of behavior of the wafer handling process, see also the sequence description on the next pages.

<sup>3</sup> Large self-loops and parallel compositions are often introduced by the Fallback cases from Section 4.2.5 on page 81, and thus indicate non-block-structured or non-fitting behavior.



**Figure 12.12:** The initial model for the ASML log obtained via the Statechart Workbench. The *Normal Log* preset is used with the path slider at 80%.  
**Legend:** 1) define lot, 2) request, 3) input and start prealign, 4) set jit parameters, 5) load, 6) unload, 7) clear lot, 5+6) exchange.

The activities labeled 1 through 7 appear mostly in a nice sequential order and roughly describe the following main process flow:

1. *define lot*, initializing a batch of wafers;
2. *request* a new input wafer from the environment;
3. *input and start prealign*, move a wafer into the wafer handler and perform prealign measurements on the wafer;
4. *set jit parameters*, prepares data structures based on prealign sensor results, readying the wafer for processing;
5. *load* a wafer from the handler onto the stage;
6. *unload* a wafer from the stage back onto the handler, the wafer is automatically outputted when finished;
7. *clear lot*, finishing a batch of wafers.

The loops in the discovered model reveal a loopback from 7 to 1 (lot clear and define), and a smaller inner loop from 6 back to 2 (wafer processing). Further inspection of the discovered structures and the underlying event data reveal that within one case window, multiple lots and wafers are being referenced. In addition, the so-called *exchange* function (labeled 5+6) refers to two wafers at the same time. Furthermore, we see that there are several error-handling function interfaces interleaved with the normal process behavior, such as *abort all wafers*, *prepare for recovery*, and *cancel request*. Based on these observations, we hypothesize that we would get a more structured model if we take these observations and patterns into account.

### Evaluation of the Preprocessing and Preliminary Results

We discussed the above results and observations with software engineers at ASML. In addition, we compared the results with the descriptions found in the various documentation. We noted that we have more interface functions/-activities than were documented. Most of the logged activities are “mappable” to documented interface functions based on name similarities. Using this mapping, the discovered patterns and overall wafer handling sequence (i.e., steps 1 to 7 in Figure 12.12) made sense according to the engineers and documentation. In addition, we noted that engineers and documentation have a hardware-centric view of the studied interface: most interface functions are described in terms of how it affects a wafer on a logical or physical location. Furthermore, from a hardware perspective, the WHxSTREAM interface was designed to process multiple wafers concurrently where the hardware allowed it. As a result, certain interface functions are designed to optimize throughput. For example, the *exchange* function we noted before is an optimized version for *loading* a wafer *x* and *unloading* a wafer *y* at the same time.

Based on the above observations and evaluation, we revised and refined our research questions: *Can we confirm the overall wafer handling sequence, and explain/elaborate on the differences between the observed and documented*

*process? Can we relate the wafer handling sequence to the logical locations view? And can we get a better view on the observed abort behavior?*

#### 12.3.4 Detailed Process Understanding

With the above observations and refined research questions, we proceeded to gain a deeper understanding of the wafer handling process. First, we investigated the observed abort behavior. After that, we further explored the core wafer handling process without taking errors like *abort all wafers* into account.

##### Investigating Abort Behavior and Cancellations

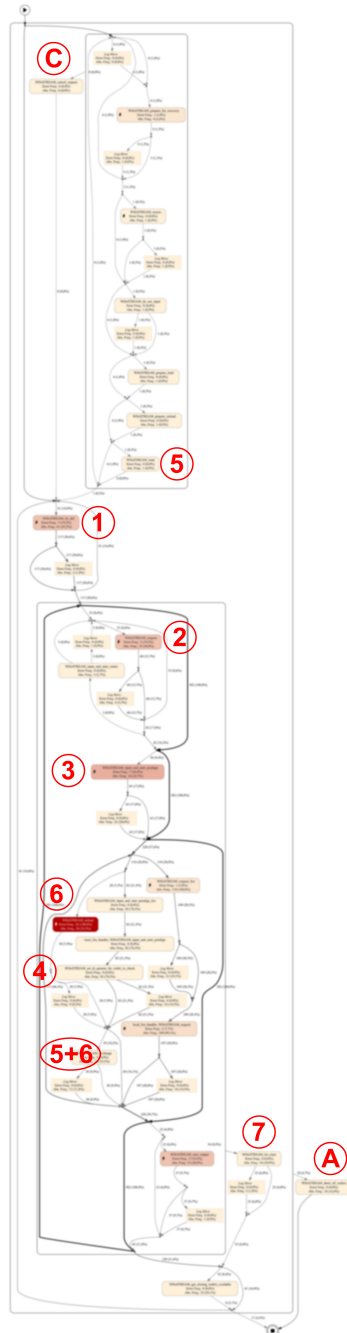
We revisit the preliminary model discovered in our Statechart Workbench tool. We reuse the *Normal Log* preset and 80% path slider, and configured a cancellation trigger oracle for the Cancellation Discovery. Based on the above results, evaluation, and by using a simple activity-name heuristic, we marked the activities *cancel request*, *lot clear*, and *abort all wafers* as cancellation triggers. Figure 12.13 shows the resulting model with three cancellation regions.

In this model, we can clearly see that the entire process is wrapped in a cancellation region associated with *A) abort all wafers*. This shows that this abort interface function is a global effect, which can be triggered at any point in the wafer handling process. Using the *Followed-by Frequency* cancellation metric overlay, we discover that in particular after the 1) *lot define*, 2) *request*, and 3) *input and start prealign* interface functions, an *abort all wafers* is triggered. Apparently, these interface functions resemble critical go-no-go points in the overall wafer handling process. From the existing interface documentation, we learned that during interface functions 1, 2, and 3, wafers are inserted into the machine and measured using sensors. From the discovered model, we learned that following up on interface functions 1, 2, and 3, an abort is possible, likely in response to critical errors or failures in the measurements made during functions 1, 2, and 3.

In the cancellation model, we see a second cancellation region associated with the interface function *C) cancel request*. Using the *Followed-by Frequency* cancellation metric overlay, we observed that this cancellation region is in particular triggered during the *recovery* and *resync* interface function calls. From the existing interface documentation, we learned that via *recovery* and *resync*, one can attempt to reset the hardware and internal process state to a known configuration. From the discovered model, we learned that, as a result, requested wafers (e.g., via function 2) can be canceled during this process.

The third cancellation region is associated with *7) clear lot*. This cancellation region is wrapped only around the core wafer handling process, i.e., steps 2 to 6. We observe that a clear lot is most often triggered after an unload, which corresponds to the normal sequence of wafer handling. However, we also observe that, in some cases, a clear lot is triggered after 2) *request* or *start*





**Figure 12.13:** The cancellation model for the ASML log obtained via the Statechart Workbench. The *Normal Log* preset is used with the path slider at 80%. **Legend:** 1) define lot, 2) request, 3) input and start prealign, 4) set jit parameters, 5) load, 6) unload, 7) clear lot, 5+6) exchange, C) cancel request, A) abort all wafers

*output*. Hence, we learned that after a request, it is not guaranteed that a wafer is loaded into the machine. Instead, we may have reached the end of a batch of wafers and the lot may be cleared.

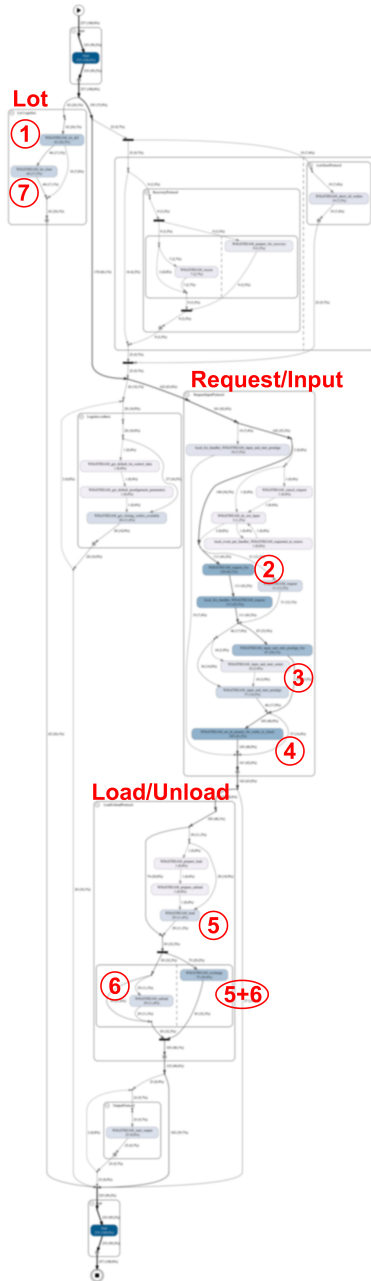
### Third Iteration – Domain Knowledge and Hierarchy

Using the insights obtained thus far, we again adapted the conversion script for interpreting the raw event data. We extended the case identification to use the windowing rules in combination with a normalized wafer and lot id. We preprocessed the *exchange* function events (labeled 5+6) by duplicating these events: one exchange for the loaded wafer and one for the unloaded wafer. In addition, we created a dictionary that maps *interface function* to the *logical locations/protocols* mentioned by engineers and documentation. We enriched the events with a so-called *protocol annotation* based on this dictionary. For example: *define* and *clear lot* are annotated with the *Lot* protocol, *request*, *input* and *start prealign*, and *set jit parameters* are annotated with *Request/Input*, and *load*, *unload*, and *exchange* are annotated with *Load/Unload*. From a logical locations view, the *Request/Input* annotation relates to the input robots manipulating wafers, while the *Load/Unload* relates to the robots putting wafers on wafer stages. See also the schematic view in Figure 12.8.

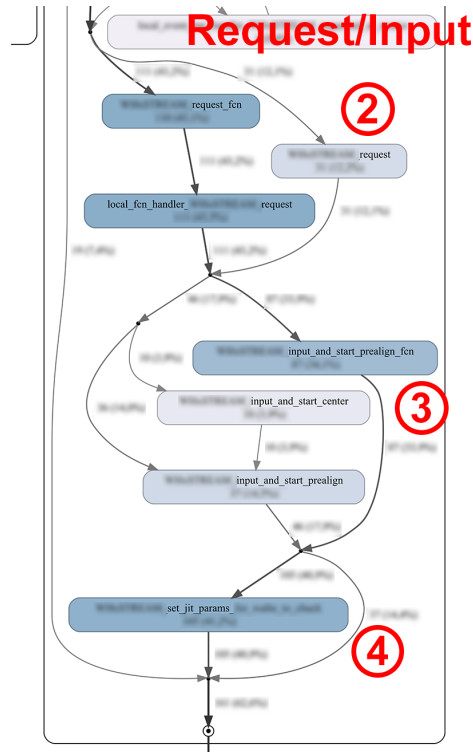
We loaded the above, revised event log with protocol annotations into our Statechart Workbench tool. For the hierarchical discovery algorithm, we constructed a two-level hierarchy based on the custom protocol annotations and interface function name attributes. With the path slider at 80% we obtained the model shown in Figure 12.14. This model has two levels: low-level interface functions are grouped into named submodels labeled with protocol annotations.

In the discovered hierarchical model, we immediately observe that the wafer handling process, excluding *Lot*, is very well structured: interface functions 2 to 6 all happen in sequential order. In addition, we observe that the wafer and lot subprocesses are separated, an artifact introduced by the use of normalized wafer and lot id in case identification: interface function 1 and 7 define no wafer id, while function 2 to 6 do. The artificial hierarchy based on protocol annotations has various benefits. For one, the hierarchy provides more structure to the discovered model based on this protocol domain knowledge. In addition, the annotations made it easier to relate the discovered model to the documented concepts and explain it to the engineers.

When zooming in on various parts of the model, we could better explain the mismatch in the number of observed and documented interface functions/activities we noted before. Figure 12.15 shows such a zoom-in on interface functions 2) *request*, 3) *input and start prealign*, and 4) *set jit parameters*. For example, for the *request* interface function, we note that three functions/activities have been modeled with an XOR choice: either a client executes the plain *request* function, or a client executes the *request\_fcn* function, followed by



**Figure 12.14:** The hierarchical model for the ASML log obtained via the Statechart Workbench. The *Normal Log* preset is used with the path slider at 80%. A hierarchy was constructed based on the custom protocol annotations and interface function names. **Legend:** 1) define lot, 2) request, 3) input and start prealign, 4) set jit parameters, 5) load, 6) unload, 7) clear lot, 5+6) exchange



**Figure 12.15:** Zoomed in version of the hierarchical statechart model in Figure 12.14. Shown are the choice constructs between the various implementations of the interface functions 2) `request` and 3) `input and start prealign`.

an execution of the `local_fcn_handler_request`. The `_fcn` is a domain-specific postfix to denote an interface implementation variant where a so-called *function completion notification* is used. That is, instead of a blocking call to the `request` function, which only returns once the corresponding actions are completed, a `_fcn` call returns directly, and triggers a callback (`local_fcn_handler_request` in this case) once the corresponding actions are completed. Hence, the `request` interface function is modeled with three activities; a developer can choose which implementation to use, possibly allowing for performance optimizations.

For the `input and start prealign` interface function, we observe three interface implementation variants in a XOR choice configuration: either a client executes the normal `input_and_start_prealign`, or a client executes the function completion notification variant `input_and_start_prealign_fcn`, or a client executes the variant `input_and_start_center`. We observe that, in this part of the process, the observed software process did not block and wait for the `local_fcn_handler_input_and_start_prealign` callback to synchronize. From

the documentation, we learn that for some input wafer types, the complete prealign actions are not necessary, and only a center is needed, hence the *input\_and\_start\_center* variant.

In this model, we see that sometimes in the implementation, developers have introduced non-blocking and alternative implementation interface function variants. These non-blocking variants allow a client to queue up action requests at lower levels in the software, without blocking the rest of the process. These alternative implementation variants differ from the documented process and reflect the evolution of the software over time, introducing new functionality while maintaining backward compatibility for the original interface signatures. The artificial hierarchy aids in putting these alternative interface function variants into context, and relate them to known domain concepts. In addition, via this artificial hierarchy, we incorporated the protocol annotations in the control-flow context, relating the observed wafer handling sequence to the logical locations view.

### 12.3.5 Follow-up Process Explorations

In the above case study discussion, we have looked at the *WHxSTREAM* software interface from the perspective of the so-called NXT machine type with a scenario-based regression test suite. We discussed the above results and observations with the software engineers at ASML, and performed various follow-up investigations. In [124], we detailed how we extended the above analysis with additional event data from different machine type configurations and different execution setups (other test suites, so-called load testers, etc.). By looking at additional event data, we obtained a more complete overview of the interface function variants, detailing the commonalities and differences between the documented and implemented interfaces. By investigating and comparing event data from different machine types, we gained insight into the commonalities and differences between machine types from a software interface perspective. And by using conformance checking and model repair techniques, we were able to reconstruct a model incorporating the one-to-many relation between lots (batches) and wafers, showing a more complete overview of the complex software behavior. In a related case study, we observed multiple software interfaces across different architectural layers. By combining such event data with the nested calls hierarchy heuristics, we could extract used-by and runtime dependency relations amongst interfaces.

During these case studies, we also investigated various performance aspects. However, software engineers found such insights unreliable when obtained from the used simulation environment. Hence, for future work, one should develop a plan for extracting performance data from either the industrial production environment itself, or a platform that better mimics such a setup.

### 12.3.6 Threats to Validity

There exist two major validity threats with the above case study: 1) the used simulation setup, and 2) the used test suites for triggering behavior.

Using a simulation setup instead of analyzing actual machine/production behavior threatens the validity of our case study. For a more accurate performance analysis, we agree that a simulation setup would be insufficient. However, for the purpose of the control-flow analysis presented above, we argue the used simulation setup is sufficient. The main argument for this claim is the fact that most developers at ASML do not have access to an industrial production environment either. Hence, the used simulation environment is well-established and is actively being used for development and (integration) testing at the company.

Secondly, the use of test suites for triggering behavior also threatens the validity of our case study. In this case, we noted that the test suites include scenarios checking typical batch wafer processing, as was discussed in Section 12.3.2. Furthermore, though the use of various filter and discovery techniques, we were able to separate the main behavior from erroneous behavior paths, see for example the *abort behavior investigation* in Section 12.3.4. Hence, we argue that this threat, for the purpose of this case study analysis, is sufficiently mitigated.

## 12.4 Conclusion

In this chapter, we discussed two software process analysis case studies. We showed how one can obtain event data and analyze software processes in both an open-source and an industrial setting.

In the JUnit case study, we have shown how the hierarchical discovery and hierarchical performance analysis can be used to investigate behavior and performance patterns across layers of method calls. In addition, we briefly looked at how our techniques and tools can be used to investigate software design patterns and how generating new event data from software is needed for obtaining reliable analysis results for different perspectives.

In the ASML case study, we have shown how the hierarchical and cancellation discovery can be used in combination with domain knowledge to obtain various detailed insights and relate these insights back to domain concepts. We showed how one can make sense of complex software event data, and how our proposed methodology and case identification techniques can aid in this task. In addition, we also noted that generating new event data from software is needed to investigate different perspectives and software configurations.

These case studies have shown how our techniques and solutions can be used in practice, some of the challenges we encountered, and how we addressed those challenges.



# V | Closure

<b>13</b>	<b>Conclusion</b> .....	<b>353</b>
13.1	Contributions	
13.2	Limitations	
13.3	Future Work	
13.4	Broader Outlook on Process Mining and Software Engineering	
<b>A</b>	<b>Proofs</b> .....	<b>365</b>
A.1	Proof for Chapter 4 – A Process Discovery Foundation	
A.2	Proof for Section 6.4 – Naïve Hierarchical Discovery	
A.3	Proof for Section 6.5 – Recursion Aware Discovery	
A.4	Proof for Chapter 7 – Cancellation Discovery	



---

## I Introduction

Chapter 1  
Overview

Chapter 2  
Preliminaries

Chapter 3  
Related Work

Chapter 4  
A Process Mining  
Foundation

---

## II Hierarchical Process Discovery

Chapter 5  
On Software Data  
and Behavior

Chapter 6  
Hierarchical and Recursion  
Aware Discovery

Chapter 7  
Cancellation Discovery

---

## III Beyond Model Discovery

Chapter 8  
Hierarchical Performance  
Analysis

Chapter 9  
Translations  
and Tracability

---

## IV Applications

Chapter 10  
Tool Implementations

Chapter 11  
The Software Process  
Analysis Methodology

Chapter 12  
Case Studies

---

## V Closure

Chapter 13  
Conclusion

Appendix A  
Proofs

In Part IV, we conclude this thesis.

**Chapter 13** summarizes the main results and discusses possible research directions that build on the presented work.

**Appendix A** presents the proof details for the various theorems and lemmas found throughout this thesis.

*“It’s the questions we can’t answer that teach us the most. They teach us how to think. If you give a man an answer, all he gains is a little fact. But give him a question and he’ll look for his own answers.”*

— Patrick Rothfuss, *The Wise Man’s Fear*

## 13 | Conclusion

In this thesis, we have presented techniques and tools based on process mining for analyzing software behavior. At the core of this thesis, we presented hierarchical, recursion aware, and cancelation discovery techniques based on extensions of the *process tree* notation and Inductive Miner framework. In addition, we reused and built upon the mature *alignments* technique [21] to provide a complementary hierarchical performance analysis. All the proposed techniques have been implemented and integrated in the *Statechart Workbench*, which is part of the *Statechart* plugin for the process mining framework ProM [187]. Furthermore, all techniques have been evaluated with respect to performance, applicability and with experiments and case studies using business event logs, open-source software event logs, and industrial software event logs.

In this chapter, conclude this thesis. Section 13.1 revisits and summarizes the contributions in this thesis. In Section 13.2, we discuss the limitations of the presented work. Section 13.3 recaps the suggested future work directions presented throughout this thesis. Finally, Section 13.4 concludes this thesis by reflecting on the broader context and outlook on process mining and software engineering.

### 13.1 Contributions

In Part I, we provided an introduction to process mining and software engineering, positioned our work and presented a basic process mining foundation upon which the techniques in the rest of the thesis build. As stated in Section 1.3, we addressed two types of challenges arising with the application of process mining for analyzing actual software system behavior. On the one hand, there is the lack of support for the type of behavior present in software system settings. On the other hand, there is a lack of support for integrating process mining in the software process analysis lifecycle (see also Figure 1.1 on page 4). The thesis contributions, as listed in Section 1.5, address these challenges on both a technical note and in a broader, more practical context.

In the remainder of this section, we revisit the contributions made in the three main parts of this thesis and reflect on these contributions.

### 13.1.1 Part II – Hierarchical Process Discovery

In Part II, we presented our novel hierarchical process discovery techniques. These hierarchical techniques were inspired by the challenges and properties of software event log data. For these techniques, we build on an existing process discovery foundation to discover *process trees*, and extend the modeling notation and discovery technique where needed. We selected the well-known *Inductive Miner (IM) framework* as it offers good discovery guarantees, scales well, and provides clear extension points for our adaptations. To recap, Part II made the following three contributions.

**Discussion of Software Data and Behavior** – Software behavior and event data differ from business process event logs in several ways. Chapter 5 presented a detailed discussion comparing business and software logs on both log size characteristics and type of behavior (Contribution 1). We concluded that, unlike business event logs, software event logs tend to have more unique activities and longer traces, which can potentially be problematic for process mining techniques. However, we also observed that data which becomes available with observing and logging software executions is “richer” than usual. Using these data in smart ways can provide various interesting process mining solutions, such as the hierarchical techniques presented in this thesis. In addition, in Chapter 5, we discussed the various sources of software event data, how to interpret such data, and how to structure such data.

**Hierarchical and Recursion Aware Discovery** – In Chapter 6, we introduced a modeling notation and two discovery techniques for hierarchical and recursive behavior (Contribution 2). With *hierarchical event logs*, we explicitly captured hierarchical behavior found in, for example, software call stack behavior, relations between software components or application interfaces, behavior described by high-level and low-level activities, etc. We presented several heuristics for transforming an ordinary event log into a hierarchical event log. With the *hierarchical process tree* we provided extensions to capture named submodels and recursive behavior. The *naïve hierarchical discovery* and *recursion aware discovery* algorithms discover named submodels and recursive references from hierarchical event logs and capture this behavior in hierarchical process trees. These discovery algorithms allow us to analyze software processes and other processes at multiple levels of granularity while offering good discovery guarantees. Moreover, these algorithms scale well and show a huge potential to speed up discovery by leveraging hierarchical information.

**Cancelation Discovery** – In Chapter 7, we introduced a modeling notation and discovery technique for cancelation behavior (Contribution 3). With the *cancelation process tree*, we provided extensions to capture sequential and loop-back based cancelation regions. With a *trigger oracle* we made the start of cancelation behavior via so-called trigger activities explicit in the input.

The *cancelation discovery* algorithm discovers multiple, possibly nested cancelation regions from event logs using a trigger oracle and captures this behavior in cancelation process trees. This discovery algorithm allows us to analyze software processes and other processes containing cancelation behavior such as exceptions and error handling. Moreover, the proposed algorithm offers good discovery guarantees and scales well. In addition, we showed how these cancelation discovery solutions can be combined with the previously discussed hierarchical discovery solutions from Chapter 6.

### 13.1.2 Part III – Beyond Model Discovery

In Part III, we further explored hierarchical process mining beyond model discovery. First of all, we extended the notions of hierarchy introduced in this thesis to performance analysis. In addition, we explored how the introduced hierarchical notions can be used across various visualization models to both improve understanding and increase visual scalability. To recap, Part III made the following two main contributions.

**Hierarchical Performance Analysis** – In Chapter 8, we introduced a hierarchical approach to performance analysis, taking into account notions such as subprocesses and cancelation behavior (Contribution 4). Building upon the work on alignments, we introduced a *hierarchical performance analysis framework* for semantic-aware execution subtraces, taking into account model execution semantics. These *execution subtraces* are based on the parts of an aligned log that correspond to a given submodel. Based on this analysis framework, we formalized a selection of existing and novel performance metrics. We evaluated the proposed framework against existing analysis approaches and showed the benefit of the added expressiveness of our hierarchical performance metrics. Moreover, the interaction with hierarchical notions, guided by the hierarchical performance analysis results, proved essential for understanding large, complex (software) behavior. In addition, we showed that the added computational overhead of our hierarchical performance metrics, compared to the end-to-end performance metric computation time, is essentially negligible.

**Model Translations and Traceability** – In Chapter 9, we provided an extensive model translation framework, taking into account the hierarchical, recursive, and cancelation semantics (Contribution 5). In addition, we showed how these translations maintain traceability across models and event logs. The implementation leverages this traceability to close the loop in the software process analysis lifecycle (see Figure 1.1 on page 4).

### 13.1.3 Part IV – Applications

In Part IV, we discussed tools and applications of the techniques and algorithms presented in this thesis. By providing tools, a methodology, and exem-

plary case studies, we helped to address the lack of structure and support for applying software process mining in a round-trip software process analysis. To recap, Part IV made the following contributions.

**Extensive Tool Support for Round-trip Software Analysis** – In Chapter 10, we presented three implemented tools, their interactions, and how our tools allow for integration with existing software artifacts (Contribution 6). With the *instrumentation agent*, we provided a structured way for recording and logging software event data. Independent of the agent, with the *Statechart Workbench* we implemented and integrated all of the techniques proposed in this thesis. This workbench is part of the *Statechart* plugin for the process mining framework ProM [187], and provides export functionality to integrate the results with other ProM plugins and other tools. Finally, with the *Software Analysis Workbench (SAW)* Eclipse plugin, we closed the software process analysis loop. This Eclipse plugin integrates the instrumentation agent and interfaces with the ProM plugin to relate results back to the source code.

**Methodology Support for Software Event Data** – In Chapter 11, we presented a *software process analysis methodology* for obtaining and analyzing software event log data in a structured way (Contribution 1). The presented methodology actively includes the system under analysis and is based on practical experiences in applying process mining on open-source software and legacy software in industry.

**Software Case Studies** – In Chapter 12, we applied the proposed techniques, tools, and methodologies in both an open-source and an industrial case study. We showed how our techniques and solutions can be used in practice, some of the challenges we encountered, and how we addressed those challenges.

## 13.2 Limitations

In this section, we discuss the limitations of the presented techniques, tools, and methodology. We summarize the most important challenges in each of the three main parts of this thesis. In addition, we provide references to the corresponding future work discussions presented in the various chapters.

### 13.2.1 Part II – Hierarchical Process Discovery

The hierarchical process discovery techniques presented in Part II, although already useful in practice (as demonstrated in Part IV), have some inherent limitations. Below, we summarize the most important limitations.

**Representational Bias Limitations** – Our hierarchical and recursion aware discovery algorithms assume a relatively simple interpretation of hierarchies and named submodels. We assume only one hierarchical dimension where each named submodel describes an independent subprocess with a single entry

and a single exit, without any relation to the world outside the named sub-process. In various situations, discovering such a simple hierarchical model is already very valuable, as we have seen in the various evaluation experiments and case studies. However, in some cases, one may wish to have a richer notion of hierarchical behavior. For example, one may wish to influence the state of another non-nested submodel, possibly via communication mechanisms (see Future Work 6.1). In other cases, one may wish to drop the single entry and single exit restriction inherited from the underlying Inductive Miner. In addition, it can be useful to adopt multiple hierarchical views (see Future Work 6.5).

**Unsupported Software Behavior** – The hierarchical and cancelation support presented in this thesis are a very generic approach to support some of the more common behavioral patterns observable in running software. However, there are various other behavioral patterns which are currently not supported. For one, we completely ignored the notion of objects and object instances (see Future Work 6.2). Current algorithms are limited in inferring interactions amongst objects, and current recursion detection ignores object instances. In addition, multi-threaded behavior lacks discovery and analysis support. Current discovery algorithms have difficulty detecting where behavior happens concurrently across different computational threads and have difficulty discovering the appropriate fork and join constructs (see Future Work 6.3 and Future Work 8.2). Furthermore, we currently cannot adequately discover multi-instance patterns where a function or subroutine is invoked in parallel for each element in a dataset (see Future Work 6.4).

**Oracle Instantiations** – The presented hierarchical and cancelation discovery algorithms rely on external knowledge via heuristics and oracles. In some cases, it is not immediately clear which settings should be used, and selecting the right heuristic can be error-prone. These limitations raise the threshold for applying the proposed techniques. Although part of the heuristic and oracle selection could be automated (e.g., see Future Work 7.1 and Future Work 7.2), the current techniques do not support this.

### 13.2.2 Part III – Beyond Model Discovery

The techniques presented in Part III put the hierarchical ideas in a broader context. By leveraging and adapting existing notations (Petri nets) and algorithms (the alignments algorithm), we automatically inherited some of their limitations.

**Alignment Computation Time** – In our performance techniques, we rely on the alignments algorithm for computing a mapping between model elements and logged events. However, calculating an alignment can be prohibitively expensive in certain cases. Especially when dealing with the large and complex models our hierarchical discovery techniques can produce, the

traditional alignment algorithm can be a limiting factor during an analysis (see Future Work 6.6 and Future Work 8.4).

**Analysing Nonfitting Behavior** – In our performance techniques, we assume a certain degree of fitting behavior: the various performance metrics are mostly defined in terms of synchronous alignment moves (i.e., fitting behavior). This approach is rather binary and limits the applicability of our approach: either we have fitting behavior, and we can compute metrics on it, or we do not (see Future Work 8.6).

### 13.2.3 Part IV – Applications

The methodology and the case studies performed with the tools implementing our techniques revealed several practical issues.

**Layout Limitations** – The introduced discovery techniques allow for the exploration and analysis of larger and more complex process models. Various tool techniques, such as expanding and collapsing named submodels, aid the user in exploring these type of models. However, the graph layout algorithms currently used are not ideal for this type of usage. These graph layout algorithm has a significant computation time and the produced layouts are not robust or deterministic, limiting the usability of our approaches (see Future Work 6.7, Future Work 7.4, and Future Work 8.3).

**Unclear Tradeoffs in Observing Software** – When observing a software system, one has to make various system scoping and data extraction decisions. What are we going to observe? At which granularity are we going to observe? And what will be the effect of the selected observation setup on the accuracy and reliability of the obtained software event data? In the presented case studies, we relied on external knowledge to make the above decisions. However, there is a lack of a structured and informed way to decide what to observe and log, what not, and why. In short, this reliance on external knowledge and expertise is a limiting factor for more novice software analysts. To the best of our knowledge, there are no established theories and guidelines on such tradeoffs in observing software (see Future Work 10.1, Future Work 11.1, and Future Work 11.2).

**Lack of Source Artifact Integration** – During the case studies, we saw the value of relating software process analysis results back to known domain concepts and artifacts. We recognized such a need by providing a proof-of-concept source code traceability in the developed tools. However, the current prototypical realization is limited to a specific source code location formatting and a specific Eclipse plugin for an Eclipse Java project. Currently, there is no support for a broader set of source artifacts integration beyond the (bidirectional) model-to-source-code link (see Future Work 8.5, Future Work 9.3, and Future Work 10.3).

## 13.3 Future Work

Throughout this thesis and in the above limitation discussions, we have touched upon various future work directions. Below, we summarize the most promising future work ideas for each of the three main parts of this thesis.

### 13.3.1 Part II – Hierarchical Process Discovery

With the foundations in hierarchical and cancelation process discovery presented in this thesis, we foresee several interesting future work directions.

**Extend Notation and Discoverable Behavior** – As already noted in the limitations above, there is still ample of room for extending the notations and discovery capabilities with richer behavior. The goal of such future work is to improve the accuracy and simplicity of the discovered process models as well as presenting end users with familiar and understandable notations (see Future Work 9.2). For one, we can extend upon the current hierarchical notations by including concepts such as *communicating submodels* (see Future Work 6.1), *mixed, multi-dimensional hierarchies* (see Future Work 6.5), and a notion of *submodel history* as introduced in the Statechart notation. In addition, we can include workflow patterns like the *milestone* [15], multi-instance patterns, and various environmental triggers like the BPMN *events* or MSD *external messages* (SEE Future Work 9.1).

**Data-Based Software Behavior Support** – The process discovery algorithms discussed in this thesis discover the modeled behavior mainly based on causality relations, e.g., by looking for patterns in a directly-follows graph. In software event logs, there is often a lot of additional information available, such as object instance identifiers (see Future Work 6.2), execution thread identifiers, and more, see also Section 5.3 on page 102. Not only is using such additional information a nice future work direction, in some cases, it is also necessary to consider such information. For example, when observing multi-threaded software, we often do not observe enough interleaving and causal relations to infer concurrency based on a directly-follows pattern. However, the recorded thread-id data (see Section 5.3.6 on page 106) in such software event logs can be accurate enough to infer the exact concurrency patterns (see Future Work 6.3, Future Work 6.4, and Future Work 8.2).

**(Semi-)Automated Oracle Instantiations** – As already discussed in the limitations above, there are various possible future work directions to automate the oracle instantiations for hierarchical and cancelation discovery. On the one hand, one could use the observations during discovery to automatically tune oracles. For example, special fallback cases could detect cancelation patterns, and thus aid in instantiating a cancelation trigger oracle (see Future Work 7.1). On the other hand, one could use heuristics to select a configuration for an oracle, and validate this configuration via metrics like fitness



and precision (see Future Work 7.2). In addition, one could use heuristics to guide users to likely correct configuration settings for a hierarchy heuristic or cancelation trigger oracle.

### 13.3.2 Part III – Beyond Model Discovery

With the proposed techniques and observations in Part III, we foresee several interesting future work directions for analyzing complex processes.

**Extended Performance Analysis** – With the notion of hierarchical performance analysis, we have shown the advantages of extending performance analysis with notions beyond a simple duration or waiting time. Likewise, we foresee potential in further extending performance analysis. On the one hand, there is a wealth of additional information to include in performance analysis. For example, one could include additional lifecycle information via concepts like queue mining, or include thread-activity information to enable thread state execution analysis (see Future Work 8.1 and Future Work 8.2). On the other hand, one could investigate how performance analysis can be made conformance-aware. We already noted the limitations in analyzing non-fitting behavior. However, one could consider non-fitting behavior as an additional source of information instead of information absence (see Future Work 8.6).

**Improved Model Exploration** – With the larger and more complex models produced by the techniques in this thesis, there is the opportunity for future work to improve model exploration. One possible direction to investigate is how performance metrics could actively aid the user in exploring a model and doing a performance analysis. For example, performance metrics can be used to automatically focus the user view on certain parts of the model, suggesting where to start an analysis (see Future Work 8.7). In addition, filtering on the model level, instead of the event log level, could be investigated in more detail. User interactions or specific algorithms could hide less interesting parts of the discovered models (see Future Work 10.2). Furthermore, future work could investigate how multiple notations or views can be used together in exploring the discovered behavior and performance (see Future Work 10.4).

**Alternative Alignment Computations** – As already discussed in the limitations above, calculating alignments for performance analysis can be prohibitively expensive. This is especially true when dealing with the large and complex models our discovery techniques can produce. However, when applying performance analysis on models discovered by our discovery techniques or similar approaches, there is additional information available. Future work could investigate how the modeled hierarchies and the information obtained during log splitting can be used to speed up alignment computations (see Future Work 6.6). Future work could also look into alternatives to optimal alignments, such as estimation and approximation techniques (see Future Work 8.4).

**Explore Source Integration Opportunities** – As noted several times throughout this thesis, there is a large potential in combining software process mining with available source artifacts such as the source code. The ProM-Eclipse source-code link is a nice example, but many other opportunities remained to be explored. Future work could investigate how the source code traceability can be used to generate additional data on demand, making the software process mining more interactive (see Future Work 6.8). In addition, one could explore how to integrate and use source code and software-quality metrics on discovered process models (see Future Work 8.5). The other way around, future work could also investigate how software process mining can be used in the realm of source artifacts such as source code, DSL models, and class diagrams (see Future Work 9.3 and Future Work 10.3).

### 13.3.3 Part IV – Applications

The methodology and the case studies performed with the tools implementing our techniques revealed an interesting future work direction.

**Process Tree Guided Layout** – As we already noted, the current layout algorithm for graph models like the Statechart limits the useability of our approaches. Not only is the graph layout computation relatively slow, it also is not robust or deterministic: changing a small part of the model (e.g., collapsing a named submodel) or simply redrawing a model can have a large effect on the model layout. The fact that our graph models are backed by the extended process tree notation, and thus a hierarchical structure, could provide interesting future work possibilities for creating fast, robust, and deterministic layout algorithms (see Future Work 6.7, Future Work 7.4, and Future Work 8.3).

**Investigate Logging Tradeoffs** – As noted in the limitation above, there is a lack of a structured and informed way to decide what to observe and log in a software system, what not, and why. First of all, future work should investigate the tradeoffs and scalability of observing software (see Future Work 11.1). At what granularity should software be observed? What impact do such observations have on the quality of the obtained data? And which logging architectures and setups are best suited for the various software observation scenarios (see Future Work 10.1)? Such future work efforts should lay the groundwork, theory, and experiments for how to observe software and the involved tradeoffs. In addition, there should be improved tool support to help users obtain software event data and measure/estimate the quality of the obtained data (see Future Work 11.2).

## 13.4 Broader Outlook on Process Mining and Software Engineering

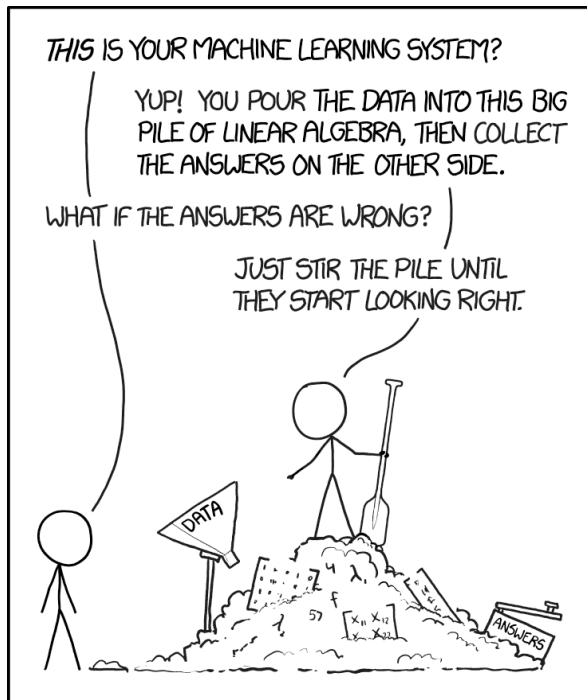
We conclude this thesis with a broader outlook on process mining and software engineering, reflecting on our contributions in a broader context.

As we already acknowledged in Chapter 1, in today's world, we increasingly rely on information technology. Complex software-driven systems can be found in all sectors: communication, production, distribution, healthcare, transportation, education, entertainment, government, trade, etc. [7] Moreover, many complex problems and automation challenges are increasingly being solved by data mining, machine learning, artificial intelligence, data science, and big data technologies. However, when these software-driven systems fail or do not behave as expected, they can have an incredible impact on society, organizations, and users. Although considerable amounts of data are recorded by software, machines, and organizations, such problems are typically only addressed in a trial-and-error and ad-hoc fashion. This is in stark contrast to the recent trend of data-driven machine learning type of solutions for complex problems.

In the field of business intelligence and process mining, we see an increasing trend of relying on data-driven analysis, designing, and optimizations. Business processes are seen as complex, living things, which we must observe and monitor to make evidence-based improvements. This trend is supported by the rise of commercial business intelligence and process mining tools as well as specialized (process mining) consultancy companies. The field of software engineering could borrow some of these practices and insights from the business process domain: complex software-driven systems should be seen as complex, living things, which we must observe and monitor to make evidence-based improvements. After all, software continues to evolve and be used long after the initial code is written.

Already, we see some progress in the direction of evidence-based software engineering. For example, recent developments in artificial intelligence and machine learning focuses on making decision and classification algorithms “explain themselves” based on observable, real-life data [79, 135, 167]. The field of process mining could borrow some ideas from this concept: process mining algorithms should explain why certain models or analysis results are derived and provide a justification, i.e., it is not sufficient to just mine results that “look right” (Figure 13.1).

Software-driven companies more and more realize the impact of their existing software, as well as their lack of understanding of their own systems, i.e., their legacy. We need to move from “developing software once” to “understanding software in the field”, and adopt an evidence-based maintenance and improvement paradigm. Rather than fixing bugs after costly failures in production, we should understand, predict, and prevent such system failures. The question is, how can we move towards an evidence-driven software engineering and maintenance paradigm in a reliable, scalable, and ethical way?



**Figure 13.1:** xkcd: Machine Learning. It is not sufficient to just mine results that “look right”. Source: <https://xkcd.com/1838/>.



“Why do you go away? So that you can come back.  
So that you can see the place you came from with  
new eyes and extra colors. And the people there see  
you differently, too. Coming back to where you  
started is not the same as never leaving.”

— Terry Pratchett, *A Hat Full of Sky*

# A | Proofs

In this appendix, we provide detailed proofs for the various theorems and lemmas found throughout this thesis, ordered by chapter.

## A.1 Proof for Chapter 4 – A Process Discovery Foundation

The Inductive Miner (IM) framework, as introduced in Chapter 4, offers good discovery guarantees, scales well, and provides clear extension points for our adaptations. In this section, we will briefly cover the proofs for the discovery guarantees provided by the basic framework as described in Chapter 4. We refer the reader to [130, 131] for the full proofs.

### A.1.1 Soundness and Termination

**Theorem 4.3.1 — IM guarantees soundness.** All models returned by the IM framework are guaranteed to be sound.

*Sketch of Proof.* The framework returns process trees, which are sound by construction.  $\square$

**Theorem 4.3.2 — IM guarantees termination.** The IM framework is guaranteed to always terminate.

*Sketch of Proof.* Termination follows from two facts: 1) the sublogs get smaller in each recursion, and 2) there are finitely many recursions for each recursion step. By construction of  $\text{SplitLog}(L, (\otimes, (\Sigma_1, \dots, \Sigma_n)))$ , fact 1 holds. By construction of  $\text{FindCut}(L)$ , finitely many partitions or log divisions are created, hence the number of recursions is also finite and fact 2 holds. See [131, Theorem 2, page 7] for the full proof.  $\square$

### A.1.2 Perfect Fitness

**Theorem 4.3.3 — IM guarantees fitness.** The IM framework returns a model that fits the log. That is, given an event log  $L$ , the IM framework returns a

model  $Q$  such that  $L \subseteq \mathcal{L}(Q)$ .

*Sketch of Proof.* The fitness proof is based on induction on the log size. As induction hypothesis, we assume that for all sublogs, the discovery framework returns a model able to reproduce all traces, and then prove that the step maintains this property. That is, for all sublogs  $L_i$  we have a corresponding submodel  $Q_i$  such that  $L_i \subseteq \mathcal{L}(Q_i)$ .

Using the induction hypothesis (IH) and the process tree language-join functions ( $\otimes_{\mathcal{L}}$ , see Definition 2.2.12 on page 40), we can show that the join operator  $\otimes$  in the step maintain the fitness property. That is, since by IH  $\forall i : L_i \subseteq \mathcal{L}(Q_i)$  we can conclude  $\otimes_{\mathcal{L}}(L_1, \dots, L_n) \subseteq \mathcal{L}(\otimes(Q_1, \dots, Q_n))$ . And since  $L \subseteq \otimes_{\mathcal{L}}(L_1, \dots, L_n)$  and  $\otimes(Q_1, \dots, Q_n) = Q$ , we can conclude  $L \subseteq \mathcal{L}(Q)$ .

See [131, Theorem 3, page 7] for the full proof.  $\square$

### A.1.3 Language Rediscoverability

**Lemma 4.3.4 — IM rediscovers base cases.** Let  $Q = a$  for some  $a \in \mathbb{A}$  or let  $Q = \tau$ ; let  $L$  be a log such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ . Then  $\text{IMdiscover}(L) = Q$ .

*Sketch of Proof.* The proof follows from a case distinction on the base cases, and code inspection. See [131, Lemma 12, page 21] for the full proof.  $\square$

**Lemma 4.3.5 — IM selects right tree operator.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the model restrictions in Definition 4.3.2 and let  $L$  be a log such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ . Then  $\text{FindCut}(L)$  selects  $\otimes$ .

*Sketch of Proof.* The proof strategy is to prove for all operators that given a log  $L$  that is directly-follows complete to the complete model,  $\otimes$  will be the only operator for which  $G(L)$  satisfies all the cut criteria. See [131, Lemma 11, page 20] for the full proof.  $\square$

**Lemma 4.3.6 — IM splits logs correctly.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the model restrictions in Definition 4.3.2 and let  $L$  be a log such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ . Let  $\otimes$  be the result of  $\text{FindCut}(L)$  and let  $(L_1, \dots, L_n)$  be the corresponding result of  $\text{SplitLog}$ . Then for the resulting sublogs  $L_i$  we have  $\forall i : L_i \diamond_{df} Q_i \wedge L_i \subseteq \mathcal{L}(Q_i)$ .

*Sketch of Proof.* The proof strategy is to prove for all operators using the following three steps. First, we check that for each operator, the cut detection returns the correct activity division. Next, using that division, we prove that the log split returns sublogs valid for their submodels. Finally, we can show that each resulting sublog produces a log that is directly-follows complete with respect to its submodel. See [131, Lemma 13, page 21] for the full proof.  $\square$

**Theorem 4.3.7 — IM guarantees language rediscoverability.** If the model restrictions in Definition 4.3.2 hold for a process tree  $Q$ , then  $\text{IMdiscover}$  language-rediscovered  $Q$ , i.e.,  $\mathcal{L}(Q) = \mathcal{L}(\text{IMdiscover}(L))$  for any log  $L$  such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ .

*Sketch of Proof.* The proof for this theorem relies on showing that a reduced version  $Q'$  of  $Q$  is isomorphic to the model returned by  $\text{IMdiscover}$  using induction on model sizes. Lemma 4.3.4 proves isomorphism of the base cases. In the induction step, Lemma 4.3.5 ensures that  $\text{IMdiscover}(L)$  has the same root operator as  $Q$ , and Lemma 4.3.6 ensures that the subtrees of  $Q'$  are isomorphically rediscovered as subtrees of  $\text{IMdiscover}(L)$ . See [131, Theorem 14, page 23] for the full proof.  $\square$

#### A.1.4 Polynomial Runtime Complexity

**Theorem 4.3.8 — IM has polynomial runtime complexity.** The runtime complexity of the IM framework is bounded by  $O(|\mathbb{A}(L)|^5 + |\mathbb{A}(L)| \cdot |L|)$ .

*Proof.* As stated on [130, page 198], the runtime complexity of the IM framework depends on the number of traces in the event log  $c = |L|$  and on the size of the activity alphabet  $n = |\mathbb{A}(L)|$ . The cut detection for the normal process tree operators can be defined as common graph problems for which polynomial algorithms exist. The exclusive choice, concurrency and loop cuts can be translated to finding connected components, the sequence cut to finding strongly connected components. There are three recursive paths we need to consider:

- The base cases stop recursion and have a runtime complexity of  $O(c)$ .
- The cut detection involves the construction of a directly-follows graph (runtime complexity  $O(c)$ ), cut detection (runtime complexity  $O(n^3)$  for computing reachability for the sequence cut detection) and log splitting (runtime complexity  $O(c)$ ). In this step, the size of the alphabet is decreased by at least one.
- Most fallback cases have a runtime complexity  $O(c)$ . Fallback 4.3 has runtime complexity  $O(n \cdot n^3) = O(n^4)$  for performing a cut detection for each activity in the alphabet. In this step, the size of the alphabet is decreased by at least one.

Based on these recursive paths, we conclude that the number of recursions made is bounded by the number of activities:  $O(n) = O(|\mathbb{A}(L)|)$ .

We get the following recurrence for the IM runtime complexity  $T(n)$ :

$$T(n) = \begin{cases} c & \text{if a base case was applied } (n = 1) \\ c + n^3 + c + T(n - 1) & \text{if a cut was detected } (n > 1) \\ c + n^4 + T(n - 1) & \text{if a fallback was used } (n > 1) \end{cases}$$



Since this runtime complexity is dominated by the fallback cases, we hypothesize for any non-negative integer  $k$ :

$$T(n) = T(n - k) + k \cdot n^4 + k \cdot c$$

We prove this new recurrence holds by induction on  $n$ :

**Base:** for  $k = 1$

$$T(n) = T(n - 1) + n^4 + c$$

**Step:** using the induction hypothesis on  $k - 1$

$$\begin{aligned} T(n) &= T(n - (k - 1)) + (k - 1) \cdot n^4 + (k - 1) \cdot c \\ &\quad \{Substituting\ original\ fallback\ recurrence\ for\ T(n - (k - 1))\} \\ &= (T(n - k) + n^4 + c) + (k - 1) \cdot n^4 + (k - 1) \cdot c \\ &= T(n - k) + k \cdot n^4 + k \cdot c \end{aligned}$$

Hence, the new recurrence holds. Using  $k = n$ , with  $T(0) = c$ , we get:

$$\begin{aligned} T(n) &= c + n \cdot n^4 + n \cdot c \\ &= n^5 + n \cdot c \end{aligned}$$

Therefore, the runtime complexity of the IM framework is bounded by:

$$\begin{aligned} &O(n^5 + n \cdot c) \\ &= O(|\mathbb{A}(L)|^5 + |\mathbb{A}(L)| \cdot |L|) \end{aligned}$$

Note that this runtime complexity is tighter than the original runtime complexity provided in [130, page 198].  $\square$

## A.2 Proof for Section 6.4 – Naïve Hierarchical Discovery

The *naïve hierarchical discovery* (NHD) algorithm maintains the IM discovery guarantees from Section 4.3 on page 84 and returns a sound hierarchical process tree (Definition 6.3.3). In this section, we will discuss the proofs for these discovery guarantees and properties.

### A.2.1 Soundness and Termination

**Theorem 6.4.1** — **NHD guarantees soundness.** All models  $Q$  returned by the NHD algorithm are guaranteed to be sound.

*Proof.* According to Definition 6.3.3, a hierarchical process tree is not sound if its language contains unresolved recursion markers or a resolved recursion has no option to terminate. Since the NHD algorithm does not support the

recursive reference operator, neither of these cases can occur in a model  $Q$  returned by `NHDiscover`. Hence, all returned hierarchical process trees are sound.  $\square$

**Theorem 6.4.2 — NHD guarantees termination.** The NHD algorithm is guaranteed to always terminate.

*Proof.* Consider Base Case 6.1: *Single Activity – No Hierarchy*. Observe that this base case returns a leaf and does not recurse. Hence, this base case terminates.

Consider Base Case 6.2: *Single Activity – Lower Level in the Hierarchy*. Observe that the log  $L$  has a finite depth, i.e., a finite number of levels in the hierarchy. Note that the sequence projection  $L|_1^*$  yields strictly smaller event logs, i.e., the number of levels in the hierarchy strictly decreases. Therefore, we can conclude that this base case yields only finitely many recursions and thus terminates.

By the base Theorem 4.3.2, we know termination is guaranteed for the original step and fallback cases. Hence, the NHD algorithm is guaranteed to always terminate.  $\square$

## A.2.2 Perfect Fitness

**Theorem 6.4.3 — NHD guarantees fitness.** The NHD algorithm returns a model that fits the log. That is, given an event log  $L$ , the NHD algorithm returns a model  $Q$  such that  $L \subseteq \mathcal{L}(Q)$ .

*Proof.* Based on the proof from the base Theorem 4.3.3, as induction hypothesis (IH), we assume that for all sublogs, the discovery framework returns a fitting model, and then prove that the step maintains this property. That is, for all sublogs  $L_i$  we have by IH a corresponding submodel  $Q_i$  such that  $L_i \subseteq \mathcal{L}(Q_i)$ , and we have to prove  $L \subseteq \mathcal{L}(Q)$ .

Consider Base Case 6.1: *Single Activity – No Hierarchy*. This proof is analogous to the base Theorem 4.3.3 proof for Base Case 4.2.

Consider Base Case 6.2: *Single Activity – Lower Level in the Hierarchy*. That is, we have  $\text{NHDiscover}(L) = \nabla_f(\text{NHDiscover}(L|_1^*)) = Q$ , and we have to prove  $L \subseteq \mathcal{L}(Q)$ . We deduce:

$$\begin{array}{l} \frac{Q_1 = \text{NHDiscover}(L|_1^*)}{L|_1^* \subseteq \mathcal{L}(Q_1)} \text{ Induction Hypothesis on } L|_1^* \text{ and } Q_1 \\ \frac{L \subseteq f.\mathcal{L}(Q_1)}{L \subseteq \mathcal{L}(\nabla_f(Q_1))} \text{ Apply } f.(L|_1^*) = L \\ \text{ By semantics of } \nabla_f: f.\mathcal{L}(Q_1) = \mathcal{L}(\nabla_f(Q_1)) \\ \frac{L \subseteq \mathcal{L}(\nabla_f(\text{NHDiscover}(L|_1^*)))}{L \subseteq \mathcal{L}(Q)} \text{ Apply } Q_1 = \text{NHDiscover}(L|_1^*) \\ \text{ Apply } \nabla_f(\text{NHDiscover}(L|_1^*)) = Q \end{array}$$

Hence, we conclude that for the named subtree operator we return a process model that fits the log  $L$ .

By the base Theorem 4.3.3, we know fitness is guaranteed for the original step and fallback cases. Hence, the NHD algorithm guarantees fitness.  $\square$

### A.2.3 Language Rediscoverability

**Lemma 6.4.4 — NHD selects the right tree operator.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the above model restrictions and let  $L$  be a log such that  $L \diamond_{df} Q$ . Then  $\text{NHDdiscover}(L)$  returns a tree with root  $\otimes$ .

*Proof.* By construction of the base IM framework, base cases are checked before any cut detection or fallback cases. Suppose  $Q = \nabla_f(Q_1)$  for some  $f \in \mathbb{A}$ , then by  $L \diamond_{df} Q$  we know that every event  $e$  in  $L$  starts with  $f$  and there exists an event with a lower level in the hierarchy. Therefore, Base Case 6.2 applies, and  $\text{NHDdiscover}(L)$  returns a tree with root  $\nabla_f$ .

In all other cases, base Lemma 4.3.5 applies. Hence,  $\text{NHDdiscover}(L)$  returns a tree with root  $\otimes$ .  $\square$

**Lemma 6.4.5 — NHD splits logs correctly.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the above model restrictions and let  $L$  be a log such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ . Then for the resulting sublogs  $L_i$  produced by NHD we have  $L_I \diamond_{df} Q_i \wedge L_i \subseteq \mathcal{L}(Q_i)$ .

*Proof.* Consider Base Case 6.2. Then we know that  $Q = \nabla_f(Q_1)$  for some  $f \in \mathbb{A}$  and we know that  $L_1 = L|_1^*$ . We have to prove that  $L_1 \diamond_{df} Q_1 \wedge L_1 \subseteq \mathcal{L}(Q_1)$ . We deduce:

$$\frac{\frac{L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)}{L \diamond_{df} \nabla_f(Q_1) \wedge L \subseteq \mathcal{L}(\nabla_f(Q_1))} \text{Apply } Q = \nabla_f(Q_1)}{\frac{L|_1^* \diamond_{df} Q_1 \wedge L|_1^* \subseteq \mathcal{L}(Q_1)}{L_1 \diamond_{df} Q_1 \wedge L_1 \subseteq \mathcal{L}(Q_1)} \text{Apply } L|_1^* \text{ and semantics of } \nabla_f} \text{Apply } L_1 = L|_1^*$$

Therefore, we conclude that Base Case 6.2 splits the log correctly. In all other cases, base Lemma 4.3.6 applies. Hence, NHD splits logs correctly.  $\square$

**Theorem 6.4.6 — NHD guarantees language rediscoverability.** If the model restrictions detailed above hold for a process tree  $Q$ , then  $\text{NHDdiscover}$  language-rediscovered  $Q$ , i.e.,  $\mathcal{L}(Q) = \mathcal{L}(\text{NHDdiscover}(L))$  for any log  $L$  such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ .

*Proof.* By Lemmas 6.4.4 and 6.4.5, and base Theorem 4.3.7, the NHD algorithm guarantees language rediscoverability.  $\square$

### A.2.4 Polynomial Runtime Complexity

**Theorem 6.4.7 — NHD has polynomial runtime complexity.** The runtime complexity of the NHD algorithm is bounded by  $O((\|\mathbb{A}(L)\| + \|L\|) \cdot \|\mathbb{A}(L)\|^4 + (\|\mathbb{A}(L)\| + \|L\|) \cdot |L|)$ .

*Proof.* We will be reusing the basic IM runtime complexity from Theorem 4.3.8 on page 86. The runtime complexity of the NHD algorithm depends not only on the number of traces in the event log  $c = |L|$  and on the hierarchical alphabet size  $n = \|\mathbb{A}(L)\|$ , but also on the number of hierarchy levels  $h = \|L\|$ . Based on the new Base Cases 6.1 and 6.2, we conclude that the number of recursions made is bounded by:  $O(n + h) = O(\|\mathbb{A}(L)\| + \|L\|)$ .

We get the following recurrence for the NHD runtime complexity  $T(n + h)$ :

$$T(n + h) = \begin{cases} c & \text{for Base Case 4.1 } (n = 1 \wedge h = 1) \\ c + T(n + h - 1) & \text{for Base Case 6.1 or 6.2 } (n = 1 \wedge h > 1) \\ c + n^3 + c + T(n + h - 1) & \text{if a cut was detected } (n > 1) \\ c + n^4 + T(n + h - 1) & \text{if a fallback was used } (n > 1) \end{cases}$$

Since this runtime complexity is dominated by the fallback cases, we hypothesize for any non-negative integer  $k$ :

$$T(n + h) = T(n + h - k) + k \cdot n^4 + k \cdot c$$

Reusing the proof from Theorem 4.3.8 on page 86, and using  $k = n + h$ , with  $T(0) = c$ , we get:

$$\begin{aligned} T(n + h) &= c + (n + h) \cdot n^4 + (n + h) \cdot c \\ &= (n + h) \cdot n^4 + (n + h) \cdot c \end{aligned}$$

Therefore, the runtime complexity of the IM framework is bounded by:

$$\begin{aligned} &O((n + h) \cdot n^4 + (n + h) \cdot c) \\ &= O((\|\mathbb{A}(L)\| + \|L\|) \cdot \|\mathbb{A}(L)\|^4 + (\|\mathbb{A}(L)\| + \|L\|) \cdot |L|) \end{aligned}$$

□

## A.3 Proof for Section 6.5 – Recursion Aware Discovery

The *recursion aware discovery* (RAD) algorithm maintains the IM discovery guarantees from Section 4.3 on page 84 and returns a sound hierarchical process tree (Definition 6.3.3). In this section, we will discuss the proofs for these discovery guarantees and properties.

### A.3.1 Soundness and Termination

**Theorem 6.5.1 — RAD guarantees soundness.** All models  $Q$  returned by the RAD algorithm are guaranteed to be sound.

*Proof.* According to Definition 6.3.3, a hierarchical process tree is not sound if its language contains unresolved recursion markers or a resolved recursion has no option to terminate.

Since Base Case 6.4 checks  $f \in C$  for the involved context path, there is always a matching named subtree  $\nabla_f$  for a recursive reference  $\Delta_f$ . Hence, any model  $Q$  returned by RADiscover cannot yield unresolved recursion markers.

Since Base Case 6.4 will continue updating the involved sublog  $L(C')$  until all observed behavior is included, there will always be an activity or a tau leaf for the empty recursion (e.g., see Example 6.4) to provide a termination option. Hence, any model  $Q$  returned by RADiscover always has an option to terminate.

Hence, all returned hierarchical process trees are sound.  $\square$

**Theorem 6.5.2 — RAD guarantees termination.** The RAD algorithm is guaranteed to always terminate.

*Proof.* Consider Base Cases 6.3 and 6.4. Observe that these base cases do not recurse. Hence, these base cases terminate. By the base Theorem 4.3.2, we know termination is guaranteed for the original step and fallback cases. Hence, the recursion aware instantiation  $\text{RADstep}(L, C)$  is guaranteed to always terminate.

Observe that the event log  $L$  has a finite depth, i.e., a finite number of levels in the hierarchy. Note that the sequence projection  $L|_1^*$  yields strictly smaller event logs, i.e., the number of remaining levels in the hierarchy strictly decreases. Hence, Base Cases 6.3 and 6.4 can only change sublogs finitely often. In addition, since a context path  $C$  is derived from the log depth, and thus finite in size, we also have finitely many sublogs  $L(C)$  that are being used.

Consider the RAD algorithm as given in Algorithm 6.2. Observe that the loop on line 4 is bounded by the number of sublog changes, which are finite, and thus terminates. Observe that the loop on line 7 is bounded by the size of the context path, which is finite, and thus also terminates. Hence, the RAD algorithm is guaranteed to always terminate.  $\square$

### A.3.2 Perfect Fitness

**Theorem 6.5.3 — RAD guarantees fitness.** The RAD algorithm returns a model that fits the log. That is, given an event log  $L$ , the RAD algorithm returns a model  $Q$  such that  $L \subseteq \mathcal{L}(Q)$ .

*Proof.* Based on the proof from the base Theorem 4.3.3, as induction hypothesis (IH), we assume that for all sublogs, the discovery framework returns a fitting model, and then prove that the step maintains this property. That is, for all sublogs  $L_i$  we have by IH a corresponding submodel  $Q_i$  such that  $L_i \subseteq \mathcal{L}(Q_i)$ , and we have to prove  $L \subseteq \mathcal{L}(Q)$ .

Consider Base Case 6.3: *Single Activity – Named Subtree*. That is, we have  $\text{RADiscover}(L) = \nabla_f(Q_{\langle \dots, f \rangle}) = Q$ , and we have to prove  $L \subseteq \mathcal{L}(Q)$ . We deduce:

$$\frac{\text{RADstep}(L, C) = \nabla_f(Q_{\langle \dots, f \rangle})}{\frac{Q_1 = \text{RADstep}(L|_1^*, C')}{\frac{L|_1^* \subseteq \mathcal{L}(Q_1)}{\frac{L \subseteq f.\mathcal{L}(Q_1)}{\frac{L \subseteq \mathcal{L}(\nabla_f(Q_1))}{\frac{L \subseteq \mathcal{L}(\nabla_f(Q_{\langle \dots, f \rangle})})}}}} \text{ Use } L(C') = L|_1^* \text{ and Alg. 6.2, line 5}$$

$$\frac{}{\frac{Q_1 = \text{RADstep}(L|_1^*, C')}{\frac{L|_1^* \subseteq \mathcal{L}(Q_1)}{\frac{L \subseteq f.\mathcal{L}(Q_1)}{\frac{L \subseteq \mathcal{L}(\nabla_f(Q_1))}{\frac{L \subseteq \mathcal{L}(\nabla_f(Q_{\langle \dots, f \rangle})})}}}} \text{ Induction Hypothesis on } L|_1^* \text{ and } Q_1$$

$$\frac{}{\frac{Q_1 = \text{RADstep}(L|_1^*, C')}{\frac{L|_1^* \subseteq \mathcal{L}(Q_1)}{\frac{L \subseteq f.\mathcal{L}(Q_1)}{\frac{L \subseteq \mathcal{L}(\nabla_f(Q_1))}{\frac{L \subseteq \mathcal{L}(\nabla_f(Q_{\langle \dots, f \rangle})})}}}} \text{ Apply } f.(L|_1^*) = L$$

$$\frac{}{\frac{Q_1 = \text{RADstep}(L|_1^*, C')}{\frac{L|_1^* \subseteq \mathcal{L}(Q_1)}{\frac{L \subseteq f.\mathcal{L}(Q_1)}{\frac{L \subseteq \mathcal{L}(\nabla_f(Q_1))}{\frac{L \subseteq \mathcal{L}(\nabla_f(Q_{\langle \dots, f \rangle})})}}}} \text{ By semantics of } \nabla_f: f.\mathcal{L}(Q_1) = \mathcal{L}(\nabla_f(Q_1))$$

$$\frac{}{\frac{Q_1 = \text{RADstep}(L|_1^*, C')}{\frac{L|_1^* \subseteq \mathcal{L}(Q_1)}{\frac{L \subseteq f.\mathcal{L}(Q_1)}{\frac{L \subseteq \mathcal{L}(\nabla_f(Q_1))}{\frac{L \subseteq \mathcal{L}(\nabla_f(Q_{\langle \dots, f \rangle})})}}}} \text{ Apply } Q_1 = Q_{\langle \dots, f \rangle} \text{ and Alg. 6.2, line 9}$$

$$\frac{}{\frac{Q_1 = \text{RADstep}(L|_1^*, C')}{\frac{L|_1^* \subseteq \mathcal{L}(Q_1)}{\frac{L \subseteq f.\mathcal{L}(Q_1)}{\frac{L \subseteq \mathcal{L}(\nabla_f(Q_1))}{\frac{L \subseteq \mathcal{L}(\nabla_f(Q_{\langle \dots, f \rangle})})}}}} \text{ Apply } \nabla_f(Q_{\langle \dots, f \rangle}) = Q$$

Hence, we conclude that for the named subtree operator we return a process model that fits the log  $L$ .

Consider Base Case 6.4: *Single Activity – Recursive Reference*. That is, we have  $\text{RADiscover}(L) = Q$  with  $\Delta_f$  somewhere in  $Q$ , and we have to prove  $L \subseteq \mathcal{L}(Q)$ . We deduce:

$$\frac{\Delta_f \text{ somewhere in } Q}{\frac{\exists Q_1 = \nabla_f(Q_{\langle \dots, f \rangle}) \text{ with } \Delta_f \text{ in } Q_1}{\frac{Q_1 = \text{RADstep}(L|_1^*, C')}{L \subseteq \mathcal{L}(Q)}}} \text{ Use } f \in C \text{ from Base Case 6.4}$$

$$\frac{}{\frac{\exists Q_1 = \nabla_f(Q_{\langle \dots, f \rangle}) \text{ with } \Delta_f \text{ in } Q_1}{\frac{Q_1 = \text{RADstep}(L|_1^*, C')}{L \subseteq \mathcal{L}(Q)}}} \text{ Use } L|_1^* \subseteq L(C') \text{ and Alg. 6.2, line 5}$$

$$\frac{}{\frac{Q_1 = \text{RADstep}(L|_1^*, C')}{L \subseteq \mathcal{L}(Q)}} \text{ Apply proof for Base Case 6.3}$$

Hence, we conclude that for the recursive reference operator we return a process model that fits the log  $L$ .

By the base Theorem 4.3.3, we know fitness is guaranteed for the original step and fallback cases. Hence, the RAD algorithm guarantees fitness.  $\square$

### A.3.3 Language Rediscoverability

**Lemma 6.5.4** — **RAD selects the right tree operator.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the above model restrictions and let  $L$  be a log such that  $L \diamond_{df} Q$ . Then  $\text{RADiscover}(L)$  returns a tree with root  $\otimes$ .

*Proof.* By construction of the base IM framework, base cases are checked before any cut detection or fallback cases. Suppose  $Q = \nabla_f(Q_1)$  for some  $f \in \mathbb{A}$ , then by  $L \diamond_{df} Q$  we know that every event  $e$  in  $L$  starts with  $f$  and there exists an event with a lower level in the hierarchy. Therefore, Base Case 6.3 applies in the run  $Q_{root} = \text{RADstep}(L, \varepsilon)$ , and  $\text{RADiscover}(L)$  returns a tree with root  $\nabla_f$ .

In all other cases, base Lemma 4.3.5 applies. Hence,  $\text{RADiscover}(L)$  returns a tree with root  $\otimes$ .  $\square$

**Lemma 6.5.5 — RAD rediscovers the recursive reference leaf.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the above model restrictions with a leaf  $\Delta_f$  somewhere in  $Q$  and let  $L$  be a log such that  $L \diamond_{df} Q$ . Then  $\text{RADiscover}(L)$  returns a tree with a leaf  $\Delta_f$ .

*Proof.* Since by the above model restrictions we only consider sound hierarchical process trees, there must exist a named subtree  $\nabla_f$  on the path from  $Q$  to  $\Delta_f$ . Hence, as a result of Base Case 6.3, at some point, Base Case 6.4 with  $f \in C$  must apply. Therefore, at some point, there exists a subtree  $Q'$  with the leaf  $\Delta_f$ . Since during discovery sublogs are only changed by adding behavior, once a leaf  $\Delta_f$  is discovered for a submodel  $Q'$ , it will always be rediscovered in subsequent runs. Therefore  $\text{RADiscover}(L)$  returns a tree with a leaf  $\Delta_f$ .  $\square$

**Lemma 6.5.6 — RAD splits logs correctly.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the above model restrictions and let  $L$  be a log such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ . Then for the resulting sublog  $L(C')$  produced by RAD we have  $L(C') \diamond_{df} Q_{C'} \wedge L(C') \subseteq \mathcal{L}(Q_{C'})$ .

*Proof.* Consider Base Cases 6.3 and 6.4. Then we know that  $L \uparrow_1^* \subseteq L(C')$ . We have to prove that  $L(C') \diamond_{df} Q_{C'} \wedge L(C') \subseteq \mathcal{L}(Q_{C'})$ . We deduce:

$$\frac{\frac{L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)}{L \diamond_{df} \nabla_f(Q_{C'}) \wedge L \subseteq \mathcal{L}(\nabla_f(Q_{C'}))} \text{Apply } Q = \nabla_f(Q_{C'})}{\frac{L \uparrow_1^* \diamond_{df} Q_{C'} \wedge L \uparrow_1^* \subseteq \mathcal{L}(Q_{C'})}{L(C') \diamond_{df} Q_{C'} \wedge L(C') \subseteq \mathcal{L}(Q_{C'})} \text{Apply } L \uparrow_1^* \text{ and semantics of } \nabla_f} \text{Apply } L \uparrow_1^* \subseteq L(C')$$

Therefore, we conclude that Base Case 6.2 splits the log correctly. In all other cases, base Lemma 4.3.6 applies. Hence, RAD splits logs correctly.  $\square$

**Theorem 6.5.7 — RAD guarantees language rediscoverability.** If the model restrictions detailed above hold for a process tree  $Q$ , then  $\text{RADiscover}$  language-rediscovered  $Q$ , i.e.,  $\mathcal{L}(Q) = \mathcal{L}(\text{RADiscover}(L))$  for any log  $L$  such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$ .

*Proof.* By Lemmas 6.5.4, 6.5.5 and 6.5.6, and base Theorem 4.3.7, the RAD algorithm guarantees language rediscoverability.  $\square$

### A.3.4 Polynomial Runtime Complexity

**Theorem 6.5.8 — RAD has polynomial runtime complexity.** The runtime complexity of the RAD algorithm is bounded by  $O(\|L\| \cdot \|\mathbb{A}(L)\|^5 + \|L\| \cdot \|\mathbb{A}(L)\| \cdot |L|)$

*Proof.* We will be reusing the basic IM runtime complexity from Theorem 4.3.8 on page 86. Like with the NHD algorithm in Theorem 6.4.7, the runtime complexity of the RAD algorithm depends not only on the number of traces in the event  $\log c = |L|$  and on the hierarchical alphabet size  $n = \|\mathbb{A}(L)\|$ , but also on the number of hierarchy levels  $h = \|L\|$ .

Consider the `RADstep()` algorithm. Based on the new Base Cases 6.1, 6.3 and 6.4, due to the delayed discovery, we conclude that the number of recursions made is bounded by:  $O(n) = O(\|\mathbb{A}(L)\|)$ . Reusing the proof from Theorem 4.3.8 on page 86, we conclude that the runtime complexity of `RADstep()` is bounded by:

$$\begin{aligned} & O(n^5 + n \cdot c) \\ &= O(|\mathbb{A}(L)|^5 + |\mathbb{A}(L)| \cdot |L|) \end{aligned}$$

Consider the RAD algorithm as presented in Algorithm 6.2. The loop on line 4 is bounded by the number of sublog changes. A sublog can only change when new information is found at a lower level in the hierarchy. Therefore, the loop on line 4 is bounded by:  $O(h) = O(\|L\|)$ . During each iteration of this loop, `RADstep()` is called. The loop on line 7 is bounded by  $O(h) = O(\|L\|)$  as well, and each tree substitution can be performed in constant time using a lookup map. Hence, we conclude that the runtime complexity of the RAD algorithm is bounded by:

$$\begin{aligned} & O(h \cdot (n^5 + n \cdot c) + h) \\ &= O(h \cdot n^5 + h \cdot n \cdot c) \\ &= O(\|L\| \cdot \|\mathbb{A}(L)\|^5 + \|L\| \cdot \|\mathbb{A}(L)\| \cdot |L|) \end{aligned}$$

□

## A.4 Proof for Chapter 7 – Cancellation Discovery

The *cancellation discovery* (CD) algorithm maintains the IM discovery guarantees from Section 4.3 on page 84 and returns a sound cancellation process tree (Definition 7.2.3). In this section, we will discuss the proofs for these discovery guarantees and properties.

### A.4.1 Soundness and Termination



**Theorem 7.4.1 — CD guarantees soundness.** All models  $Q$  returned by the CD algorithm are guaranteed to be sound.

*Proof.* According to Definition 7.2.3, a cancelation process tree is not sound if its language contains unresolved trigger markers, or if there is a cancelation path in a cancelation subtree that has no corresponding cancelation trigger (i.e., a dead subtree).

Consider a cancelation path in a cancelation subtree starting with activity  $b$ . Then, by Log Splits 7.1 and 7.2, there are traces in the first sublog that end in an activity  $a$  with  $b \in \text{triggers}(a) \neq \emptyset$ . Therefore, Base Case 7.2 applies eventually. Hence, there exists a cancelation trigger  $\star_a^{\text{triggers}(a)}$  that enabled the cancelation path starting with activity  $b$ .

Consider a cancelation trigger  $\star_a^C$ . Then, by Log Splits 7.1 and 7.2, and Definition 7.4.1, there exists an edge  $(a, b) \in G$  with  $\text{isTrigger}(a, b)$  for every  $b \in C$ . Therefore, Cut Detection 7.1 or 7.2 was applied. Hence, there exists a cancelation subtree providing a cancelation path starting with activity  $b$ , ensuring that every  $b \in C$  is resolved.

Hence, all returned cancelation process trees are sound. □

**Theorem 7.4.2 — CD guarantees termination.** The CD algorithm is guaranteed to always terminate.

*Proof.* Consider Base Cases 7.1 and 7.2. Observe that these base cases return a leaf and do not recurse. Hence, these base cases terminate.

Consider the cancelation cut detection and log splits. By the construction of Log Splits 7.1 and 7.2, the sublogs get strictly smaller upon recursion. By the construction of Cut Detections 7.1 and 7.2, only finitely many partitions or log divisions are created. Hence, the new cancelation operators maintain the termination guarantees.

By the base Theorem 4.3.2, we know termination is guaranteed for the original step and fallback cases. Hence, the CD algorithm is guaranteed to always terminate. □

#### A.4.2 Perfect Fitness

**Theorem 7.4.3 — CD guarantees fitness.** The CD algorithm returns a model that fits the log. That is, given an event log  $L$ , the CD algorithm returns a model  $Q$  such that  $L \subseteq \mathcal{L}(Q)$ .

*Proof.* Based on the proof from the base Theorem 4.3.3, as induction hypothesis (IH), we assume that for all sublogs, the discovery framework returns a fitting model, and then prove that the step maintains this property. That

is, for all sublogs  $L_i$  we have by IH a corresponding submodel  $Q_i$  such that  $L_i \subseteq \mathcal{L}(Q_i)$ , and we have to prove  $L \subseteq \mathcal{L}(Q)$ .

Consider Base Cases 7.1 and 7.2. This proof is analogous to the base Theorem 4.3.3 proof for Base Case 4.2.

Consider the sequence  $(\overset{\star}{\rightarrow})$  and loop  $(\overset{\star}{\circ})$  cancellation join operators. We have to show that these step cases maintain the fitness property. That is, we have  $\text{CDiscover}(L, \text{isTrigger}) = \overset{\star}{\otimes}(\text{CDiscover}(L_1, \text{isTrigger}), \text{CDiscover}(L_2, \text{isTrigger}), \dots, \text{CDiscover}(L_n, \text{isTrigger})) = Q$  with  $\overset{\star}{\otimes} \in \{\overset{\star}{\rightarrow}, \overset{\star}{\circ}\}$ , and we have to prove  $L \subseteq \mathcal{L}(Q)$ . We deduce:

$$\frac{\frac{Q_1 = \text{CDiscover}(L_1, \dots)}{L_1 \subseteq \Phi_{\mathcal{L}}(\mathcal{L}(Q_1))} \text{ IH, Def. 7.2.2} \quad \frac{Q_{i \geq 2} = \text{CDiscover}(L_{i \geq 2}, \dots)}{L_{i \geq 2} \subseteq \mathcal{L}(Q_{i \geq 2})} \text{ IH}}{\overset{\star}{\otimes}_{\mathcal{L}}(L_1, \dots, L_n) \subseteq \mathcal{L}(\overset{\star}{\otimes}(Q_1, \dots, Q_n)) = \mathcal{L}(Q)} \text{ Def. 7.2.2}}{\frac{L \subseteq \overset{\star}{\otimes}_{\mathcal{L}}(L_1, \dots, L_n) \subseteq \mathcal{L}(Q)}{L \subseteq \mathcal{L}(Q)}} \text{ Cut Dect. 7.1, 7.2}$$

Hence, we conclude that for the cancellation operators we return a process model that fits the log  $L$ .

By the base Theorem 4.3.3, we know fitness is guaranteed for the original step and fallback cases. Hence, the CD algorithm guarantees fitness.  $\square$

### A.4.3 Language Rediscoverability

**Lemma 7.4.4 — CD rediscovers the cancellation trigger leaf.** Let  $Q$  be a reduced model that adheres to the above model restrictions with a leaf  $Q' \in \mathbb{A} \cup \{\tau\} \cup \{\star_a^C \mid a \in \mathbb{A}, C \subseteq \mathbb{A}\}$ , let  $L$  be a log such that  $L \diamond_{df} Q$ , and let  $\text{isTrigger}$  be a trigger oracle such that  $\text{isTrigger} \diamond_{\star} Q$ . Then  $\text{CDiscover}(L)$  returns a tree with a leaf  $Q'$ .

*Proof.* Apply case distinction on  $Q'$ .

**Case  $Q' = \tau$ .** Base Case 4.1 applies, thus  $\text{CDiscover}(L)$  returns a tree with a leaf  $\tau$  holds by Lemma 4.3.4.

**Case  $Q' = a$  with  $a \in \mathbb{A}$ .** We assumed  $L \diamond_{df} Q$ , so at some point  $L'$  must be  $\{a\}$ . We assumed  $\text{isTrigger} \diamond_{\star} Q$ , so  $\text{triggers}(a) = \emptyset$ . Hence, Base Case 7.1 must apply at some point, and  $\text{CDiscover}(L') = a$  holds. Therefore  $\text{CDiscover}(L)$  returns a tree with a leaf  $a$ .

**Case  $Q' = \star_a^C$  with  $a \in \mathbb{A}, C \subseteq \mathbb{A}$ .** We assumed  $L \diamond_{df} Q$ , so at some point  $L'$  must be  $\{a\}$ . We assumed  $\text{isTrigger} \diamond_{\star} Q$ , so  $\text{triggers}(a) \neq \emptyset$ . Hence, Base Case 7.2 must apply at some point, and  $\text{CDiscover}(L') = \star_a^{\text{triggers}(a)} = \star_a^C$  holds. Therefore  $\text{CDiscover}(L)$  returns a tree with a leaf  $\star_a^C$ .  $\square$

**Lemma 7.4.5 — CD selects the right tree operator.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the above model restrictions, let  $L$  be a log such that  $L \diamond_{df} Q$ , and let  $isTrigger$  be a trigger oracle such that  $isTrigger \diamond_{\star} Q$ . Then  $CDiscover(L)$  returns a tree with root  $\otimes$ .

*Proof.* Let  $\otimes'$  be the operator selected after cut detection. Let  $\Sigma_1, \dots, \Sigma_n$  be the corresponding cut of  $G(L)$ . Apply case distinction on  $\otimes$ .

**Case  $\otimes = \overset{\star}{\rightarrow}$ .** As  $L$  is directly-follows complete, and by the above model restrictions,  $G$  is connected and therefore the cut detection does not return  $\times$ . Take any submodel  $Q_i$  with  $i > 1$ . By semantics of  $\overset{\star}{\rightarrow}$ , and since  $isTrigger \diamond_{\star} Q$ ,  $G$  only contains trigger-edges from  $Q_1$  to  $Q_i$ , and no edges from  $Q_i$  to  $Q_1$ . Hence, the cut detection does not select  $\rightarrow$ ,  $\wedge$ ,  $\cup$  or  $\overset{\star}{\cup}$ . Since no  $\tau$ 's are allowed, no  $Q_j$  (for any  $j$ ) can produce the empty trace. Therefore all traces in  $L$  start with activities from  $Q_1$ . Hence, we have  $Start(Q_1) = Start(Q)$ . Since also no duplicate activities are allowed, all the cut criteria from Cut Detection 7.1 are satisfied. Hence  $CDiscover(L)$  returns a tree with root  $\overset{\star}{\rightarrow}$ .

**Case  $\otimes = \overset{\star}{\cup}$ .** Because of  $L \diamond_{df} Q$ ,  $G$  is a single strongly connected component. Hence, the cut detection does not return  $\times$ , and by the semantic definitions, also not  $\rightarrow$  or  $\overset{\star}{\rightarrow}$ . By semantics of  $\overset{\star}{\cup}$ , and since  $isTrigger \diamond_{\star} Q$ ,  $G$  only contains trigger-edges from  $Q_1$  to  $Q_i$ , and there exist edges from  $Q_i$  to  $Start(Q_1)$ . Hence, the cut detection doesn't select  $\rightarrow$  or  $\cup$ . We identify the following clusters in  $G(L)$ :  $S = Start(L)$ ,  $E = End(L)$  and  $R = \bigcup_{i \geq 2} \mathbb{A}(Q_i)$ .

By the model restrictions, these clusters are disjoint. As  $\otimes = \overset{\star}{\cup}$ , there is no edge from any node in  $R$  to any node in  $E$ . Therefore, by the semantic definitions, the cut detection does not return  $\wedge$ . Since no  $\tau$ 's are allowed, no  $Q_j$  (for any  $j$ ) can produce the empty trace. Therefore, all traces in  $L$  start and end with activities from  $Q_1$ . Hence  $Start(Q) = Start(Q_1)$  and  $End(Q) = End(Q_1)$ .

For any  $i \geq 2, j \geq 2, i \neq j$ , by semantics of  $\overset{\star}{\cup}$ , no activity of  $Q_i$  can directly follow any activity of  $Q_j$ . Since also  $L \diamond_{df} Q$  holds, all the cut criteria from Cut Detection 7.2 are satisfied. Hence  $findCut'$  returns  $\overset{\star}{\cup}$ .

In all other cases, base Lemma 4.3.5 applies, taking into account the trigger edges. Hence,  $CDiscover(L)$  returns a tree with root  $\otimes$ .  $\square$

**Lemma 7.4.6 — CD splits logs correctly.** Let  $Q = \otimes(Q_1, \dots, Q_n)$  be a reduced model that adheres to the above model restrictions, let  $L$  be a log such that  $L \diamond_{df} Q$ , and let  $isTrigger$  be a trigger oracle such that  $isTrigger \diamond_{\star} Q$ . Then for the resulting sublogs  $L_i$  produced by CD we have  $L_i \diamond_{df} Q_i \wedge L_i \subseteq \mathcal{L}(Q_i)$ .

*Proof.* Let  $G'$  be the undirected version of graph  $G$ . Apply case distinction on  $\otimes$ .

**Case  $\otimes = \overset{\star}{\rightarrow}$ .** By semantics of  $\overset{\star}{\rightarrow}$ ,  $G$  can be seen as having a body  $\mathbb{A}(Q_1)$  and mutually exclusive cancelation alternative parts  $\mathbb{A}(Q_{i \geq 2})$ . Take two activities  $a_i, a'_i \in \mathbb{A}(Q_{i \geq 2})$ . By the reduction rules,  $Q_{i \geq 2}$  is not a  $\times$ , and therefore the directly-follows graph for  $Q_{i \geq 2}$  is connected. By semantics of  $\overset{\star}{\rightarrow}$ , for any  $a_1 \in \mathbb{A}(Q_1)$ , there is no edge  $(a_i, a_1) \in G$ . Hence,  $a_i$  and  $a'_i$  will end up in the same partition  $\Sigma_k$ . Take two activities  $a_i \in \mathbb{A}(Q_{i \geq 2}), a_j \in \mathbb{A}(Q_{j \geq 2})$  with  $i \neq j$ . By semantics of  $\overset{\star}{\rightarrow}$ , there cannot be a path  $a_i \rightsquigarrow a_j \in G'$ . Therefore, there cannot be an edge connecting  $\Sigma_{i \geq 2}$  and  $\Sigma_{j \geq 2}$  in  $G$ . Hence,  $a_i$  and  $a_j$  end up in  $\Sigma_k$  and  $\Sigma_l$  with  $k \neq l$ . Let  $a_1 \in \mathbb{A}(Q_1)$ . Then there exists  $a_s \in \text{Start}(Q)$  such that  $a_s \rightsquigarrow a_1 \in G' \wedge \neg \text{isTrigger}(a_s, a_1)$ . By Log Split 7.1, we have  $a_1 \in \Sigma_1$ . Hence  $\forall i : \mathbb{A}(L_i) = \mathbb{A}(Q_i)$ , where without loss of generality the order of the non-first children is arbitrary.

Pick any  $i \leq n$  and pick any trace  $\sigma \in L_i$ . By the construction of Log Split 7.1, there must be a trace  $\sigma' \in L$  such that  $\sigma$  is a projection of  $\sigma'$ . By lemma assumption,  $\sigma' \subseteq \mathcal{L}(Q)$ . Since no duplicate activities are allowed, the activities of  $\sigma$  in  $\sigma'$  can only be produced by  $Q_i$ . Therefore  $Q'_i$  must have produced  $\sigma$ , and hence,  $L_i \subseteq \mathcal{L}(Q_i)$ .

Left to prove:  $\forall i : L_i \diamond_{df} Q_i$ . We prove the clauses of  $\diamond_{df}$  separately. Pick any two activities  $a, b \in \mathbb{A}(Q_i)$  such that  $\langle \dots, a, b, \dots \rangle \in \mathcal{L}(Q_i)$ . As  $L \diamond_{df} Q$ , and the fact that  $Q$  can produce a trace  $\langle \dots, a, b, \dots \rangle$ , there must be a trace  $\sigma \in L$  such that  $\sigma = \langle \dots, a, b, \dots \rangle$ . By construction of Log Split 7.1, then there will be a trace  $\langle \dots, a, b, \dots \rangle \in L_i$ . Pick an activity  $a \in \text{Start}(Q_i)$ . As  $L \diamond_{df} Q$ , there must be a trace  $\sigma \cdot \langle a \rangle \cdot \sigma' \in L$ , such that  $\mathbb{A}(\sigma) \cap \Sigma_i = \emptyset$ . Then, by construction of Log Split 7.1, there is a trace in  $L_i$  that starts with  $a$ . Hence  $\text{Start}(Q_i) \subseteq \text{Start}(L_i)$ . A similar argument holds for  $\text{End}(Q_i) \subseteq \text{End}(L_i)$ .

**Case  $\otimes = \overset{\star}{\circ}$ .** By semantics of  $\overset{\star}{\circ}$ ,  $G$  can be seen as having a body  $\mathbb{A}(Q_1)$  and mutually exclusive cancelation loop back parts  $\mathbb{A}(Q_{i \geq 2})$ . Take two activities  $a_i, a'_i \in \mathbb{A}(Q_{i \geq 2})$ . By the reduction rules,  $Q_{i \geq 2}$  is not a  $\times$ , and therefore  $Q_{i \geq 2}$  is connected. By semantics of  $\overset{\star}{\circ}$ , for any  $a_e \in \text{End}(Q_1)$ , there is no edge  $(a_i, a_e) \in G$ . Hence,  $a_i$  and  $a'_i$  will end up in the same partition  $\Sigma_k$ . Take two activities  $a_i \in \mathbb{A}(Q_{i \geq 2}), a_j \in \mathbb{A}(Q_{j \geq 2})$  with  $i \neq j$ . By semantics of  $\overset{\star}{\circ}$ , there cannot be a path  $a_i \rightsquigarrow a_j \in G'$ . Therefore, there cannot be an edge connecting  $\Sigma_{i \geq 2}$  and  $\Sigma_{j \geq 2}$  in  $G$ . Hence,  $a_i$  and  $a_j$  end up in  $\Sigma_k$  and  $\Sigma_l$  with  $k \neq l$ .

Let  $a_1 \in \mathbb{A}(Q_1)$ . Then there exists  $a_s \in \text{Start}(Q)$  such that  $a_s \rightsquigarrow a_1 \in G' \wedge \neg \text{isTrigger}(a_s, a_1)$ . By Cut Detection 7.2, we have  $a_1 \in \Sigma_1$ . Hence  $\forall i : \mathbb{A}(L_i) = \mathbb{A}(Q_i)$ , where without loss of generality the order of the non-first children is arbitrary.

Pick any  $i \leq n$  and trace  $\sigma \in L_i$ . Apply case distinction on  $i$  to prove that  $L_i \subseteq \mathcal{L}(Q_i)$ . *Case  $i = 1$ .* By construction of Log Split 7.2, there exists a trace  $\sigma' \cdot \sigma \cdot \sigma'' \in L$ , such that  $\sigma' = \varepsilon$  or ends with an activity  $a' \notin \Sigma_1$ , and  $\sigma'' = \varepsilon$  or

starts with an activity  $a'' \notin \Sigma_1$ . *Case  $i > 1$ .* By construction of Log Split 7.2, there exists a trace  $\sigma' \cdot \langle a' \rangle \cdot \sigma \cdot \langle a'' \rangle \cdot \sigma'' \in L$ , such that  $a', a'' \notin \Sigma_i$ . Since there are no duplicate activities, in both cases, by the semantics of  $\overset{\star}{\circ}$  and the assumption that  $L \subseteq \mathcal{L}(Q)$ , it holds that  $\sigma$  must have been produced by  $Q_i$ . Hence,  $L_i \subseteq \mathcal{L}(Q_i)$ .

Left to prove:  $\forall i : L_i \diamond_{df} Q_i$ . We prove the clauses of  $\diamond_{df}$  separately. Pick any two activities  $a, b \in \mathbb{A}(Q_i)$  such that  $\langle \dots, a, b, \dots \rangle \in \mathcal{L}(Q_i)$ . As  $L \diamond_{df} Q$ , and the fact that  $Q$  can produce a trace  $\langle \dots, a, b, \dots \rangle$ , there must be a trace  $\sigma \in L$  such that  $\sigma = \langle \dots, a, b, \dots \rangle$ . By construction of Log Split 7.2, then there will be a trace  $\langle \dots, a, b, \dots \rangle \in L_i$ . Pick an activity  $a \in \text{Start}(Q_i)$ . As  $L \diamond_{df} Q$ , there must be a trace  $\sigma \cdot \langle a \rangle \cdot \sigma' \in L$ , such that  $\mathbb{A}(\sigma) \cap \Sigma_i = \emptyset$ . Then, by construction of Log Split 7.2, there is a trace in  $L_i$  that starts with  $a$ . Hence  $\text{Start}(Q_i) \subseteq \text{Start}(L_i)$ . A similar argument holds for  $\text{End}(Q_i) \subseteq \text{End}(L_i)$ .

In all other cases, base Lemma 4.3.6 applies, taking into account the trigger edges. Hence for the resulting sublogs  $L_i$  produced by CD we have  $L_i \diamond_{df} Q_i \wedge L_i \subseteq \mathcal{L}(Q_i)$ .  $\square$

**Theorem 7.4.7 — CD guarantees language rediscoverability.** If the model restrictions detailed above hold for a process tree  $Q$ , then  $\text{CDisc}$  language-rediscovered  $Q$ , i.e.,  $\mathcal{L}(Q) = \mathcal{L}(\text{CDisc}(L, \text{isTrigger}))$  for any log  $L$  such that  $L \diamond_{df} Q \wedge L \subseteq \mathcal{L}(Q)$  and any trigger oracle  $\text{isTrigger}$  such that  $\text{isTrigger} \diamond_{\star} Q$ .

*Proof.* By Lemmas 6.5.4, 6.5.5 and 6.5.6, and base Theorem 4.3.7, the CD algorithm guarantees language rediscoverability.  $\square$

#### A.4.4 Polynomial Runtime Complexity

**Theorem 7.4.8 — CD has polynomial runtime complexity.** The runtime complexity of the CD algorithm is bounded by  $O(|\mathbb{A}(L)|^5 + |\mathbb{A}(L)| \cdot |L|)$ .

*Proof.* The new Base Cases 7.1 and 7.2 have a runtime complexity of  $O(|L|)$ . The new Cut Detections 7.1 and 7.2 have runtime complexity  $O(|\mathbb{A}(L)|^3)$  for computing reachability, similar to the sequence cut detection. The new Log Splitting 7.1 and 7.2 have runtime complexity  $O(|L|)$ . In this step, the size of the alphabet is decreased by at least one.

Hence, reusing the proof for Theorem 4.3.8 on page 86, the runtime complexity of the CD algorithm is bounded by  $O(|\mathbb{A}(L)|^5 + |\mathbb{A}(L)| \cdot |L|)$ .  $\square$

# Bibliography

- [1] W. M. P. van der Aalst. “The Application of Petri nets to Workflow Management”. In: *Journal of Circuits, Systems and Computers* 08.01 (1998), pages 21–66. DOI: 10.1142/S0218126698000043. URL: <https://www.worldscientific.com/doi/abs/10.1142/S0218126698000043> (cited on pages 27, 28).
- [2] W. M. P. van der Aalst. “Discovery, Verification and Conformance of Workflows with Cancellation”. In: *Graph Transformations: 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*. Edited by H. Ehrig, R. Heckel, G. Rozenberg, et al. Berlin, Heidelberg: Springer, 2008, pages 18–37. ISBN: 978-3-540-87405-8. DOI: 10.1007/978-3-540-87405-8\_2. URL: [http://dx.doi.org/10.1007/978-3-540-87405-8\\_2](http://dx.doi.org/10.1007/978-3-540-87405-8_2) (cited on pages 59, 62, 63).
- [3] W. M. P. van der Aalst. “Decomposing Process Mining Problems Using Passages”. In: *Application and Theory of Petri Nets: 33rd International Conference, Hamburg, Germany, June 25-29, 2012. Proceedings*. Edited by S. Haddad and L. Pomello. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pages 72–91. ISBN: 978-3-642-31131-4. DOI: 10.1007/978-3-642-31131-4\_5. URL: [https://doi.org/10.1007/978-3-642-31131-4\\_5](https://doi.org/10.1007/978-3-642-31131-4_5) (cited on pages 66, 67, 209).
- [4] W. M. P. van der Aalst. “Decomposing Petri nets for process mining: A generic approach”. In: *Distributed and Parallel Databases* 31.4 (Dec. 2013), pages 471–507. ISSN: 1573-7578. DOI: 10.1007/s10619-013-7127-5. URL: <https://doi.org/10.1007/s10619-013-7127-5> (cited on pages 66, 67, 209).
- [5] W. M. P. van der Aalst. “Process Cubes: Slicing, Dicing, Rolling Up and Drilling Down Event Data for Process Mining”. In: *Asia Pacific Business Process Management*. Edited by M. Song, M. T. Wynn, and J. Liu. Cham: Springer International Publishing, 2013, pages 1–22. ISBN: 978-3-319-02922-1 (cited on page 310).
- [6] W. M. P. van der Aalst. “Service Mining: Using Process Mining to Discover, Check, and Improve Service Behavior”. In: *IEEE Transactions on Services Computing* 6.4 (Oct. 2013), pages 525–535. ISSN: 1939-1374. DOI: 10.1109/TSC.2012.25 (cited on pages 95, 99, 101).
- [7] W. M. P. van der Aalst. “Big Software on the Run: In Vivo Software Analytics Based on Process Mining (Keynote)”. In: *Proceedings of the 2015 International Conference on Software and System Process*. ICSSP 2015. Tallinn, Estonia: ACM, 2015, pages 1–5. ISBN: 978-1-4503-3346-7. DOI: 10.1145/2785592.2785593. URL: <http://doi.acm.org/10.1145/2785592.2785593> (cited on pages 3, 362).
- [8] W. M. P. van der Aalst. *Process Mining: Data Science in Action*. Springer Berlin Heidelberg, 2016. ISBN: 978-3-662-49851-4. DOI: 10.1007/978-3-662-49851-4 (cited on pages 3, 4, 28, 31, 33, 34, 38, 43, 45, 46, 49, 55, 57, 58, 69, 70, 91, 104, 256, 258, 293, 304, 309).

- [9] W. M. P. van der Aalst, A. Adriansyah, A. K. A. de Medeiros, et al. “Process Mining Manifesto”. In: *Business Process Management Workshops*. Edited by F. Daniel, K. Barkaoui, and S. Dustdar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pages 169–194. ISBN: 978-3-642-28108-2 (cited on page 101).
- [10] W. M. P. van der Aalst, A. K. A. Alves de Medeiros, and A. J. M. M. Weijters. “Genetic Process Mining”. In: *Applications and Theory of Petri Nets 2005: 26th International Conference, ICATPN 2005, Miami, USA, June 20–25, 2005. Proceedings*. Edited by G. Ciardo and P. Darondeau. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pages 48–69. ISBN: 978-3-540-31559-9. DOI: 10.1007/11494744\_5. URL: [https://doi.org/10.1007/11494744\\_5](https://doi.org/10.1007/11494744_5) (cited on pages 59, 62–64, 140, 142, 154–157, 189, 192, 199–202).
- [11] W. M. P. van der Aalst, R. De Masellis, C. Di Francescomarino, et al. “Learning Hybrid Process Models from Events”. In: *Business Process Management*. Edited by J. Carmona, G. Engels, and A. Kumar. Cham: Springer International Publishing, 2017, pages 59–76. ISBN: 978-3-319-65000-5 (cited on pages 7, 57).
- [12] W. M. P. van der Aalst, J. Desel, and E. Kindler. “On the semantics of EPCs: A vicious circle”. In: *Proceedings of the EPK 2002: Business Process Management using EPCs*. Gesellschaft für Informatik, Bonn, 2002, pages 71–80 (cited on page 34).
- [13] W. M. P. van der Aalst and B. F. van Dongen. “Discovering Workflow Performance Models from Timed Logs”. In: *Engineering and Deployment of Cooperative Information Systems: First International Conference, EDCIS 2002 Beijing, China, September 17–20, 2002 Proceedings*. Edited by Y. Han, S. Tai, and D. Wikarski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pages 45–63. ISBN: 978-3-540-45785-5. DOI: 10.1007/3-540-45785-2\_4. URL: [https://doi.org/10.1007/3-540-45785-2\\_4](https://doi.org/10.1007/3-540-45785-2_4) (cited on page 70).
- [14] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, et al. “Soundness of workflow nets: classification, decidability, and analysis”. In: *Formal Aspects of Computing* 23.3 (May 2011), pages 333–363. ISSN: 1433-299X. DOI: 10.1007/s00165-010-0161-4. URL: <https://doi.org/10.1007/s00165-010-0161-4> (cited on pages 27–29).
- [15] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, et al. “Workflow Patterns”. In: *Distributed and Parallel Databases* 14.1 (July 2003), pages 5–51. ISSN: 1573-7578. DOI: 10.1023/A:1022883727209. URL: <https://doi.org/10.1023/A:1022883727209> (cited on pages 33, 63, 273, 359).
- [16] W. M. P. van der Aalst, V. Rubin, H. M. W. Verbeek, et al. “Process mining: a two-step approach to balance between underfitting and overfitting”. In: *Software & Systems Modeling* 9.1 (Nov. 2008), page 87. ISSN: 1619-1374. DOI: 10.1007/s10270-008-0106-z. URL: <https://doi.org/10.1007/s10270-008-0106-z> (cited on pages 59, 62–64, 140, 144, 154–157, 189, 193, 199–202).
- [17] W. M. P. van der Aalst and M. Song. “Mining Social Networks: Uncovering Interaction Patterns in Business Processes”. In: *Business Process Management*. Edited by J. Desel, B. Pernici, and M. Weske. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pages 244–260. ISBN: 978-3-540-25970-1 (cited on page 313).
- [18] W. M. P. van der Aalst, A. J. M. M. Weijters, and L. Maruster. “Workflow mining: discovering process models from event logs”. In: *IEEE Transactions on Knowledge and Data Engineering* 16.9 (Sept. 2004), pages 1128–1142. ISSN: 1041-4347. DOI: 10.1109/TKDE.2004.47 (cited on pages 59, 62, 139, 143, 154–157, 188, 191, 199–202).

- 
- [19] Abego. *abego TreeLayout*. <http://treelayout.sourceforge.net/>, <https://github.com/abego/treelayout>. [Online, accessed 12 July 2018] (cited on page 282).
- [20] C. Ackermann, M. Lindvall, and R. Cleaveland. “Recovering Views of Inter-System Interaction Behaviors”. In: *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*. IEEE. Oct. 2009, pages 53–61. DOI: 10.1109/WCRE.2009.34 (cited on pages 59, 61, 62, 69, 70).
- [21] A. Adriansyah. “Aligning observed and modeled behavior”. PhD thesis. Technische Universiteit Eindhoven, Apr. 2014. ISBN: 978-90-386-3574-3. URL: <http://repository.tue.nl/770080> (cited on pages 16, 59, 63, 70, 71, 97, 139, 150, 151, 188, 196, 197, 210, 211, 213, 218, 239, 242, 248, 249, 263, 289, 312, 353).
- [22] A. Adriansyah, B. F. van Dongen, D. A. M. Piessens, et al. “Robust Performance Analysis on YAWL Process Models with Advanced Constructs”. English. In: *JITTA : Journal of Information Technology Theory and Application* 12.3 (Sept. 2011), pages 5–25 (cited on pages 70, 71, 211).
- [23] R. Agrawal, D. Gunopulos, and F. Leymann. “Mining process models from workflow logs”. In: *Advances in Database Technology — EDBT'98*. Edited by H. Schek, G. Alonso, F. Saltor, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pages 467–483. ISBN: 978-3-540-69709-1 (cited on page 62).
- [24] M. H. Alalfi, J. R. Cordy, and T. R. Dean. “Automated Reverse Engineering of UML Sequence Diagrams for Dynamic Web Applications”. In: *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*. IEEE. Apr. 2009, pages 287–294. DOI: 10.1109/ICSTW.2009.8 (cited on pages 59, 61).
- [25] The Apdex Alliance. *Apdex*. <http://www.apdex.org/>. [Online, accessed 27 June 2018] (cited on page 258).
- [26] P. Alvaro, D. V. Ryaboy, and D. Agrawal. “Towards Scalable Architectures for Clickstream Data Warehousing”. In: *Databases in Networked Information Systems*. Edited by S. Bhalla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pages 154–177. ISBN: 978-3-540-75512-8 (cited on page 301).
- [27] A. Amighi, P. de C. Gomes, D. Gurov, et al. “Sound Control-flow Graph Extraction for Java Programs with Exceptions”. In: *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*. SEFM'12. Thessaloniki, Greece: Springer-Verlag, 2012, pages 33–47. ISBN: 978-3-642-33825-0. DOI: 10.1007/978-3-642-33826-7\_3. URL: [http://dx.doi.org/10.1007/978-3-642-33826-7\\_3](http://dx.doi.org/10.1007/978-3-642-33826-7_3) (cited on pages 58–60).
- [28] D. Angluin. “Learning regular sets from queries and counterexamples”. In: *Information and Computation* 75.2 (1987), pages 87–106. ISSN: 0890-5401. DOI: [https://doi.org/10.1016/0890-5401\(87\)90052-6](https://doi.org/10.1016/0890-5401(87)90052-6). URL: <http://www.sciencedirect.com/science/article/pii/0890540187900526> (cited on pages 56, 59–61, 65).
- [29] Apache Commons Documentation Team. *Apache Commons Crypto 1.0.0-src*. <http://commons.apache.org/proper/commons-crypto>. [Online, accessed 3 March 2017] (cited on pages 96, 151).
- [30] Apache Software Foundation. *Apache HTTP Server*. <https://httpd.apache.org/>. [Online, accessed 25 Juli 2018] (cited on page 91).
- [31] AppDynamics. *AppDynamics*. <https://www.appdynamics.com/>. [Online, accessed 2 Februari 2018] (cited on pages 56, 92, 95).



- [32] ATT Research and Lucent Bell Labs. *Graphviz*. <https://www.graphviz.org/>. [Online, accessed 19 Februari 2018] (cited on pages 253, 282, 298, 299).
- [33] A. Augusto, R. Conforti, M. Dumas, et al. “Automated Discovery of Structured Process Models: Discover Structured vs. Discover and Structure”. In: *Conceptual Modeling: 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings*. Edited by I. Comyn-Wattiau, K. Tanaka, I. Song, et al. Cham: Springer International Publishing, 2016, pages 313–329. ISBN: 978-3-319-46397-1. DOI: 10.1007/978-3-319-46397-1\_25. URL: [http://dx.doi.org/10.1007/978-3-319-46397-1\\_25](http://dx.doi.org/10.1007/978-3-319-46397-1_25) (cited on pages 59, 62, 63).
- [34] T. Berg, O. Grinchtein, B. Jonsson, et al. “On the Correspondence Between Conformance Testing and Regular Inference”. In: *Fundamental Approaches to Software Engineering*. Edited by M. Cerioli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pages 175–189. ISBN: 978-3-540-31984-9 (cited on page 55).
- [35] R. Bergenthum, J. Desel, R. Lorenz, et al. “Process Mining Based on Regions of Languages”. In: *Business Process Management: 5th International Conference, BPM 2007, Brisbane, Australia, September 24-28, 2007. Proceedings*. Edited by G. Alonso, P. Dadam, and M. Rosemann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pages 375–383. ISBN: 978-3-540-75183-0. DOI: 10.1007/978-3-540-75183-0\_27. URL: [https://doi.org/10.1007/978-3-540-75183-0\\_27](https://doi.org/10.1007/978-3-540-75183-0_27) (cited on pages 59, 62, 63).
- [36] S. Bernardi, S. Donatelli, and J. Merseguer. “From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models”. In: *Proceedings of the 3rd International Workshop on Software and Performance*. WOSP '02. Rome, Italy: ACM, 2002, pages 35–45. ISBN: 1-58113-563-7. DOI: 10.1145/584369.584376. URL: <http://doi.acm.org/10.1145/584369.584376> (cited on page 37).
- [37] I. Beschastnikh, J. Abrahamson, Y. Brun, et al. “Synoptic: Studying Logged Behavior with Inferred Models”. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ES-EC/FSE '11. Szeged, Hungary: ACM, 2011, pages 448–451. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025188. URL: <http://doi.acm.org/10.1145/2025113.2025188> (cited on pages 59, 69, 70, 141, 145, 154–157, 189, 193, 199–202).
- [38] I. Beschastnikh, Y. Brun, M. D. Ernst, et al. “Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pages 468–479. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568246. URL: <http://doi.acm.org/10.1145/2568225.2568246> (cited on pages 59, 61, 64, 65).
- [39] K. Bhattacharya, C. Gerede, R. Hull, et al. “Towards Formal Analysis of Artifact-Centric Business Process Models”. In: *Business Process Management*. Edited by G. Alonso, P. Dadam, and M. Rosemann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pages 288–304. ISBN: 978-3-540-75183-0 (cited on pages 99, 101).
- [40] S. Birkner, E. Gamma, and K. Beck. *JUnit 4 - Getting Started*. <https://github.com/junit-team/junit4/wiki/Getting-started>. [Online, accessed 19 July 2016] (cited on pages 96, 151, 254, 324).
- [41] M. Bozkaya, J. Gabriels, and J. M. van der Werf. “Process Diagnostics: A Method Based on Process Mining”. In: *2009 Int. Conf. on Information, Process, and Knowledge Management*. Feb. 2009, pages 22–27. DOI: 10.1109/eKNOW.2009.29 (cited on pages 304, 306).

- 
- [42] L. C. Briand, Y. Labiche, and J. Leduc. “Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software”. In: *Software Engineering, IEEE Transactions on* 32.9 (Sept. 2006), pages 642–663. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.96 (cited on pages 59, 61, 64, 65).
- [43] L. C. Briand, Y. Labiche, and Y. Miao. “Towards the Reverse Engineering of UML Sequence Diagrams”. In: *2013 20th Working Conference on Reverse Engineering (WCRE)* (2003), page 57. ISSN: 1095-1350. DOI: 10.1109/WCRE.2003.1287237. URL: <http://doi.ieeecomputersociety.org/10.1109/WCRE.2003.1287237> (cited on pages 59, 61).
- [44] S. K. L. M. vanden Broucke. “Advances in Process Mining: Artificial negative events and other techniques”. PhD thesis. Katholieke Universiteit Leuven, Belgium, 2014. URL: <https://lirias.kuleuven.be/handle/123456789/459143> (cited on page 64).
- [45] J. C. A. M. Buijs. *Receipt phase of an environmental permit application process ('WABO'), CoSeLoG project*. en. <https://doi.org/10.4121/uuid:a07386a5-7be3-4367-9535-70bc9e77dbe6>. 2014. DOI: 10.4121/uuid:a07386a5-7be3-4367-9535-70bc9e77dbe6. URL: <https://doi.org/10.4121/uuid:a07386a5-7be3-4367-9535-70bc9e77dbe6> (cited on pages 96, 97, 150–152, 196, 197).
- [46] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. “On the Role of Fitness, Precision, Generalization and Simplicity in Process Discovery”. English. In: *On the Move to Meaningful Internet Systems: OTM 2012*. Edited by R. Meersman, H. Panetto, T. Dillon, et al. Volume 7565. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pages 305–322. ISBN: 978-3-642-33605-8. DOI: 10.1007/978-3-642-33606-5\_19. URL: [http://dx.doi.org/10.1007/978-3-642-33606-5\\_19](http://dx.doi.org/10.1007/978-3-642-33606-5_19) (cited on pages 7, 59, 62, 63, 140, 144, 154–157, 189, 192, 199–202).
- [47] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. “Quality Dimensions in Process Discovery: The Importance of Fitness, Precision, Generalization and Simplicity”. In: *International Journal of Cooperative Information Systems* 23.01 (2014), page 1440001. DOI: 10.1142/S0218843014400012. URL: <https://www.worldscientific.com/doi/abs/10.1142/S0218843014400012> (cited on page 64).
- [48] J. Cardoso and W. M. P. van der Aalst. *Semantic Web Services: Theory, Tools and Applications: Theory, Tools and Applications*. IGI Global research collection. Information Science Reference, 2007. ISBN: 9781599040479. URL: <https://books.google.nl/books?id=5kQCBUBfNQC> (cited on page 38).
- [49] J. Carmona, J. Cortadella, and M. Kishinevsky. “A Region-Based Algorithm for Discovering Petri Nets from Event Logs”. In: *Business Process Management: 6th International Conference, BPM 2008, Milan, Italy, September 2-4, 2008. Proceedings*. Edited by M. Dumas, M. Reichert, and M. Shan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 358–373. ISBN: 978-3-540-85758-7. DOI: 10.1007/978-3-540-85758-7\_26. URL: [http://dx.doi.org/10.1007/978-3-540-85758-7\\_26](http://dx.doi.org/10.1007/978-3-540-85758-7_26) (cited on pages 59, 62, 63).
- [50] S. Cassel, F. Howar, B. Jonsson, et al. “Active learning for extended finite state machines”. In: *Formal Aspects of Computing* 28.2 (Apr. 2016), pages 233–263. ISSN: 1433-299X. DOI: 10.1007/s00165-016-0355-5. URL: <https://doi.org/10.1007/s00165-016-0355-5> (cited on pages 59, 61).
- [51] Celonis GmbH. *Celonis Process Mining*. <https://www.celonis.com>. [Online, accessed 06 July 2017] (cited on pages 7, 59, 62, 70, 279).

- [52] S. Chiba. “Javassist – A Reflection-based Programming Wizard for Java”. In: *Proceedings of OOPSLA98 Workshop on Reflective Programming in C++ and Java*. Oct. 1998, page 5 (cited on pages 93, 290).
- [53] S. Chiba. “Load-Time Structural Reflection in Java”. English. In: *European Conference on Object-Oriented Programming 2000 – Object-Oriented Programming*. Edited by E. Bertino. Volume 1850. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pages 313–336. ISBN: 978-3-540-67660-7. DOI: 10.1007/3-540-45102-1\_16. URL: [http://dx.doi.org/10.1007/3-540-45102-1\\_16](http://dx.doi.org/10.1007/3-540-45102-1_16) (cited on pages 93, 290).
- [54] M. Chow, D. Meisner, J. Flinn, et al. “The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI’14. Broomfield, CO: USENIX Association, 2014, pages 217–231. ISBN: 978-1-931971-16-4. URL: <http://dl.acm.org/citation.cfm?id=2685048.2685066> (cited on pages 59, 61, 62, 69, 70, 92, 93, 95).
- [55] R. Conforti, M. Dumas, L. García-Banuelos, et al. “BPMN Miner: Automated discovery of BPMN process models with hierarchical structure”. In: *Information Systems* 56 (2016), pages 284–303. ISSN: 0306-4379. DOI: <http://dx.doi.org/10.1016/j.is.2015.07.004>. URL: <http://www.sciencedirect.com/science/article/pii/S0306437915001325> (cited on pages 59, 62, 63, 66, 209, 210).
- [56] J. E. Cook and A. L. Wolf. “Discovering Models of Software Processes from Event-based Data”. In: *ACM Transactions on Software Engineering and Methodology, TOSEM* 7.3 (July 1998), pages 215–249. ISSN: 1049-331X. DOI: 10.1145/287000.287001. URL: <http://doi.acm.org/10.1145/287000.287001> (cited on page 62).
- [57] J. R. Cordy. “The TXL source transformation language”. In: *Science of Computer Programming* 61.3 (2006). Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 04), pages 190–210. ISSN: 0167-6423. DOI: <http://dx.doi.org/10.1016/j.scico.2006.04.002>. URL: <http://www.sciencedirect.com/science/article/pii/S0167642306000669> (cited on page 93).
- [58] J. Cortadella, M. Kishinevsky, L. Lavagno, et al. “Deriving Petri nets from finite transition systems”. In: *IEEE Transactions on Computers* 47.8 (Aug. 1998), pages 859–882. ISSN: 0018-9340. DOI: 10.1109/12.707587 (cited on pages 59, 62).
- [59] W. Damm and D. Harel. “LSCs: Breathing Life into Message Sequence Charts”. In: *Formal Methods in System Design* 19.1 (July 2001), pages 45–80. ISSN: 1572-8102. DOI: 10.1023/A:1011227529550. URL: <https://doi.org/10.1023/A:1011227529550> (cited on page 38).
- [60] M. De Leoni and F. Mannhardt. *Road Traffic Fine Management Process*. en. 2015. DOI: 10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5. URL: <https://doi.org/10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5> (cited on pages 96, 197).
- [61] W. De Pauw, D. Lorenz, J. Vlissides, et al. “Execution Patterns in Object-oriented Visualization”. In: *Proceedings of the 4th Conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4*. COOTS’98. Santa Fe, New Mexico: USENIX Association, 1998, pages 16–16. URL: <http://dl.acm.org/citation.cfm?id=1268009.1268025> (cited on pages 59, 61, 62, 69, 70).
- [62] J. De Ruiter and E. Poll. “Protocol State Fuzzing of TLS Implementations.” In: *Proceedings of the 24th USENIX Security Symposium*. USENIX, Aug. 2015. URL: <https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-de-ruiter.pdf> (cited on page 55).

- 
- [63] B. F. van Dongen. *BPI Challenge 2012*. 2012. DOI: 10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f. URL: <http://dx.doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f> (cited on pages 96, 97, 150–152, 196, 197).
- [64] M. L. van Eck, X. Lu, S. J. J. Leemans, et al. “PM<sup>2</sup>: A Process Mining Project Methodology”. In: *Advanced Information Systems Engineering*. Edited by J. Zdravkovic, M. Kirikova, and P. Johannesson. Cham: Springer International Publishing, 2015, pages 297–313. ISBN: 978-3-319-19069-3 (cited on pages 304, 306, 309).
- [65] T. Elrad, Robert E. Filman, and A. Bader. “Aspect-oriented Programming: Introduction”. In: *Communications of the ACM* 44.10 (Oct. 2001), pages 29–32. ISSN: 0001-0782. DOI: 10.1145/383845.383853. URL: <http://doi.acm.org/10.1145/383845.383853> (cited on pages 93, 290).
- [66] W. Fokkink. *Introduction to Process Algebra*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 3642085849, 9783642085840 (cited on page 38).
- [67] E. Gamma and K. Beck. *JUnit 4.12*. <https://mvnrepository.com/artifact/junit/junit/4.12>. [Online, accessed 19 July 2016] (cited on pages 96, 151, 254, 324).
- [68] E. Gamma, R. Helm, R. Johnson, et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2 (cited on pages 330, 333).
- [69] E. González López de Murillas, H. A. Reijers, and W. M. P. van der Aalst. “Connecting Databases with Process Mining: A Meta Model and Toolset”. In: *Enterprise, Business-Process and Information Systems Modeling: 17th International Conference, BPMDS 2016, 21st International Conference, EMMSAD 2016, Held at CAiSE 2016, Ljubljana, Slovenia, June 13-14, 2016, Proceedings*. Edited by R. Schmidt, W. Guédria, I. Bider, et al. Cham: Springer International Publishing, 2016, pages 231–249. ISBN: 978-3-319-39429-9. DOI: 10.1007/978-3-319-39429-9\_15. URL: [https://doi.org/10.1007/978-3-319-39429-9\\_15](https://doi.org/10.1007/978-3-319-39429-9_15) (cited on pages 66, 68, 209, 210).
- [70] Google. *Google Analytics*. <https://www.google.com/analytics>. [Online, accessed 25 Juli 2018] (cited on page 92).
- [71] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ. Aspect-Oriented Programming in Java*. Volume 456. John Wiley & Sons, 2003 (cited on page 93).
- [72] Gradient ECM. *Minit*. <https://www.minit.io/>. [Online, accessed 06 July 2017] (cited on pages 7, 59, 62, 70).
- [73] S. L. Graham, P. B. Kessler, and M. K. Mckusick. “gprof: call graph execution profiler”. In: *Proceedings of the 1982 SIGPLAN symposium on Compiler construction - SIGPLAN '82*. New York, New York, USA: ACM Press, 1982, pages 120–126. ISBN: 0897910745. DOI: 10.1145/800230.806987. URL: <http://portal.acm.org/citation.cfm?doid=800230.806987> (cited on pages 56, 59, 60, 69, 70, 293).
- [74] P. Graubmann, E. Rudolph, and J. Grabowski. “Towards a Petri net based semantics definition for Message Sequence Charts”. In: *SDL '93, using objects : proceedings of the Sixth SDL Forum*. Edited by O. Færgemand and A. Sarma. Volume 93. Darmstadt, Germany: North-Holland, Oct. 1993, pages 415–418. ISBN: 9780444814869. URL: <https://books.google.nl/books?id=TrlZAAAYAAJ> (cited on page 37).

- [75] C. W. Günther and W. M. P. van der Aalst. “Fuzzy Mining – Adaptive Process Simplification Based on Multi-perspective Metrics”. In: *Business Process Management: 5th International Conference, BPM 2007, Brisbane, Australia, September 24–28, 2007. Proceedings*. Edited by G. Alonso, P. Dadam, and M. Rosemann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pages 328–343. ISBN: 978-3-540-75183-0. DOI: 10.1007/978-3-540-75183-0\_24. URL: [https://doi.org/10.1007/978-3-540-75183-0\\_24](https://doi.org/10.1007/978-3-540-75183-0_24) (cited on pages 59, 62, 63, 66, 68, 70, 139, 143, 154, 155, 189, 191, 199, 200).
- [76] C. W. Günther and A. Rozinat. “Disco: Discover Your Processes”. In: *Proceedings of the Demonstration Track of the 10th International Conference on Business Process Management (BPM 2012)*. Edited by N. Lohmann and S. Moser. Volume 940. CEUR Workshop Proceedings, 2012, pages 40–44. URL: <http://ceur-ws.org/Vol-940/paper8.pdf> (cited on pages 7, 59, 62, 70, 279, 311).
- [77] C. W. Günther and H. M. W. Verbeek. *XES – standard definition*. Technical report BPM reports 1409. Eindhoven University of Technology, 2014, page 24. DOI: URN: NBN:NL:UI:25-777826. URL: <http://repository.tue.nl/777826> (cited on pages 16, 46, 47).
- [78] A. Hagerer, H. Hungar, O. Niese, et al. “Model Generation by Moderated Regular Extrapolation”. In: *Fundamental Approaches to Software Engineering*. Edited by R. Kutsche and H. Weber. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pages 80–95. ISBN: 978-3-540-45923-1 (cited on page 55).
- [79] L. Hardesty. *Making computers explain themselves*. <http://news.mit.edu/2016/making-computers-explain-themselves-machine-learning-1028>. Oct. 2016 (cited on page 362).
- [80] D. Harel. “Statecharts: a visual formalism for complex systems”. In: *Science of Computer Programming* 8.3 (1987), pages 231–274. ISSN: 0167-6423. DOI: [http://dx.doi.org/10.1016/0167-6423\(87\)90035-9](http://dx.doi.org/10.1016/0167-6423(87)90035-9). URL: <http://www.sciencedirect.com/science/article/pii/0167642387900359> (cited on page 35).
- [81] D. Harel and H. Kugler. “The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)”. In: *Integration of Software Specification Techniques for Applications in Engineering: Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*. Edited by H. Ehrig, W. Damm, J. Desel, et al. Volume 3147. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pages 325–354. ISBN: 978-3-540-27863-4. DOI: 10.1007/978-3-540-27863-4\_19. URL: [http://dx.doi.org/10.1007/978-3-540-27863-4\\_19](http://dx.doi.org/10.1007/978-3-540-27863-4_19) (cited on page 35).
- [82] D. Harel and A. Naamad. “The STATEMATE Semantics of Statecharts”. In: *ACM Trans. Softw. Eng. Methodol.* 5.4 (Oct. 1996), pages 293–333. ISSN: 1049-331X. DOI: 10.1145/235321.235322. URL: <http://doi.acm.org/10.1145/235321.235322> (cited on page 35).
- [83] D. Harel, A. Pnueli, J. Schmidt, et al. “On the formal semantics of Statecharts”. In: *Proceedings of Symposium on Logic in Computer Science*. Ithaca, USA, June 1987, pages 55–64. URL: [https://www.researchgate.net/publication/236157145\\_On\\_the\\_formal\\_semantics\\_of\\_statecharts](https://www.researchgate.net/publication/236157145_On_the_formal_semantics_of_statecharts) (cited on page 35).
- [84] J. Herbst. “A Machine Learning Approach to Workflow Management”. In: *Machine Learning: ECML 2000*. Edited by R. López de Mántaras and E. Plaza. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pages 183–194. ISBN: 978-3-540-45164-8 (cited on page 64).

- 
- [85] M. J. H. Heule and S. Verwer. “Exact DFA Identification Using SAT Solvers”. In: *Grammatical Inference: Theoretical Results and Applications: 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings*. Edited by J. M. Sempere and P. García. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 66–79. ISBN: 978-3-642-15488-1. DOI: 10.1007/978-3-642-15488-1\_7. URL: [https://doi.org/10.1007/978-3-642-15488-1\\_7](https://doi.org/10.1007/978-3-642-15488-1_7) (cited on pages 55, 59, 61, 65).
- [86] A. H. M. ter Hofstede, W. M. P. van der Aalst, M. Adams, et al. *Modern Business Process Automation: YAWL and its Support Environment*. Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-03121-2. DOI: 10.1007/978-3-642-03121-2 (cited on page 33).
- [87] B. F. A. Hompes, J. C. A. M. Buijs, and W. M. P. van der Aalst. “A Generic Framework for Context-Aware Process Performance Analysis”. In: *On the Move to Meaningful Internet Systems: OTM 2016 Conferences: Confederated International Conferences: CoopIS, C&TC, and ODBASE 2016, Rhodes, Greece, October 24-28, 2016, Proceedings*. Edited by C. Debruyne, H. Panetto, R. Meersman, et al. Cham: Springer International Publishing, 2016, pages 300–317. ISBN: 978-3-319-48472-3. DOI: 10.1007/978-3-319-48472-3\_17. URL: [https://doi.org/10.1007/978-3-319-48472-3\\_17](https://doi.org/10.1007/978-3-319-48472-3_17) (cited on page 70).
- [88] B. F. A. Hompes, H. M. W. Verbeek, and W. M. P. van der Aalst. “Finding Suitable Activity Clusters for Decomposed Process Discovery”. In: *Data-Driven Process Discovery and Analysis: 4th International Symposium, SIMPDA 2014, Milan, Italy, November 19-21, 2014, Revised Selected Papers*. Edited by P. Ceravolo, B. Russo, and R. Accorsi. Cham: Springer International Publishing, 2015, pages 32–57. ISBN: 978-3-319-27243-6. DOI: 10.1007/978-3-319-27243-6\_2. URL: [https://doi.org/10.1007/978-3-319-27243-6\\_2](https://doi.org/10.1007/978-3-319-27243-6_2) (cited on pages 66, 67).
- [89] A. van Hoorn, J. Waller, and W. Hasselbring. “Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis”. In: *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, Apr. 2012, pages 247–248 (cited on pages 59, 60, 69, 70, 92).
- [90] H. Hungar, O. Niese, and B. Steffen. “Domain-Specific Optimization in Automata Learning”. In: *Computer Aided Verification*. Edited by W. A. Hunt and F. Somenzi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pages 315–327. ISBN: 978-3-540-45069-6 (cited on page 55).
- [91] M. Isberner. “Foundations of active automata learning: an algorithmic perspective”. PhD thesis. Technische Universität Dortmund, Oct. 2015. URL: <https://d-nb.info/1110893728/34> (cited on page 55).
- [92] M. Isberner, F. Howar, and B. Steffen. “The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning”. In: *Runtime Verification*. Edited by B. Bonakdarpour and S. A. Smolka. Cham: Springer International Publishing, 2014, pages 307–322. ISBN: 978-3-319-11164-3 (cited on pages 59, 61).
- [93] R. P. Jagadeesh Chandra Bose and W. M. P. van der Aalst. “Abstractions in Process Mining: A Taxonomy of Patterns”. In: *Business Process Management: 7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009. Proceedings*. Edited by U. Dayal, J. Eder, J. Koehler, et al. Volume 5701. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pages 159–175. ISBN: 978-3-642-03848-8. DOI: 10.1007/978-3-642-03848-8\_12. URL: [http://dx.doi.org/10.1007/978-3-642-03848-8\\_12](http://dx.doi.org/10.1007/978-3-642-03848-8_12) (cited on pages 66, 68, 210).

- [94] R. P. Jagadeesh Chandra Bose, H. M. W. Verbeek, and W. M. P. van der Aalst. "Discovering Hierarchical Process Models Using ProM". In: *IS Olympics: Information Systems in a Diverse World: CAiSE Forum 2011, London, UK, June 20-24, 2011, Selected Extended Papers*. Edited by S. Nurcan. Volume 107. Lecture Notes in Business Information Processing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pages 33–48. ISBN: 978-3-642-29749-6. DOI: 10.1007/978-3-642-29749-6\_3. URL: [http://dx.doi.org/10.1007/978-3-642-29749-6\\_3](http://dx.doi.org/10.1007/978-3-642-29749-6_3) (cited on pages 59, 62, 63, 66, 68, 210).
- [95] K. Jensen. "Coloured Petri nets". In: *Petri Nets: Central Models and Their Properties: Advances in Petri Nets 1986, Part I Proceedings of an Advanced Course Bad Honnef, 8–19 September 1986*. Edited by W. Brauer, W. Reisig, and G. Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pages 248–299. ISBN: 978-3-540-47919-2. DOI: 10.1007/BFb0046842. URL: <https://doi.org/10.1007/BFb0046842> (cited on page 102).
- [96] I. Jonyer, L. B. Holder, and D. J. Cook. "MDL-Based Context-Free Graph Grammar Induction". In: *International Journal on Artificial Intelligence Tools* 13.01 (Mar. 2004), pages 65–79. ISSN: 0218-2130. DOI: 10.1142/S0218213004001429. URL: <http://www.worldscientific.com/doi/abs/10.1142/S0218213004001429> (cited on pages 59, 61, 62).
- [97] A. A. Kalenkova and I. A. Lomazova. "Discovery of Cancellation Regions Within Process Mining Techniques". In: *Fundamenta Informaticae* 133.2-3 (Apr. 2014), pages 197–209. ISSN: 0169-2968. DOI: 10.3233/FI-2014-1071. URL: <http://dx.doi.org/10.3233/FI-2014-1071> (cited on pages 59, 62, 63, 66, 68, 197, 199–202, 210).
- [98] E. Kindler. "On the Semantics of EPCs: A Framework for Resolving the Vicious Circle". In: *Business Process Management*. Edited by J. Desel, B. Pernici, and M. Weske. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pages 82–97. ISBN: 978-3-540-25970-1 (cited on page 34).
- [99] KNIME. *KNIME Analytics Platform*. <https://www.knime.com/> (cited on page 320).
- [100] R. Kollmann and M. Gogolla. "Capturing dynamic program behaviour with UML collaboration diagrams". In: *Software Maintenance and Reengineering, 2001. Fifth European Conference on*. 2001, pages 58–67. DOI: 10.1109/.2001.914969 (cited on pages 56, 58, 59).
- [101] E. Korshunova, M. Petkovic, M. G. J. van den Brand, et al. "CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code". In: *2006 13th Working Conference on Reverse Engineering*. Oct. 2006, pages 297–298. DOI: 10.1109/WCRE.2006.21 (cited on pages 58–60).
- [102] L. A. Kurgan and P. Musilek. "A survey of Knowledge Discovery and Data Mining process models". In: *The Knowledge Engineering Review* 21.1 (2006), 124. DOI: 10.1017/S0269888906000737 (cited on page 304).
- [103] Y. Labiche, B. Kolbah, and H. Mehrfard. "Combining Static and Dynamic Analyses to Reverse-Engineer Scenario Diagrams". In: *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE. Sept. 2013, pages 130–139. DOI: 10.1109/ICSM.2013.24 (cited on pages 59, 61).
- [104] P. B. Ladkin and S. Leue. "Interpreting Message Flow Graphs". In: *Formal Aspects of Computing* 7.5 (Sept. 1995), pages 473–509. ISSN: 1433-299X. DOI: 10.1007/BF01211629. URL: <https://doi.org/10.1007/BF01211629> (cited on page 37).

- [105] D. Latella, I. Majzik, and M. Massink. “Towards a Formal Operational Semantics of UML Statechart Diagrams”. In: *Formal Methods for Open Object-Based Distributed Systems: IFIP TC6 / WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), February 15–18, 1999, Florence, Italy*. Edited by P. Ciancarini, A. Fantechi, and R. Gorrieri. Volume 10. The International Federation for Information Processing. Boston, MA: Springer US, 1999, pages 331–347. ISBN: 978-0-387-35562-7. DOI: 10.1007/978-0-387-35562-7\_25. URL: [http://dx.doi.org/10.1007/978-0-387-35562-7\\_25](http://dx.doi.org/10.1007/978-0-387-35562-7_25) (cited on page 35).
- [106] B. Laugwitz, T. Held, and M. Schrepp. “Construction and Evaluation of a User Experience Questionnaire”. In: *HCI and Usability for Education and Work: 4th Symposium of the Workgroup Human-Computer Interaction and Usability Engineering of the Austrian Computer Society, USAB 2008, Graz, Austria, November 20-21, 2008. Proceedings*. Edited by A. Holzinger. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 63–76. ISBN: 978-3-540-89350-9. DOI: 10.1007/978-3-540-89350-9\_6. URL: [https://doi.org/10.1007/978-3-540-89350-9\\_6](https://doi.org/10.1007/978-3-540-89350-9_6) (cited on pages 297, 299, 300).
- [107] M. Leemans. *Episode miner plugin for ProM 6*. <https://svn.win.tue.nl/repos/prom/Packages/EpisodeMiner/>. 2015 (cited on page 409).
- [108] M. Leemans. *JUnit 4.12 Software Event Log*. 2016. DOI: 10.4121/uuid:cfed8007-91c8-4b12-98d8-f233e5cd25bb. URL: <http://doi.org/10.4121/uuid:cfed8007-91c8-4b12-98d8-f233e5cd25bb> (cited on pages 97, 150, 152, 254, 283, 296, 409).
- [109] M. Leemans. *Apache Commons Crypto 1.0.0 - Stream CbcNopad Unit Test Software Event Log*. 2017. DOI: 10.4121/uuid:bb3286d6-dde1-4e74-9a64-fd4e32f10677. URL: <http://doi.org/10.4121/uuid:bb3286d6-dde1-4e74-9a64-fd4e32f10677> (cited on pages 97, 150, 152, 409).
- [110] M. Leemans. *NASA Crew Exploration Vehicle (CEV) Software Event Log*. 2017. DOI: 10.4121/uuid:60383406-ffcd-441f-aa5e-4ec763426b76. URL: <http://doi.org/10.4121/uuid:60383406-ffcd-441f-aa5e-4ec763426b76> (cited on pages 97, 150, 152, 196, 197, 409).
- [111] M. Leemans. *Recursion Aware Modeling and Discovery For Hierarchical Software Event Log Analysis (Extended) - Technical Report version with guarantee proofs for the discovery algorithms*. Technical report. <https://arxiv.org/abs/1710.09323>. Eindhoven University of Technology, 2017 (cited on page 408).
- [112] M. Leemans. *Statechart Workbench and Alignments Software Event Log*. 2018. DOI: 10.4121/uuid:7f787965-da13-4bb8-a3fd-242f08aef9c4. URL: <http://doi.org/10.4121/uuid:7f787965-da13-4bb8-a3fd-242f08aef9c4> (cited on pages 97, 150, 152, 196, 197, 409).
- [113] M. Leemans. *SAW Eclipse plugin for the Statechart Workbench*. <https://svn.win.tue.nl/repos/prom/XPort/> (cited on pages 278, 409).
- [114] M. Leemans. *Statechart plugin for ProM 6*. <https://svn.win.tue.nl/repos/prom/Packages/Statechart/> (cited on pages 278, 409).
- [115] M. Leemans. *Statechart Workbench*. <https://youtu.be/xR4xfU3E5mk> (cited on pages 278, 409).
- [116] M. Leemans. *Statechart Workbench User Manual*. <https://svn.win.tue.nl/repos/prom/Packages/Statechart/Trunk/doc/Manual.pdf> (cited on pages 278, 283, 296).



- [117] M. Leemans and W. M. P. van der Aalst. “Discovery of frequent episodes in event logs”. English. In: *4th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2014, Milan, Italy, November 19-21, 2014)*. Edited by R. Accorsi, P. Ceravolo, and B. Russo. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pages 31–45 (cited on page 408).
- [118] M. Leemans and W. M. P. van der Aalst. “Discovery of Frequent Episodes in Event Logs (Extended)”. In: *Data-Driven Process Discovery and Analysis*. Edited by P. Ceravolo, B. Russo, and R. Accorsi. Cham: Springer International Publishing, 2015, pages 1–31. ISBN: 978-3-319-27243-6 (cited on page 408).
- [119] M. Leemans and W. M. P. van der Aalst. “Process mining in software systems: Discovering real-life business transactions and process models from distributed systems”. In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Sept. 2015, pages 44–53. DOI: 10.1109/MODELS.2015.7338234 (cited on pages 95, 96, 150, 196, 293, 307, 309, 408).
- [120] M. Leemans and W. M. P. van der Aalst. “Modeling and Discovering Cancellation Behavior”. In: *On the Move to Meaningful Internet Systems. OTM 2017 Conferences: Conf. Int. Conf.: CoopIS*. Edited by H. Panetto, C. Debruyne, W. Gaaloul, et al. Springer International Publishing, 2017, pages 93–113. ISBN: 978-3-319-69462-7. DOI: 10.1007/978-3-319-69462-7\_8 (cited on page 408).
- [121] M. Leemans, W. M. P. van der Aalst, and M. G. J. van den Brand. “Hierarchical Performance Analysis for Process Mining”. In: *Proceedings of the 2018 International Conference on Software and System Process. ICSSP '18*. Gothenburg, Sweden: ACM, 2018, pages 96–105. ISBN: 978-1-4503-6459-1. DOI: 10.1145/3202710.3203151 (cited on page 407).
- [122] M. Leemans, W. M. P. van der Aalst, and M. G. J. van den Brand. “Recursion aware modeling and discovery for hierarchical software event log analysis”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Mar. 2018, pages 185–196. DOI: 10.1109/SANER.2018.8330208 (cited on page 408).
- [123] M. Leemans, W. M. P. van der Aalst, and M. G. J. van den Brand. “The Statechart Workbench: Enabling scalable software event log analysis using process mining”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Mar. 2018, pages 502–506. DOI: 10.1109/SANER.2018.8330248 (cited on page 408).
- [124] M. Leemans, W. M. P. van der Aalst, M. G. J. van den Brand, et al. “Software Process Analysis Methodology: A Methodology based on Lessons Learned in Embracing Legacy Software”. In: *Software Maintenance and Evolution (ICSME), 2018 IEEE 34th International Conference on*. To be published. Sept. 2018 (cited on pages 348, 407).
- [125] M. Leemans, J. F. Groote, K. Kotterink, et al. *Multi-layer System Modelling and Verification of Fine Wafer Alignment (ASML Graduation Project)*. Technical report. Confidential, public version available. ASML, Aug. 2014. URL: <https://research.tue.nl/files/46998509/784676-1.pdf> (cited on page 409).
- [126] M. Leemans, R. P. J. Koolen, S. Cranen, et al. *Analyse van Besturingssystemen voor Beweegbare Bruggen (Analysis of the Control System for Movable Bridges)*. Technical report. Confidential. Rijkswaterstaat, May 2014 (cited on page 409).

- 
- [127] M. Leemans and C. Liu. *XES Software Communication Extension*. Technical report. IEEE CIS Task Force on Process Mining, Nov. 2017. URL: <http://www.win.tue.nl/ieetfpm/lib/exe/fetch.php?media=shared:downloads:2017-06-22-xes-software-communication-v5-2.pdf> (cited on pages 102, 409).
- [128] M. Leemans and C. Liu. *XES Software Event Extension*. Technical report. IEEE CIS Task Force on Process Mining, Sept. 2017. URL: <http://www.win.tue.nl/ieetfpm/lib/exe/fetch.php?media=shared:downloads:2017-06-22-xes-software-event-v5-2.pdf> (cited on pages 102, 409).
- [129] M. Leemans and C. Liu. *XES Software Telemetry Extension*. Technical report. IEEE CIS Task Force on Process Mining, Nov. 2017. URL: <http://www.win.tue.nl/ieetfpm/lib/exe/fetch.php?media=shared:downloads:2017-06-22-xes-software-telemetry-v5-2.pdf> (cited on pages 102, 408).
- [130] S. J. J. Leemans. “Robust process mining with guarantees”. PhD thesis. Eindhoven University of Technology, May 2017. ISBN: 978-90-386-4257-4. URL: <http://repository.tue.nl/864191> (cited on pages 16, 29, 49, 52, 58, 59, 62, 63, 66, 67, 71, 73, 74, 81, 87, 141, 146, 154–157, 190, 194, 199–202, 209, 210, 289, 365, 367, 368).
- [131] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst. “Discovering Block-Structured Process Models from Event Logs - A Constructive Approach”. In: *Application and Theory of Petri Nets and Concurrency*. Edited by J. M. Colom and J. Desel. Springer Berlin Heidelberg, 2013, pages 311–329. DOI: 10.1007/978-3-642-38697-8\_17. URL: [http://link.springer.com/10.1007/978-3-642-38697-8%7B%5C\\_%7D17](http://link.springer.com/10.1007/978-3-642-38697-8%7B%5C_%7D17) (cited on pages 74, 365–367).
- [132] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst. “Exploring Processes and Deviations”. In: *Business Process Management Workshops: BPM 2014 International Workshops, Eindhoven, The Netherlands, September 7-8, 2014, Revised Papers*. Edited by F. Fournier and J. Mendling. Springer International Publishing, 2015, pages 304–316. ISBN: 978-3-319-15895-2. DOI: 10.1007/978-3-319-15895-2\_26. URL: [https://doi.org/10.1007/978-3-319-15895-2\\_26](https://doi.org/10.1007/978-3-319-15895-2_26) (cited on page 279).
- [133] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst. “Scalable process discovery and conformance checking”. In: *Software & Systems Modeling* (July 2016). ISSN: 1619-1374. DOI: 10.1007/s10270-016-0545-x. URL: <https://doi.org/10.1007/s10270-016-0545-x> (cited on page 209).
- [134] S. J. J. Leemans, Dirk Fahland, and W. M. P. van der Aalst. “Scalable process discovery and conformance checking”. In: *Software & Systems Modeling* (July 2016). ISSN: 1619-1374. DOI: 10.1007/s10270-016-0545-x. URL: <https://doi.org/10.1007/s10270-016-0545-x> (cited on pages 67, 150, 196).
- [135] T. Lei, R. Barzilay, and T. S. Jaakkola. “Rationalizing Neural Predictions”. In: *CoRR* abs/1606.04155 (2016). arXiv: 1606.04155. URL: <http://arxiv.org/abs/1606.04155> (cited on page 362).
- [136] LLC YourKit. *The YourKit Profiler*. <https://www.yourkit.com/>. [Online, accessed 12 Januari 2018]. 2003 (cited on pages 56, 59, 60, 69, 70, 93, 239, 240, 245, 246, 293).
- [137] X. Lu, D. Fahland, F. J. H. M. van den Biggelaar, et al. “Handling Duplicated Tasks in Process Discovery by Refining Event Labels”. In: *Business Process Management*. Edited by M. La Rosa, P. Loos, and O. Pastor. Cham: Springer International Publishing, 2016, pages 90–107. ISBN: 978-3-319-45348-4 (cited on page 64).

- [138] F. Mannhardt, M. de Leoni, and H. A. Reijers. “Heuristic Mining Revamped: an Interactive, Data-aware, and Conformance-aware Miner”. In: *Proceedings of the BPM Demo Track and BPM Dissertation Award*. Edited by R. Clarisó, H. Leopold, J. Mendling, et al. Volume 1920. CEUR Workshop Proceedings, 2017. URL: [http://ceur-ws.org/Vol-1920/BPM\\_2017\\_paper\\_167.pdf](http://ceur-ws.org/Vol-1920/BPM_2017_paper_167.pdf) (cited on page 279).
- [139] F. Mannhardt, M. de Leoni, H. A. Reijers, et al. “Guided Process Discovery: A Pattern-based Approach”. In: *Information Systems* (2018) (cited on pages 14, 59, 62, 63, 66, 68, 109).
- [140] M. A. Marsan, G. Balbo, A. Bobbio, et al. “The effect of execution policies on the semantics and analysis of stochastic Petri nets”. In: *IEEE Transactions on Software Engineering* 15.7 (July 1989), pages 832–846. ISSN: 0098-5589. DOI: 10.1109/32.29483 (cited on page 221).
- [141] K. S. McKinley. *Twenty Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 2004 (cited on page 60).
- [142] D. Mills, J. Martin, J. Burbank, et al. “Network time protocol version 4: Protocol and algorithms specification”. In: *IETF RFC5905* (June 2010). URL: <http://tools.ietf.org/html/rfc5905> (cited on page 319).
- [143] P. Mukala, J. Buijs, M. Leemans, et al. “Learning analytics on coursera event data: a process mining approach”. English. In: *Proceedings of the 5th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2015), Vienna, Austria, December 9-11, 2015*. Edited by P. Caravolo and S. Rinderle-Ma. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pages 18–32 (cited on page 408).
- [144] J. Munoz-Gama, J. Carmona, and W. M. P. van der Aalst. “Conformance Checking in the Large: Partitioning and Topology”. In: *Business Process Management: 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings*. Edited by F. Daniel, J. Wang, and B. Weber. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pages 130–145. ISBN: 978-3-642-40176-3. DOI: 10.1007/978-3-642-40176-3\_11. URL: [https://doi.org/10.1007/978-3-642-40176-3\\_11](https://doi.org/10.1007/978-3-642-40176-3_11) (cited on pages 66, 67).
- [145] J. Munoz-Gama, J. Carmona, and W. M. P. van der Aalst. “Single-Entry Single-Exit decomposed conformance checking”. In: *Information Systems* 46.Supplement C (2014), pages 102–122. ISSN: 0306-4379. DOI: <https://doi.org/10.1016/j.is.2014.04.003>. URL: <http://www.sciencedirect.com/science/article/pii/S0306437914000696> (cited on pages 66, 67, 209, 210).
- [146] T. Murata. “Petri nets: Properties, analysis and applications”. In: *Proceedings of the IEEE* 77.4 (Apr. 1989), pages 541–580. ISSN: 0018-9219. DOI: 10.1109/5.24143 (cited on page 25).
- [147] J. Nakatumba and W. M. P. van der Aalst. “Analyzing Resource Behavior Using Process Mining”. In: *Business Process Management Workshops*. Edited by S. Rinderle-Ma, S. Sadiq, and F. Leymann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 69–80. ISBN: 978-3-642-12186-9 (cited on page 211).
- [148] NASA. *JPF Statechart and CEV example*. <http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-statechart>. [Online, accessed 3 March 2017] (cited on pages 97, 151, 196).
- [149] Netcraft. *March 2018 Web Server Survey*. <https://news.netcraft.com/archives/2018/03/27/march-2018-web-server-survey.html>. [Online, accessed 25 Juli 2018] (cited on page 91).

- 
- [150] C. G. Nevill-Manning and I. H. Witten. “Identifying Hierarchical Structure in Sequences: A Linear-time Algorithm”. In: *Journal of Artificial Intelligence Research* 7.1 (Sept. 1997), pages 67–82. ISSN: 1076-9757. URL: <http://dl.acm.org/citation.cfm?id=1622776.1622780> (cited on pages 55, 59, 61, 62).
- [151] E. H. J. Nooijen, B. F. van Dongen, and D. Fahland. “Automatic Discovery of Data-Centric and Artifact-Centric Processes”. In: *Business Process Management Workshops*. Edited by M. La Rosa and P. Soffer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pages 316–327. ISBN: 978-3-642-36285-9 (cited on pages 99, 101).
- [152] R. Oechsle and T. Schmitt. “JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)”. English. In: *Software Visualization*. Edited by S. Diehl. Volume 2269. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pages 176–190. ISBN: 978-3-540-43323-1. DOI: 10.1007/3-540-45875-1\_14. URL: [http://dx.doi.org/10.1007/3-540-45875-1\\_14](http://dx.doi.org/10.1007/3-540-45875-1_14) (cited on pages 59, 61).
- [153] OMG. “Business Process Modeling Notation 2.0”. In: *OMG Final Adopted Specification, Object Management Group* (2011). formal/2011-01-03. URL: <http://www.omg.org/spec/BPMN/2.0/PDF> (cited on page 34).
- [154] OMG. “Unified Modeling Language (UML) 2.5”. In: *OMG Final Adopted Specification, Object Management Group* (2015). formal/2015-03-01. URL: <http://www.omg.org/spec/UML/2.5/> (cited on pages 35, 37, 267).
- [155] Oracle. *The Java™ Tutorials – Trail: The Reflection API*. <https://docs.oracle.com/javase/tutorial/reflect/index.html>. [Online, accessed 24 July 2018] (cited on page 330).
- [156] J. L. Peterson. “Petri Nets”. In: *ACM Computing Surveys* 9.3 (Sept. 1977), pages 223–252. ISSN: 0360-0300. DOI: 10.1145/356698.356702. URL: <http://doi.acm.org/10.1145/356698.356702> (cited on page 31).
- [157] D. A. M. Piessens, M. T. Wynn, M. J. Adams, et al. “Performance analysis of business process models with advanced constructs”. In: *21st Australasian Conference on Information Systems (ACIS 2010) :Information Systems : Defining and Establishing a High Impact Discipline*. Brisbane, Qld., Dec. 2010. URL: <https://eprints.qut.edu.au/43356/> (cited on page 70).
- [158] A. Polyvyanyy, J. Vanhatalo, and H. Völzer. “Simplified Computation and Generalization of the Refined Process Structure Tree”. In: *Web Services and Formal Methods: 7th International Workshop, WS-FM 2010, Hoboken, NJ, USA, September 16-17, 2010. Revised Selected Papers*. Edited by M. Bravetti and T. Bultan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pages 25–41. ISBN: 978-3-642-19589-1. DOI: 10.1007/978-3-642-19589-1\_2. URL: [https://doi.org/10.1007/978-3-642-19589-1\\_2](https://doi.org/10.1007/978-3-642-19589-1_2) (cited on pages 66, 67, 210).
- [159] Process Mining Group, Eindhoven University of Technology. *ProM 6.8*. <http://www.promtools.org/doku.php?id=prom68>. [Online, accessed 11 July 2018] (cited on page 278).
- [160] Process Mining Group, Eindhoven University of Technology. *RapidProM*. <http://www.promtools.org/doku.php?id=rapidprom:home> (cited on page 321).
- [161] RapidMiner, Inc. *RapidMiner*. <https://rapidminer.com/> (cited on page 320).

- [162] D. Redlich, T. Molka, W. Gilani, et al. “Constructs Competition Miner: Process Control-Flow Discovery of BP-Domain Constructs”. In: *Business Process Management: 12th International Conference, BPM 2014, Haifa, Israel, September 7-11, 2014. Proceedings*. Edited by S. Sadiq, P. Soffer, and H. Völzer. Cham: Springer International Publishing, 2014, pages 134–150. ISBN: 978-3-319-10172-9. DOI: 10.1007/978-3-319-10172-9\_9. URL: [http://dx.doi.org/10.1007/978-3-319-10172-9\\_9](http://dx.doi.org/10.1007/978-3-319-10172-9_9) (cited on pages 59, 62).
- [163] W. Reisig. *A Primer in Petri Net Design*. Springer Compass International. Springer Berlin Heidelberg, 1992. ISBN: 978-3-642-75329-9. DOI: 10.1007/978-3-642-75329-9 (cited on page 25).
- [164] New Relic. *New Relic APM*. <https://newrelic.com/application-monitoring>. [Online, accessed 2 Februari 2018] (cited on pages 56, 92, 95).
- [165] A. Rogge-Solti, W. M. P. van der Aalst, and M. Weske. “Discovering Stochastic Petri Nets with Arbitrary Delay Distributions from Event Logs”. In: *Business Process Management Workshops: BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers*. Edited by N. Lohmann, M. Song, and P. Wohed. Cham: Springer International Publishing, 2014, pages 15–27. ISBN: 978-3-319-06257-0. DOI: 10.1007/978-3-319-06257-0\_2. URL: [https://doi.org/10.1007/978-3-319-06257-0\\_2](https://doi.org/10.1007/978-3-319-06257-0_2) (cited on pages 70, 71, 221).
- [166] A. Rountev and B. H. Connell. “Object Naming Analysis for Reverse-engineered Sequence Diagrams”. In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: ACM, 2005, pages 254–263. ISBN: 1-58113-963-2. DOI: 10.1145/1062455.1062510. URL: <http://doi.acm.org/10.1145/1062455.1062510> (cited on pages 56, 58–60).
- [167] W. Samek, A. Binder, G. Montavon, et al. “Evaluating the Visualization of What a Deep Neural Network Has Learned”. In: *IEEE Transactions on Neural Networks and Learning Systems* 28.11 (Nov. 2017), pages 2660–2673. ISSN: 2162-237X. DOI: 10.1109/TNNLS.2016.2599820 (cited on page 362).
- [168] W. Schutz. “On the testability of distributed real-time systems”. In: *Reliable Distributed Systems, 1991. Proceedings., Tenth Symposium on*. IEEE. Sept. 1991, pages 52–61. DOI: 10.1109/RELDIS.1991.145404 (cited on pages 93, 316).
- [169] A. Senderovich, S. J. J. Leemans, S. Harel, et al. “Discovering Queues from Event Logs with Varying Levels of Information”. In: *Business Process Management Workshops*. Edited by M. Reichert and H. A. Reijers. Cham: Springer International Publishing, 2016, pages 154–166. ISBN: 978-3-319-42887-1 (cited on page 256).
- [170] B. H. Sigelman, L. A. Barroso, M. Burrows, et al. *Dapper, a large-scale distributed systems tracing infrastructure*. Technical report. April: Google, Inc., 2010 (cited on pages 92, 95, 101).
- [171] P. Siyari, B. Dilkina, and C. Dovrolis. “Lexis: An Optimization Framework for Discovering the Hierarchical Structure of Sequential Data”. In: *Gecco* (Feb. 2016), pages 421–434. ISSN: 0146-4833. DOI: 10.1145/2939672.2939741. arXiv: 1602.05561. URL: <http://dx.doi.org/10.1145/2939672.2939741> (cited on pages 59, 61, 62).
- [172] M. S. Song and W. M. P. van der Aalst. “Supporting process mining by showing events at a glance”. In: *Workshop on Information Technologies and Systems (WITS), 17th Annual Workshop on*. 2007, pages 139–145 (cited on pages 70, 239, 240, 246–248, 309, 314, 325, 326, 338).

- 
- [173] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. “AspectC++: An Aspect-oriented Extension to the C++ Programming Language”. In: *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*. CRPIT '02. Sydney, Australia: Australian Computer Society, Inc., 2002, pages 53–60. ISBN: 0-909925-88-7. URL: <http://dl.acm.org/citation.cfm?id=564092.564100> (cited on page 93).
- [174] W. Steeman. *BPI Challenge 2013, incidents*. 2013. DOI: 10.4121/uuid:500573e6-acc-4b0c-9576-aa5468b10cee. URL: <http://dx.doi.org/10.4121/uuid:500573e6-acc-4b0c-9576-aa5468b10cee> (cited on pages 96, 97, 150–152, 196).
- [175] Sun Microsystems. *UNIX Snoop*. [https://docs.oracle.com/cd/E23824\\_01/html/821-1453/gexkw.html](https://docs.oracle.com/cd/E23824_01/html/821-1453/gexkw.html). [Online, accessed 25 Juli 2018] (cited on page 92).
- [176] T. Systä, K. Koskimies, and H. Müller. “Shimba – an environment for reverse engineering Java software systems”. In: *Software: Practice and Experience* 31.4 (2001), pages 371–394. ISSN: 1097-024X. DOI: 10.1002/spe.386. URL: <http://dx.doi.org/10.1002/spe.386> (cited on pages 59, 61).
- [177] M. Szvetits and U. Zdun. “Reusable event types for models at runtime to support the examination of runtime phenomena”. In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Sept. 2015, pages 4–13. DOI: 10.1109/MODELS.2015.7338230 (cited on pages 59, 70).
- [178] N. Tax, N. Sidorova, and W. M. P. van der Aalst. “Discovering More Precise Process Models from Event Logs by Filtering Out Chaotic Activities”. In: *arXiv preprint arXiv:1711.01287* (2017). URL: <https://arxiv.org/abs/1711.01287> (cited on page 174).
- [179] The Wireshark team and Gerald Combs. *Wireshark*. <https://www.wireshark.org/>. [Online, accessed 25 Juli 2018] (cited on page 92).
- [180] K. Thompson and D. M. Ritchie. *Unix Programmer’s Manual, Fourth Edition*. Bell Telephone Laboratories, Incorporated, Nov. 1973 (cited on page 60).
- [181] P. Tonella and A. Potrich. “Reverse engineering of the interaction diagrams from C++ code”. In: *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. Sept. 2003, pages 159–168. DOI: 10.1109/ICSM.2003.1235418 (cited on pages 58, 59).
- [182] Zero Turnaround. *XRebel*. <https://zeroturnaround.com/software/xrebel/>. [Online, accessed 2 Februari 2018] (cited on pages 56, 92, 95).
- [183] F. Vaandrager. “Model Learning”. In: *Commun. ACM* 60.2 (Jan. 2017), pages 86–95. ISSN: 0001-0782. DOI: 10.1145/2967606. URL: <http://doi.acm.org/10.1145/2967606> (cited on pages 55–57, 60, 61).
- [184] D. Varró. “A Formal Semantics of UML Statecharts by Model Transition Systems”. In: *Graph Transformation: First International Conference, ICGT 2002 Barcelona, Spain, October 7–12, 2002 Proceedings*. Edited by A. Corradini, H. Ehrig, H. J. Kreowski, et al. Volume 2505. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pages 378–392. ISBN: 978-3-540-45832-6. DOI: 10.1007/3-540-45832-8\_28. URL: [http://dx.doi.org/10.1007/3-540-45832-8\\_28](http://dx.doi.org/10.1007/3-540-45832-8_28) (cited on page 35).
- [185] H. M. W. Verbeek. *ProM 6.7 Release Notes*. <https://svn.win.tue.nl/trac/prom/wiki/ProM67/ReleaseNotes>. [Online, accessed 25 July 2018] (cited on page 92).

- [186] H. M. W. Verbeek and W. M. P. van der Aalst. “Decomposed Process Mining: The ILP Case”. In: *Business Process Management Workshops: BPM 2014 International Workshops, Eindhoven, The Netherlands, September 7-8, 2014, Revised Papers*. Edited by F. Fournier and J. Mendling. Cham: Springer International Publishing, 2015, pages 264–276. ISBN: 978-3-319-15895-2. DOI: 10.1007/978-3-319-15895-2\_23. URL: [https://doi.org/10.1007/978-3-319-15895-2\\_23](https://doi.org/10.1007/978-3-319-15895-2_23) (cited on pages 66, 67, 209).
- [187] H. M. W. Verbeek, J. C. A. M. Buijs, B. F. van Dongen, et al. “XES, XESame, and ProM 6”. English. In: *Information Systems Evolution*. Edited by P. Soffer and E. Proper. Volume 72. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2011, pages 60–75. ISBN: 978-3-642-17721-7. DOI: 10.1007/978-3-642-17722-4\_5. URL: [http://dx.doi.org/10.1007/978-3-642-17722-4\\_5](http://dx.doi.org/10.1007/978-3-642-17722-4_5) (cited on pages 16, 46, 47, 97, 151, 197, 279, 353, 356).
- [188] M. Volpato and J. Tretmans. “Approximate Active Learning of Nondeterministic Input Output Transition Systems”. In: *Electronic Communications of the EASST 72 (2015)*. DOI: <http://dx.doi.org/10.14279/tuj.eceasst.72.1008> (cited on pages 59, 61, 64, 65).
- [189] W3Techs. *Usage of traffic analysis tools for websites*. [Online, accessed 25 Juli 2018]. URL: [https://w3techs.com/technologies/overview/traffic\\_analysis/all](https://w3techs.com/technologies/overview/traffic_analysis/all) (cited on page 92).
- [190] N. Walkinshaw, R. Taylor, and J. Derrick. “Inferring extended finite state machine models from software executions”. In: *Empirical Software Engineering 21.3 (2016)*, pages 811–853. ISSN: 1573-7616. DOI: 10.1007/s10664-015-9367-7. URL: <http://dx.doi.org/10.1007/s10664-015-9367-7> (cited on pages 55, 59, 61, 64, 65, 140, 145, 146, 154–157, 189, 193, 199–202).
- [191] J. Weidendorfer. “Sequential Performance Analysis with Callgrind and KCachegrind”. In: *Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRs, Stuttgart*. Edited by M. Resch, R. Keller, V. Himmler, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 93–113. ISBN: 978-3-540-68564-7. DOI: 10.1007/978-3-540-68564-7\_7. URL: [https://doi.org/10.1007/978-3-540-68564-7\\_7](https://doi.org/10.1007/978-3-540-68564-7_7) (cited on pages 56, 59, 60, 69, 70).
- [192] A. J. M. M. Weijters and J. T. S. Ribeiro. “Flexible Heuristics Miner (FHM)”. In: *2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*. Apr. 2011, pages 310–317. DOI: 10.1109/CIDM.2011.5949453 (cited on pages 52, 59, 62, 63, 140, 142, 154–157, 189, 192, 199–202).
- [193] J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, et al. “Process Discovery Using Integer Linear Programming”. In: *Applications and Theory of Petri Nets: 29th International Conference, PETRI NETS 2008, Xi’an, China, June 23-27, 2008. Proceedings*. Edited by K. M. van Hee and R. Valk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 368–387. ISBN: 978-3-540-68746-7. DOI: 10.1007/978-3-540-68746-7\_24. URL: [http://dx.doi.org/10.1007/978-3-540-68746-7\\_24](http://dx.doi.org/10.1007/978-3-540-68746-7_24) (cited on pages 59, 62, 63, 154–157, 199–202).
- [194] R. Wester and J. Koster. “The Software behind Moore’s Law”. In: *IEEE Software 32.2 (Mar. 2015)*, pages 37–40. ISSN: 0740-7459. DOI: 10.1109/MS.2015.53 (cited on page 335).

- [195] R. Wieman, M. F. Aniche, W. Lobbezoo, et al. “An Experience Report on Applying Passive Learning in a Large-Scale Payment Company”. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Sept. 2017, pages 564–573. DOI: 10.1109/ICSME.2017.71 (cited on page 92).
- [196] Wikipedia. *Mealy machine*. [https://en.wikipedia.org/wiki/Mealy\\_machine](https://en.wikipedia.org/wiki/Mealy_machine). [Online, accessed 7 Februari 2018]. 2017 (cited on page 61).
- [197] Wikipedia. *Profiling (computer programming)*. [https://en.wikipedia.org/wiki/Profiling\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Profiling_(computer_programming)). [Online, accessed 2 Februari 2018]. 2017 (cited on page 60).
- [198] R. Wirth and J. Hipp. “CRISP-DM: Towards a standard process model for data mining”. In: *Proceedings of the Fourth International Conference on the Practical Application of Knowledge Discovery and Data Mining*. 2000, pages 29–39 (cited on pages 304, 316, 317).
- [199] M. T. Wynn, W. M. P. van der Aalst, A. H. M. ter Hofstede, et al. “Verifying Workflows with Cancellation Regions and OR-Joins: An Approach Based on Reset Nets and Reachability Analysis”. In: *Business Process Management*. Edited by S. Dustdar, J. L. Fiadeiro, and A. P. Sheth. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pages 389–394. ISBN: 978-3-540-38903-3 (cited on page 34).
- [200] Z. Yao, Q. Zheng, and G. Chen. “AOP++: A Generic Aspect-Oriented Programming Framework in C++”. English. In: *Generative Programming and Component Engineering*. Edited by R. Glück and M. Lowry. Volume 3676. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pages 94–108. ISBN: 978-3-540-29138-1. DOI: 10.1007/11561347\_8. URL: [http://dx.doi.org/10.1007/11561347\\_8](http://dx.doi.org/10.1007/11561347_8) (cited on page 93).
- [201] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. Recommendation Z.120. Geneva: International Telecommunication Union – Telecommunication Standardization Sector, Feb. 2011. URL: <https://www.itu.int/rec/T-REC-Z.120-201102-I/en> (cited on page 37).
- [202] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC) – Annex B: Algebraic Semantics of Message Sequence Charts*. Recommendation Z.120 Annex B. Geneva: International Telecommunication Union – Telecommunication Standardization Sector, Apr. 1998. URL: <https://www.itu.int/rec/T-REC-Z.120-199804-I!AnnB/en> (cited on page 37).
- [203] S. J. Zelst, van, A. Burattin, B. F. Dongen, van, et al. “Data streams in ProM 6 : a single-node architecture”. English. In: *BPM Demo Sessions 2014 (co-located with BPM 2014, Eindhoven, The Netherlands, September 20, 2014)*. Edited by L. Limonad and B. Weber. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pages 81–85 (cited on page 293).
- [204] S. J. van Zelst, B. F. van Dongen, and W. M. P. van der Aalst. “Avoiding Over-Fitting in ILP-Based Process Discovery”. In: *Business Process Management: 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 – September 3, 2015, Proceedings*. Edited by H. R. Motahari-Nezhad, J. Recker, and M. Weidlich. Cham: Springer International Publishing, 2015, pages 163–171. ISBN: 978-3-319-23063-4. DOI: 10.1007/978-3-319-23063-4\_10. URL: [http://dx.doi.org/10.1007/978-3-319-23063-4\\_10](http://dx.doi.org/10.1007/978-3-319-23063-4_10) (cited on pages 59, 62, 63, 140, 142, 154–157, 189, 192, 199–202).



- [205] S. J. van Zelst, B. F. van Dongen, and W. M. P. van der Aalst. “Event stream-based process discovery using abstract representations”. In: *Knowledge and Information Systems* 54.2 (Feb. 2018), pages 407–435. ISSN: 0219-3116. DOI: 10.1007/s10115-017-1060-2. URL: <https://doi.org/10.1007/s10115-017-1060-2> (cited on page 293).
- [206] S. J. van Zelst, B. F. van Dongen, W. M. P. van der Aalst, et al. “Discovering Relaxed Sound Workflow Nets using Integer Linear Programming”. In: *CoRR* abs/1703.06733 (2017). arXiv: 1703.06733. URL: <http://arxiv.org/abs/1703.06733> (cited on pages 59, 62).

# Summary

## Hierarchical Process Mining for Scalable Software Analysis

In today's world, we increasingly rely on information technology. Complex software-driven systems are supporting and automating all kinds of tasks and such systems can be found in all sectors. In this thesis, we aim to understand and analyze such software systems using process mining techniques. That is, we see such software systems as complex processes which we can analyze using recorded observations known as execution data. The idea is that we can use such process execution data to gain insights about the real execution of these processes. So far, process mining techniques have had great success in understanding and analyzing organizational/business processes, but there has been little work on using process mining on software systems.

In this thesis, we addressed research challenges in using process mining techniques for analyzing software. First of all, we looked into obtaining execution data from running software. How can we record such data, what does this data look like, and what are the various properties of such data? Next, we looked into using such execution data for process discovery. In process discovery, we automatically discover a process model from the recorded execution data. Such a discovered process model explains what tasks or activities actually happened in which order in the observed/recorded process. In addition, we investigated how we can use such discovered models together with the recorded execution data to analyze various additional perspectives. We looked at which parts of the software are executed more frequently, where various bottlenecks or other performance issues are, and more. Finally, we investigated various ways of showing such models and results to end users.

To summarize, this thesis presents the following contributions:

- We provided a detailed discussion of software execution data, how to use such data for process mining, as well as tool support for obtaining and recording such data.
- We acknowledged that, in software, reality is not flat. Typically, the software behavior is large, complex, and contains some form of hierarchical or layered structure. Therefore, we developed a modeling notation and

discovery techniques for hierarchical and recursive behavior, exploiting hierarchical models.

- In addition, we accept that software can produce errors and this needs to be analyzed. Therefore, we developed a modeling notation and discovery techniques for cancelation or error-handling behavior.
- With the more complex models introduced by the above discovery techniques, we needed to revisit the way we can perform frequencies and performance analysis. Therefore, we introduced a framework for visualization-independent performance analysis, supporting hierarchy, recursion, and cancelation.
- In the software domain, there are various ways to model software behavior, and there is not one correct standard or solution for all use cases. Therefore, we developed a family of model translations, supporting hierarchy, recursion, and cancelation. This way, we can show results to end users in the type of model they prefer.
- To make the ideas and theories in this thesis useable, we developed and made available extensive tool support for round-trip software analysis.

All methods have been implemented in various tools (in the context of ProM and XES), which are publicly available with documentation. In addition, all techniques have been systematically evaluated and have been applied in real-life situations in the context of several case studies.

# Samenvatting

## Hiërarchische Process Mining voor Schaalbare Software Analyse

In de hedendaagse wereld zijn we steeds meer afhankelijk van informatie technologie. Complexe software-gedreven systemen ondersteunen en automatiseren verscheidende taken en zulke systemen zijn te vinden in alle sectoren. In deze dissertatie streven we ernaar om zulke softwaresystemen te begrijpen en te analyseren door middel van process mining technieken. Dat is, we zien zulke softwaresystemen als complexe processen die we kunnen analyseren met behulp van vastgelegde observaties, ook wel executie data genoemd. Het idee is dat we zulke procesexecutie data kunnen gebruiken om inzicht te creëren in de daadwerkelijke executie van dergelijke processen. Tot nu toe hebben process mining technieken veel succes geboekt in het begrijpen en analyseren van organisatorische/bedrijfsprocessen, maar is er weinig gekeken naar hoe process mining kan ingezet worden voor software systemen.

In deze dissertatie behandelen we onderzoeksvragen over het gebruik van process mining voor software analyse. Ten eerste hebben we gekeken naar hoe men executie data kan vergaren van draaiende software. Hoe kunnen we zulke data opslaan, hoe ziet zulke data eruit, en wat zijn de eigenschappen van dergelijke datasets? Daarnaast hebben we gekeken naar hoe we zulke executie data kunnen gebruiken voor process discovery (proces ontdekken). In process discovery gebruiken we opgeslagen executie data om automatisch proces modellen te ontdekken. Een dergelijk ontdekt model verklaart welke taken of activiteiten daadwerkelijk gebeuren in het geobserveerde proces, en in welke volgorde deze activiteiten plaats vonden. Verder hebben we onderzocht hoe we zulke ontdekte modellen en executie data kunnen combineren voor het analyseren van diverse perspectieven. We hebben gekeken naar welke onderdelen van de software vaker worden uitgevoerd, waar diverse knelpunten en andere prestatieproblemen zitten, en meer. Tot slot hebben we diverse manieren onderzocht hoe we zulke data en modellen aan eindgebruikers kunnen tonen.

Samenvattend, deze dissertatie beschrijft de volgende bijdragen:

- We geven een gedetailleerde discussie van software executie data, hoe we zulke data kunnen gebruiken voor process mining, en we leveren tool-

ing/software voor het observeren en opslaan van zulke data.

- We erkennen dat, in software, de realiteit niet plat is. Typisch voor software is groot en complex gedrag met een vorm van hiërarchie of een gelaagde structuur. Daarom hebben we een model notatie en discovery techniek ontwikkeld die hiërarchisch en recursief gedrag ondersteunt, gebruik makend van gelaagde modellen.
- Daarnaast erkennen we dat software ook fouten kan produceren. Daarom hebben we model notatie en discovery techniek ontwikkeld welke annullering en fout afhandelend gedrag ondersteunt.
- Met de meer complexe modellen geïntroduceerd door de bovenstaande technieken waren we genoodzaakt om de manier van frequentie en prestatie analyse te herzien. Daarom introduceren we een raamwerk voor visualisatie-onafhankelijke prestatie analyse, die ondersteuning biedt voor hiërarchisch, recursief, en annulerend gedrag.
- Om de ideeën en theorieën uit deze dissertatie toepasbaar te maken hebben we uitgebreide tooling/software beschikbaar gesteld voor complete software analyse, van begin tot eind.

Alle methodieken zijn geïmplementeerd in diverse tooling/software, welke publiekelijk beschikbaar zijn met documentatie. Daarnaast zijn alle technieken systematisch geëvalueerd en toegepast in praktische situaties via diverse case studies.

# Acknowledgments

For me, these last four years of pursuing a PhD were life-changing. Both within and during my PhD project, I have been supported, shaped, and affected by wonderful people that joined me for (part of) the ride.

First of all, I thank my first promotor, Wil van der Aalst, for seeing my potential as a PhD researcher before I did. Wil, thank you for inviting me on this PhD journey, for guiding me during these last four years, and for the many interesting collaborations. It was a true pleasure to grow in this researcher role under your expert guidance. I also thank my second promotor, Mark van den Brand, for enriching my PhD career. Mark, thank you for bringing a fresh perspective to this PhD journey and for the many refreshing talks and insights. Thanks to the contrasting and complementary expertises and knowledge of both promotors, together with the many interesting discussions, I was able to grow, diversify, and broaden my horizon.

Secondly, I thank my external PhD committee members, Jack van Wijk, Arie van Deursen, Serge Demeyer, and Alistair Barros, for their time to read this thesis and to take the effort required to travel to Eindhoven.

Next, I would like to thank our wonderful and caring secretaries, Ine van der Ligt, Riet van Buul, and José Jong. Without their help and support, both professionally and personally, this PhD would have been a lot tougher. In addition, special thanks to Ine for taking the time to proof read this thesis.

Furthermore, I also would like to thank my colleagues. I thank the people I got to share an office with at some point during my PhD for all the humor, jokes, and interesting discussions, thank you Elham, Dennis, Joos, Guangming, Marie, Felix, Eduardo, Bart, Nour, and Marcus. Thank you Xixi for convincing me that doing a PhD is fun and for all the chats we had. Special thanks to Bram, Bram, and Josh for sharing the joy and burdens both during our studies and during our PhD adventures. And of course a thank you to all my other colleagues: Alfredo, Alifah, Alok, Boudewijn, Cong, Dirk, Eric, Farideh, George, Hajo, Julia, Maikel van E., Marwan, Massimiliano, Murat, Mykola, Natalia, Natasha, Niek, Paul, Rémi, Renata, Sander, Shiva, and the rest.

During my PhD, I had several interesting collaboration opportunities. I would like to thank Artem, Arthur, Moe, Robert, and the rest for their warm

welcome and hospitality during my visit to the BPM group at QUT Brisbane. Also, I would like to thank the people at ASML for the interesting collaborations, discussions, and hospitality, thank you Dennis, Dennis, Jacques, Jan, Jeroen, Kousar, Leonard, Luna, Ramon, Rolf, Sven, Yuri, and the rest.

Last but not least, I wish to thank my family and friends for their unconditional support and love. Thanks to my mother, father, sister, grandfather, grandmother, aunt, and uncle, for always being there for me, in good and in bad times. Thanks to my friends, most of all Erik, Kelly, Tim, Joep, for their understanding and the many hours of chatting, cooking, playing games, and bringing me a smile. Thanks to Deni and Lee for listening and reminding me of the importance of taking time off. A big and loving thank you to Nick for warming my hearth and showing me what's really important in life. And most of all, thank you dear mom, for always believing in me, till the end and beyond.

Maikel Leemans  
Eindhoven, August 2018

# Curriculum Vitae

Maikel Leemans was born on August 7, 1990 in 's-Hertogenbosch, the Netherlands. He received a Bachelor and a Master of Science in Computer Science and Engineering at Eindhoven University of Technology in Eindhoven, the Netherlands. During his bachelor, he also completed an Honors program, worked part time as a software developer, and did a minor in Connecting Intelligence at the faculty of Electrical Engineering. During his master, he also completed an Honors program, participated in the Solar Team Eindhoven as a software architect, and worked part time as a software developer. He completed his master thesis on the topic of formally modeling and analyzing a multi-layered system in the ASML TWINSKAN, focusing on formal, model-based design and model checking techniques. He graduated from his bachelor and master cum laude.

From 2014 on Maikel started a PhD project in the Department of Mathematics and Computer Science at Eindhoven University of Technology under the supervision of prof.dr.ir. Wil M. P. van der Aalst and prof.dr. Mark G. J. van den Brand. The work of Maikel resulted in a number of publications at international conferences and an international journal, various open-source tools and datasets, and the work presented in this dissertation.

## List of Publications

Maikel Leemans has the following publications:

### Published Articles

- 2018 M. Leemans, W. M. P. van der Aalst, M. G. J. van den Brand, et al. “Software Process Analysis Methodology: A Methodology based on Lessons Learned in Embracing Legacy Software”. In: *Software Maintenance and Evolution (ICSME), 2018 IEEE 34th International Conference on*. To be published. Sept. 2018
- 2018 M. Leemans, W. M. P. van der Aalst, and M. G. J. van den Brand. “Hierarchical Performance Analysis for Process Mining”. In: *Proceedings of the 2018 International Conference on Software and System Process. ICSSP '18*. Gothenburg, Sweden: ACM, 2018, pages 96–105. ISBN: 978-1-4503-6459-1. DOI: [10.1145/3202710.3203151](https://doi.org/10.1145/3202710.3203151)



- 2018 M. Leemans, W. M. P. van der Aalst, and M. G. J. van den Brand. “The Statechart Workbench: Enabling scalable software event log analysis using process mining”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Mar. 2018, pages 502–506. DOI: 10.1109/SANER.2018.8330248
- 2018 M. Leemans, W. M. P. van der Aalst, and M. G. J. van den Brand. “Recursion aware modeling and discovery for hierarchical software event log analysis”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Mar. 2018, pages 185–196. DOI: 10.1109/SANER.2018.8330208
- 2017 M. Leemans and W. M. P. van der Aalst. “Modeling and Discovering Cancellation Behavior”. In: *On the Move to Meaningful Internet Systems. OTM 2017 Conferences: Conf. Int. Conf.: CoopIS*. edited by H. Panetto, C. Debruyne, W. Gaaloul, et al. Springer International Publishing, 2017, pages 93–113. ISBN: 978-3-319-69462-7. DOI: 10.1007/978-3-319-69462-7\_8
- 2015 M. Leemans and W. M. P. van der Aalst. “Process mining in software systems: Discovering real-life business transactions and process models from distributed systems”. In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Sept. 2015, pages 44–53. DOI: 10.1109/MODELS.2015.7338234
- 2015 P. Mukala, J. Buijs, M. Leemans, et al. “Learning analytics on coursera event data: a process mining approach”. English. In: *Proceedings of the 5th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2015), Vienna, Austria, December 9-11, 2015*. Edited by P. Caravolo and S. Rinderle-Ma. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pages 18–32
- 2015 M. Leemans and W. M. P. van der Aalst. “Discovery of Frequent Episodes in Event Logs (Extended)”. In: *Data-Driven Process Discovery and Analysis*. Edited by P. Caravolo, B. Russo, and R. Accorsi. Cham: Springer International Publishing, 2015, pages 1–31. ISBN: 978-3-319-27243-6
- 2014 M. Leemans and W. M. P. van der Aalst. “Discovery of frequent episodes in event logs”. English. In: *4th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2014, Milan, Italy, November 19-21, 2014)*. Edited by R. Accorsi, P. Caravolo, and B. Russo. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pages 31–45

## Technical Reports

- 2017 M. Leemans. *Recursion Aware Modeling and Discovery For Hierarchical Software Event Log Analysis (Extended) - Technical Report version with guarantee proofs for the discovery algorithms*. Technical report. <https://arxiv.org/abs/1710.09323>. Eindhoven University of Technology, 2017
- 2017 M. Leemans and C. Liu. *XES Software Telemetry Extension*. Technical report. IEEE CIS Task Force on Process Mining, Nov. 2017. URL: <http://www.win.tue.nl/ieeetfpm/lib/exe/fetch.php?media=shared:downloads:2017-06-22-xes-software-telemetry-v5-2.pdf>

- 2017 M. Leemans and C. Liu. *XES Software Communication Extension*. Technical report. IEEE CIS Task Force on Process Mining, Nov. 2017. URL: <http://www.win.tue.nl/ieeetfpm/lib/exe/fetch.php?media=shared:downloads:2017-06-22-xes-software-communication-v5-2.pdf>
- 2017 M. Leemans and C. Liu. *XES Software Event Extension*. Technical report. IEEE CIS Task Force on Process Mining, Sept. 2017. URL: <http://www.win.tue.nl/ieeetfpm/lib/exe/fetch.php?media=shared:downloads:2017-06-22-xes-software-event-v5-2.pdf>
- 2014 M. Leemans, J. F. Groote, K. Kotterink, et al. *Multi-layer System Modelling and Verification of Fine Wafer Alignment (ASML Graduation Project)*. Technical report. Confidential, public version available. ASML, Aug. 2014. URL: <https://research.tue.nl/files/46998509/784676-1.pdf>
- 2014 M. Leemans, R. P. J. Koolen, S. Cranen, et al. *Analyse van Besturingssystemen voor Beweegbare Bruggen (Analysis of the Control System for Movable Bridges)*. Technical report. Confidential. Rijkswaterstaat, May 2014

## Software, Media, and Data Sets

- 2018 M. Leemans. *Statechart Workbench and Alignments Software Event Log*. 2018. DOI: 10.4121/uuid:7f787965-da13-4bb8-a3fd-242f08aef9c4. URL: <http://doi.org/10.4121/uuid:7f787965-da13-4bb8-a3fd-242f08aef9c4>
- 2017 M. Leemans. *Statechart Workbench*. <https://youtu.be/xR4XfU3E5mk>
- 2017 M. Leemans. *Statechart plugin for ProM 6*. <https://svn.win.tue.nl/repos/prom/Packages/Statechart/>
- 2017 M. Leemans. *SAW Eclipse plugin for the Statechart Workbench*. <https://svn.win.tue.nl/repos/prom/XPort/>
- 2017 M. Leemans. *NASA Crew Exploration Vehicle (CEV) Software Event Log*. 2017. DOI: 10.4121/uuid:60383406-ffcd-441f-aa5e-4ec763426b76. URL: <http://doi.org/10.4121/uuid:60383406-ffcd-441f-aa5e-4ec763426b76>
- 2017 M. Leemans. *Apache Commons Crypto 1.0.0 - Stream CbcNopad Unit Test Software Event Log*. 2017. DOI: 10.4121/uuid:bb3286d6-dde1-4e74-9a64-fd4e32f10677. URL: <http://doi.org/10.4121/uuid:bb3286d6-dde1-4e74-9a64-fd4e32f10677>
- 2016 M. Leemans. *JUnit 4.12 Software Event Log*. 2016. DOI: 10.4121/uuid:cfed8007-91c8-4b12-98d8-f233e5cd25bb. URL: <http://doi.org/10.4121/uuid:cfed8007-91c8-4b12-98d8-f233e5cd25bb>
- 2015 M. Leemans. *Episode miner plugin for ProM 6*. <https://svn.win.tue.nl/repos/prom/Packages/EpisodeMiner/>. 2015



# SIKS Dissertations

- 
- 2011 01 Botond Cseke (RUN), Variational Algorithms for Bayesian Inference in Latent Gaussian Models
  - 02 Nick Tinnemeier (UU), Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language
  - 03 Jan Martijn van der Werf (TUE), Compositional Design and Verification of Component-Based Information Systems
  - 04 Hado van Hasselt (UU), Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference
  - 05 Bas van der Raadt (VU), Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.
  - 06 Yiwen Wang (TUE), Semantically-Enhanced Recommendations in Cultural Heritage
  - 07 Yujia Cao (UT), Multimodal Information Presentation for High Load Human Computer Interaction
  - 08 Nieske Vergunst (UU), BDI-based Generation of Robust Task-Oriented Dialogues
  - 09 Tim de Jong (OU), Contextualised Mobile Media for Learning
  - 10 Bart Bogaert (UvT), Cloud Content Contention
  - 11 Dhaval Vyas (UT), Designing for Awareness: An Experience-focused HCI Perspective
  - 12 Carmen Bratosin (TUE), Grid Architecture for Distributed Process Mining
  - 13 Xiaoyu Mao (UvT), Airport under Control. Multiagent Scheduling for Airport Ground Handling
  - 14 Milan Lovric (EUR), Behavioral Finance and Agent-Based Artificial Markets
  - 15 Marijn Koolen (UvA), The Meaning of Structure: the Value of Link Evidence for Information Retrieval
  - 16 Maarten Schadd (UM), Selective Search in Games of Different Complexity
  - 17 Jiyin He (UVA), Exploring Topic Structure: Coherence, Diversity and Relatedness
  - 18 Mark Ponsen (UM), Strategic Decision-Making in complex games
  - 19 Ellen Rusman (OU), The Mind's Eye on Personal Profiles
  - 20 Qing Gu (VU), Guiding service-oriented software engineering - A view-based approach
  - 21 Linda Terlouw (TUD), Modularization and Specification of Service-Oriented Systems
  - 22 Junte Zhang (UVA), System Evaluation of Archival Description and Access
  - 23 Wouter Weerkamp (UVA), Finding People and their Utterances in Social Media
  - 24 Herwin van Welbergen (UT), Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior
  - 25 Syed Waqar ul Qounain Jaffry (VU), Analysis and Validation of Models for Trust Dynamics
  - 26 Matthijs Aart Pontier (VU), Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots
  - 27 Aniel Bhulai (VU), Dynamic website optimization through autonomous management of design patterns
  - 28 Rianne Kaptein (UVA), Effective Focused Retrieval by Exploiting Query Context and Document Structure
  - 29 Faisal Kamiran (TUE), Discrimination-aware Classification

- 
- 30 Egon van den Broek (UT), Affective Signal Processing (ASP): Unraveling the mystery of emotions
  - 31 Ludo Waltman (EUR), Computational and Game-Theoretic Approaches for Modeling Bounded Rationality
  - 32 Nees-Jan van Eck (EUR), Methodological Advances in Bibliometric Mapping of Science
  - 33 Tom van der Weide (UU), Arguing to Motivate Decisions
  - 34 Paolo Turrini (UU), Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations
  - 35 Maaïke Harbers (UU), Explaining Agent Behavior in Virtual Training
  - 36 Erik van der Spek (UU), Experiments in serious game design: a cognitive approach
  - 37 Adriana Burlutiu (RUN), Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference
  - 38 Nyree Lemmens (UM), Bee-inspired Distributed Optimization
  - 39 Joost Westra (UU), Organizing Adaptation using Agents in Serious Games
  - 40 Viktor Clerc (VU), Architectural Knowledge Management in Global Software Development
  - 41 Luan Ibraimi (UT), Cryptographically Enforced Distributed Data Access Control
  - 42 Michal Sindlar (UU), Explaining Behavior through Mental State Attribution
  - 43 Henk van der Schuur (UU), Process Improvement through Software Operation Knowledge
  - 44 Boris Reuderink (UT), Robust Brain-Computer Interfaces
  - 45 Herman Stehouwer (UvT), Statistical Language Models for Alternative Sequence Selection
  - 46 Beibei Hu (TUD), Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work
  - 47 Azizi Bin Ab Aziz (VU), Exploring Computational Models for Intelligent Support of Persons with Depression
  - 48 Mark Ter Maat (UT), Response Selection and Turn-taking for a Sensitive Artificial Listening Agent
  - 49 Andreea Niculescu (UT), Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality
- 
- 2012 01 Terry Kakeeto (UvT), Relationship Marketing for SMEs in Uganda
  - 02 Muhammad Umair (VU), Adaptivity, emotion, and Rationality in Human and Ambient Agent Models
  - 03 Adam Vanya (VU), Supporting Architecture Evolution by Mining Software Repositories
  - 04 Jurriaan Souer (UU), Development of Content Management System-based Web Applications
  - 05 Marijn Plomp (UU), Maturing Interorganisational Information Systems
  - 06 Wolfgang Reinhardt (OU), Awareness Support for Knowledge Workers in Research Networks
  - 07 Rianne van Lambalgen (VU), When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions
  - 08 Gerben de Vries (UVA), Kernel Methods for Vessel Trajectories
  - 09 Ricardo Neisse (UT), Trust and Privacy Management Support for Context-Aware Service Platforms
  - 10 David Smits (TUE), Towards a Generic Distributed Adaptive Hypermedia Environment
  - 11 J.C.B. Rantham Prabhakara (TUE), Process Mining in the Large: Preprocessing, Discovery, and Diagnostics
  - 12 Kees van der Sluijs (TUE), Model Driven Design and Data Integration in Semantic Web Information Systems
  - 13 Suleman Shahid (UvT), Fun and Face: Exploring non-verbal expressions of emotion during playful interactions
  - 14 Evgeny Knutov (TUE), Generic Adaptation Framework for Unifying Adaptive Web-based Systems
  - 15 Natalie van der Wal (VU), Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes.
  - 16 Fiemke Both (VU), Helping people by understanding them - Ambient Agents supporting task execution and depression treatment
  - 17 Amal Elgammal (UvT), Towards a Comprehensive Framework for Business Process Compliance

- 
- 18 Eltjo Poort (VU), Improving Solution Architecting Practices
  - 19 Helen Schonenberg (TUE), What's Next? Operational Support for Business Process Execution
  - 20 Ali Bahramisharif (RUN), Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing
  - 21 Roberto Cornacchia (TUD), Querying Sparse Matrices for Information Retrieval
  - 22 Thijs Vis (UvT), Intelligence, politie en veiligheidsdienst: verenigbare grootheden?
  - 23 Christian Muehl (UT), Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction
  - 24 Laurens van der Werff (UT), Evaluation of Noisy Transcripts for Spoken Document Retrieval
  - 25 Silja Eckartz (UT), Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application
  - 26 Emile de Maat (UVA), Making Sense of Legal Text
  - 27 Hayretin Gurkok (UT), Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games
  - 28 Nancy Pascall (UvT), Engendering Technology Empowering Women
  - 29 Almer Tigelaar (UT), Peer-to-Peer Information Retrieval
  - 30 Alina Pommeranz (TUD), Designing Human-Centered Systems for Reflective Decision Making
  - 31 Emily Bagarakayo (RUN), A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure
  - 32 Wietske Visser (TUD), Qualitative multi-criteria preference representation and reasoning
  - 33 Rory Sie (OUN), Coalitions in Cooperation Networks (COCOON)
  - 34 Pavol Jancura (RUN), Evolutionary analysis in PPI networks and applications
  - 35 Evert Haasdijk (VU), Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics
  - 36 Denis Ssebugwawo (RUN), Analysis and Evaluation of Collaborative Modeling Processes
  - 37 Agnes Nakakawa (RUN), A Collaboration Process for Enterprise Architecture Creation
  - 38 Selmar Smit (VU), Parameter Tuning and Scientific Testing in Evolutionary Algorithms
  - 39 Hassan Fatemi (UT), Risk-aware design of value and coordination networks
  - 40 Agus Gunawan (UvT), Information Access for SMEs in Indonesia
  - 41 Sebastian Kelle (OU), Game Design Patterns for Learning
  - 42 Dominique Verpoorten (OU), Reflection Amplifiers in self-regulated Learning
  - 43 Withdrawn
  - 44 Anna Tordai (VU), On Combining Alignment Techniques
  - 45 Benedikt Kratz (UvT), A Model and Language for Business-aware Transactions
  - 46 Simon Carter (UVA), Exploration and Exploitation of Multilingual Data for Statistical Machine Translation
  - 47 Manos Tsagkias (UVA), Mining Social Media: Tracking Content and Predicting Behavior
  - 48 Jorn Bakker (TUE), Handling Abrupt Changes in Evolving Time-series Data
  - 49 Michael Kaisers (UM), Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions
  - 50 Steven van Kervel (TUD), Ontology driven Enterprise Information Systems Engineering
  - 51 Jeroen de Jong (TUD), Heuristics in Dynamic Scheduling; a practical framework with a case study in elevator dispatching
- 
- 2013 01 Viorel Milea (EUR), News Analytics for Financial Decision Support
  - 02 Erietta Liarou (CWI), MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing
  - 03 Szymon Klarman (VU), Reasoning with Contexts in Description Logics
  - 04 Chetan Yadati (TUD), Coordinating autonomous planning and scheduling
  - 05 Dulce Pumareja (UT), Groupware Requirements Evolutions Patterns
  - 06 Romulo Goncalves (CWI), The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience
  - 07 Giel van Lankveld (UvT), Quantifying Individual Player Differences
  - 08 Robbert-Jan Merk (VU), Making enemies: cognitive modeling for opponent agents in fighter pilot simulators

- 
- 09 Fabio Gori (RUN), Metagenomic Data Analysis: Computational Methods and Applications
  - 10 Jeewanee Jayasinghe Arachchige (UvT), A Unified Modeling Framework for Service Design.
  - 11 Evangelos Pournaras (TUD), Multi-level Reconfigurable Self-organization in Overlay Services
  - 12 Marian Razavian (VU), Knowledge-driven Migration to Services
  - 13 Mohammad Safiri (UT), Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly
  - 14 Jafar Tanha (UVA), Ensemble Approaches to Semi-Supervised Learning Learning
  - 15 Daniel Hennes (UM), Multiagent Learning - Dynamic Games and Applications
  - 16 Eric Kok (UU), Exploring the practical benefits of argumentation in multi-agent deliberation
  - 17 Koen Kok (VU), The PowerMatcher: Smart Coordination for the Smart Electricity Grid
  - 18 Jeroen Janssens (UvT), Outlier Selection and One-Class Classification
  - 19 Renze Steenhuizen (TUD), Coordinated Multi-Agent Planning and Scheduling
  - 20 Katja Hofmann (UvA), Fast and Reliable Online Learning to Rank for Information Retrieval
  - 21 Sander Wubben (UvT), Text-to-text generation by monolingual machine translation
  - 22 Tom Claassen (RUN), Causal Discovery and Logic
  - 23 Patricio de Alencar Silva (UvT), Value Activity Monitoring
  - 24 Haitham Bou Ammar (UM), Automated Transfer in Reinforcement Learning
  - 25 Agnieszka Anna Latoszek-Berendsen (UM), Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System
  - 26 Alireza Zarghami (UT), Architectural Support for Dynamic Homecare Service Provisioning
  - 27 Mohammad Huq (UT), Inference-based Framework Managing Data Provenance
  - 28 Frans van der Sluis (UT), When Complexity becomes Interesting: An Inquiry into the Information eXperience
  - 29 Iwan de Kok (UT), Listening Heads
  - 30 Joyce Nakatumba (TUE), Resource-Aware Business Process Management: Analysis and Support
  - 31 Dinh Khoa Nguyen (UvT), Blueprint Model and Language for Engineering Cloud Applications
  - 32 Kamakshi Rajagopal (OUN), Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development
  - 33 Qi Gao (TUD), User Modeling and Personalization in the Microblogging Sphere
  - 34 Kien Tjin-Kam-Jet (UT), Distributed Deep Web Search
  - 35 Abdallah El Ali (UvA), Minimal Mobile Human Computer Interaction
  - 36 Than Lam Hoang (TUE), Pattern Mining in Data Streams
  - 37 Dirk Börner (OUN), Ambient Learning Displays
  - 38 Eelco den Heijer (VU), Autonomous Evolutionary Art
  - 39 Joop de Jong (TUD), A Method for Enterprise Ontology based Design of Enterprise Information Systems
  - 40 Pim Nijssen (UM), Monte-Carlo Tree Search for Multi-Player Games
  - 41 Jochem Liem (UVA), Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning
  - 42 Léon Planken (TUD), Algorithms for Simple Temporal Reasoning
  - 43 Marc Bron (UVA), Exploration and Contextualization through Interaction and Concepts
- 
- 2014 01 Nicola Barile (UU), Studies in Learning Monotone Models from Data
  - 02 Fiona Tulyano (RUN), Combining System Dynamics with a Domain Modeling Method
  - 03 Sergio Raul Duarte Torres (UT), Information Retrieval for Children: Search Behavior and Solutions
  - 04 Hanna Jochmann-Mannak (UT), Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation
  - 05 Jurriaan van Reijnsen (UU), Knowledge Perspectives on Advancing Dynamic Capability
  - 06 Damian Tamburri (VU), Supporting Networked Software Development
  - 07 Arya Adriansyah (TUE), Aligning Observed and Modeled Behavior

- 
- 08 Samur Araujo (TUD), Data Integration over Distributed and Heterogeneous Data End-points
  - 09 Philip Jackson (UvT), Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language
  - 10 Ivan Salvador Razo Zapata (VU), Service Value Networks
  - 11 Janneke van der Zwaan (TUD), An Empathic Virtual Buddy for Social Support
  - 12 Willem van Willigen (VU), Look Ma, No Hands: Aspects of Autonomous Vehicle Control
  - 13 Arlette van Wissen (VU), Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains
  - 14 Yangyang Shi (TUD), Language Models With Meta-information
  - 15 Natalya Mogles (VU), Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare
  - 16 Krystyna Milian (VU), Supporting trial recruitment and design by automatically interpreting eligibility criteria
  - 17 Kathrin Dentler (VU), Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability
  - 18 Mattijs Ghijsen (UVA), Methods and Models for the Design and Study of Dynamic Agent Organizations
  - 19 Vinicius Ramos (TUE), Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support
  - 20 Mena Habib (UT), Named Entity Extraction and Disambiguation for Informal Text: The Missing Link
  - 21 Cassidy Clark (TUD), Negotiation and Monitoring in Open Environments
  - 22 Marieke Peeters (UU), Personalized Educational Games - Developing agent-supported scenario-based training
  - 23 Eleftherios Sidirourgos (UvA/CWI), Space Efficient Indexes for the Big Data Era
  - 24 Davide Ceolin (VU), Trusting Semi-structured Web Data
  - 25 Martijn Lappenschaar (RUN), New network models for the analysis of disease interaction
  - 26 Tim Baarslag (TUD), What to Bid and When to Stop
  - 27 Rui Jorge Almeida (EUR), Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty
  - 28 Anna Chmielowiec (VU), Decentralized k-Clique Matching
  - 29 Jaap Kabbedijk (UU), Variability in Multi-Tenant Enterprise Software
  - 30 Peter de Cock (UvT), Anticipating Criminal Behaviour
  - 31 Leo van Moergestel (UU), Agent Technology in Agile Multiparallel Manufacturing and Product Support
  - 32 Naser Ayat (UvA), On Entity Resolution in Probabilistic Data
  - 33 Tesfa Tegegne (RUN), Service Discovery in eHealth
  - 34 Christina Manteli (VU), The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems.
  - 35 Joost van Ooijen (UU), Cognitive Agents in Virtual Worlds: A Middleware Design Approach
  - 36 Joos Buijs (TUE), Flexible Evolutionary Algorithms for Mining Structured Process Models
  - 37 Maral Dadvar (UT), Experts and Machines United Against Cyberbullying
  - 38 Danny Plass-Oude Bos (UT), Making brain-computer interfaces better: improving usability through post-processing.
  - 39 Jasmina Maric (UvT), Web Communities, Immigration, and Social Capital
  - 40 Walter Omona (RUN), A Framework for Knowledge Management Using ICT in Higher Education
  - 41 Frederic Hogenboom (EUR), Automated Detection of Financial Events in News Text
  - 42 Carsten Eijckhof (CWI/TUD), Contextual Multidimensional Relevance Models
  - 43 Kevin Vlaanderen (UU), Supporting Process Improvement using Method Increments
  - 44 Paulien Meesters (UvT), Intelligent Blauw. Met als ondertitel: Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden.
  - 45 Birgit Schmitz (OUN), Mobile Games for Learning: A Pattern-Based Approach
  - 46 Ke Tao (TUD), Social Web Data Analytics: Relevance, Redundancy, Diversity
  - 47 Shangsong Liang (UVA), Fusion and Diversification in Information Retrieval
-



- 
- 02 Faiza Bukhsh (UvT), Smart auditing: Innovative Compliance Checking in Customs Controls
  - 03 Twan van Laarhoven (RUN), Machine learning for network data
  - 04 Howard Spoelstra (OUN), Collaborations in Open Learning Environments
  - 05 Christoph Bösch (UT), Cryptographically Enforced Search Pattern Hiding
  - 06 Farideh Heidari (TUD), Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes
  - 07 Maria-Hendrike Peetz (UvA), Time-Aware Online Reputation Analysis
  - 08 Jie Jiang (TUD), Organizational Compliance: An agent-based model for designing and evaluating organizational interactions
  - 09 Randy Klaassen (UT), HCI Perspectives on Behavior Change Support Systems
  - 10 Henry Hermans (OUN), OpenU: design of an integrated system to support lifelong learning
  - 11 Yongming Luo (TUE), Designing algorithms for big graph datasets: A study of computing bisimulation and joins
  - 12 Julie M. Birkholz (VU), Modi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks
  - 13 Giuseppe Procaccianti (VU), Energy-Efficient Software
  - 14 Bart van Straalen (UT), A cognitive approach to modeling bad news conversations
  - 15 Klaas Andries de Graaf (VU), Ontology-based Software Architecture Documentation
  - 16 Changyun Wei (UT), Cognitive Coordination for Cooperative Multi-Robot Teamwork
  - 17 André van Cleeff (UT), Physical and Digital Security Mechanisms: Properties, Combinations and Trade-offs
  - 18 Holger Pirk (CWI), Waste Not, Want Not! - Managing Relational Data in Asymmetric Memories
  - 19 Bernardo Tabuenca (OUN), Ubiquitous Technology for Lifelong Learners
  - 20 Lois Vanhée (UU), Using Culture and Values to Support Flexible Coordination
  - 21 Sibren Fetter (OUN), Using Peer-Support to Expand and Stabilize Online Learning
  - 22 Zheming Zhu (UT), Co-occurrence Rate Networks
  - 23 Luit Gazendam (VU), Catalogue Support in Cultural Heritage
  - 24 Richard Berendsen (UVA), Finding People, Papers, and Posts: Vertical Search Algorithms and Evaluation
  - 25 Steven Woudenberg (UU), Bayesian Tools for Early Disease Detection
  - 26 Alexander Hogenboom (EUR), Sentiment Analysis of Text Guided by Semantics and Structure
  - 27 Sándor Héman (CWI), Updating compressed column stores
  - 28 Janet Bagorogoza (TiU), Knowledge Management and High Performance; The Uganda Financial Institutions Model for HPO
  - 29 Hendrik Baier (UM), Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains
  - 30 Kiavash Bahreini (OU), Real-time Multimodal Emotion Recognition in E-Learning
  - 31 Yakup Koç (TUD), On the robustness of Power Grids
  - 32 Jerome Gard (UL), Corporate Venture Management in SMEs
  - 33 Frederik Schadd (TUD), Ontology Mapping with Auxiliary Resources
  - 34 Victor de Graaf (UT), Gesocial Recommender Systems
  - 35 Jungxiao Xu (TUD), Affective Body Language of Humanoid Robots: Perception and Effects in Human Robot Interaction
- 
- 2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
  - 02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
  - 03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
  - 04 Laurens Rietveld (VU), Publishing and Consuming Linked Data
  - 05 Evgeny Sherkhonov (UVA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
  - 06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
  - 07 Jeroen de Man (VU), Measuring and modeling negative emotions for virtual training
  - 08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data

- 
- 09 Archana Nottamkandath (VU), Trusting Crowdsourced Information on Cultural Artefacts
  - 10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
  - 11 Anne Schuth (UVA), Search Engines that Learn from Their Users
  - 12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
  - 13 Nana Baah Gyan (VU), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
  - 14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
  - 15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
  - 16 Guangliang Li (UVA), Socially Intelligent Autonomous Agents that Learn from Human Reward
  - 17 Berend Weel (VU), Towards Embodied Evolution of Robot Organisms
  - 18 Albert Meroño Peñuela (VU), Refining Statistical Data on the Web
  - 19 Julia Efreanova (Tu/e), Mining Social Structures from Genealogical Data
  - 20 Daan Odijk (UVA), Context & Semantics in News & Web Search
  - 21 Alejandro Moreno Célieri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
  - 22 Grace Lewis (VU), Software Architecture Strategies for Cyber-Foraging Systems
  - 23 Fei Cai (UVA), Query Auto Completion in Information Retrieval
  - 24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
  - 25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
  - 26 Dilhan Thilakarathne (VU), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
  - 27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
  - 28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
  - 29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
  - 30 Ruud Mattheij (UvT), The Eyes Have It
  - 31 Mohammad Khelghati (UT), Deep web content monitoring
  - 32 Eelco Vriezেকolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
  - 33 Peter Bloem (UVA), Single Sample Statistics, exercises in learning from just one example
  - 34 Dennis Schunselaar (TUE), Configurable Process Trees: Elicitation, Analysis, and Enactment
  - 35 Zhaochun Ren (UVA), Monitoring Social Media: Summarization, Classification and Recommendation
  - 36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
  - 37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
  - 38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
  - 39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
  - 40 Christian Detweiler (TUD), Accounting for Values in Design
  - 41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
  - 42 Spyros Martzoukos (UVA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
  - 43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
  - 44 Thibault Sellam (UVA), Automatic Assistants for Database Exploration
  - 45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
  - 46 Jorge Gallego Perez (UT), Robots to Make you Happy

- 
- 47 Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
- 48 Tanja Buttler (TUD), Collecting Lessons Learned
- 49 Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
- 50 Yan Wang (UVT), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains
- 
- 2017 01 Jan-Jaap Oerlemans (UL), Investigating Cybercrime
- 02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
- 03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
- 04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
- 05 Mahdieh Shadi (UVA), Collaboration Behavior
- 06 Damir Vandić (EUR), Intelligent Information Systems for Web Product Search
- 07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
- 08 Rob Konijn (VU) , Detecting Interesting Differences: Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
- 09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
- 10 Robby van Delden (UT), (Steering) Interactive Play Behavior
- 11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
- 12 Sander Leemans (TUE), Robust Process Mining with Guarantees
- 13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
- 14 Shoshannah Tekofsky (UvT), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
- 15 Peter Berck (RUN), Memory-Based Text Correction
- 16 Aleksandr Chuklin (UVA), Understanding and Modeling Users of Modern Search Engines
- 17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
- 18 Ridho Reinanda (UVA), Entity Associations for Search
- 19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
- 20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
- 21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
- 22 Sara Magliacane (VU), Logics for causal inference under uncertainty
- 23 David Graus (UVA), Entities of Interest — Discovery in Digital Traces
- 24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
- 25 Veruska Zamborlini (VU), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
- 26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
- 27 Michiel Jooisse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
- 28 John Klein (VU), Architecture Practices for Complex Contexts
- 29 Adel Alhuraibi (UvT), From IT-Business Strategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
- 30 Wilma Latuny (UvT), The Power of Facial Expressions
- 31 Ben Ruijl (UL), Advances in computational methods for QFT calculations
- 32 Thaer Samar (RUN), Access to and Retrieval of Content in Web Archives
- 33 Brigit van Loggen (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity
- 34 Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
- 35 Martine de Vos (VU), Interpreting natural science spreadsheets
- 36 Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging

- 
- 37 Alejandro Montes Garcia (TUE), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
  - 38 Alex Kayal (TUD), Normative Social Applications
  - 39 Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR
  - 40 Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
  - 41 Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
  - 42 Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
  - 43 Maaïke de Boer (RUN), Semantic Mapping in Video Retrieval
  - 44 Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering
  - 45 Bas Testerink (UU), Decentralized Runtime Norm Enforcement
  - 46 Jan Schneider (OU), Sensor-based Learning Support
  - 47 Jie Yang (TUD), Crowd Knowledge Creation Acceleration
  - 48 Angel Suarez (OU), Collaborative inquiry-based learning
- 
- 2018 01 Han van der Aa (VUA), Comparing and Aligning Process Representations
  - 02 Felix Mannhardt (TUE), Multi-perspective Process Mining
  - 03 Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction
  - 04 Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks
  - 05 Hugo Hurdeman (UVA), Supporting the Complex Dynamics of the Information Seeking Process
  - 06 Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems
  - 07 Jieting Luo (UU), A formal account of opportunism in multi-agent systems
  - 08 Rick Smetsers (RUN), Advances in Model Learning for Software Systems
  - 09 Xu Xie (TUD), Data Assimilation in Discrete Event Simulations
  - 10 Julienka Mollee (VUA), Moving forward: supporting physical activity behavior change through intelligent technology
  - 11 Mahdi Sargolzaei (UVA), Enabling Framework for Service-oriented Collaborative Networks
  - 12 Xixi Lu (TUE), Using behavioral context in process mining
  - 13 Seyed Amin Tabatabaei (VUA), Computing a Sustainable Future
  - 14 Bart Joosten (UVT), Detecting Social Signals with Spatiotemporal Gabor Filters
  - 15 Naser Davarzani (UM), Biomarker discovery in heart failure
  - 16 Jaebok Kim (UT), Automatic recognition of engagement and emotion in a group of children
  - 17 Jianpeng Zhang (TUE), On Graph Sample Clustering
  - 18 Henriette Nakad (UL), De Notaris en Private Rechtspraak
  - 19 Minh Duc Pham (VUA), Emergent relational schemas for RDF
  - 20 Manxia Liu (RUN), Time and Bayesian Networks
  - 21 Aad Sloopmaker (OUN), EMERGO: a generic platform for authoring and playing scenario-based serious games
  - 22 Eric Fernandes de Mello Araujo (VUA), Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks
  - 23 Kim Schouten (EUR), Semantics-driven Aspect-Based Sentiment Analysis
  - 24 Jered Vroon (UT), Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots
  - 25 Riste Gligorov (VUA), Serious Games in Audio-Visual Collections
  - 26 Roelof de Vries (UT), THEORY-BASED AND TAILOR-MADE: Motivational Messages for Behavior Change Technology
  - 27 Maikel Leemans (TUE), Hierarchical Process Mining for Scalable Software Analysis**
-



# List of Figures

1.1	The software process analysis lifecycle . . . . .	4
1.2	Process mining and the three major classes of techniques focused on in this thesis. . . . .	5
1.3	Example discovered BPMN model for the web service provider process. . . . .	6
1.4	Fitness and precision metrics illustrated. . . . .	8
1.5	Annotated model learned for the web service provider process. . . . .	10
1.6	Analyzing software with static analysis, dynamic analysis, and process mining techniques. . . . .	12
1.7	Illustrative example of named submodels, recursion, and cancelation. . . . .	13
1.8	The structure of this thesis . . . . .	19
2.1	Example graph with strongly connected components indicated . . . . .	24
2.2	Example Petri net for a web service provider process . . . . .	26
2.3	Examples of unsound workflow nets . . . . .	30
2.4	Single-entry-single-exit regions . . . . .	30
2.5	Example block-structured workflow net; sound variant of the example Petri net in Figure 2.2 . . . . .	30
2.6	Examples of a counting Petri net using the reset and inhibitor extensions. . . . .	32
2.7	Example reset workflow net; sound variant of the example Petri net in Figure 2.2 . . . . .	32
2.8	Example YAWL model for a loan application process . . . . .	33
2.9	Example BPMN model for a loan application process . . . . .	34
2.10	Example Statechart model for a loan application process . . . . .	36
2.11	Example Message Sequence Diagram for a loan application process . . . . .	37
2.12	Example process tree for a loan application process . . . . .	39
2.13	The XES meta model. . . . .	46
2.14	Example Petri nets and their corresponding directly-follows graphs. . . . .	51
2.15	Directly-follows graph with frequency annotations . . . . .	52
3.1	Taxonomy positioning Model Learning and Process Discovery. . . . .	56
4.1	Cuts of the directly-follows graph for the basic operators $\rightarrow$ , $\wedge$ , $\times$ , and $\circ$ . . . . .	76
5.1	Comparison of business event logs and software event logs. . . . .	98
5.2	State machine for the method execution transactional model. . . . .	104

6.1	Case 1 from Table 6.1 depicts as intervals	112
6.2	Example hierarchical process tree for the event log in Table 6.1	118
6.3	Example recursive hierarchical process tree	121
6.4	Hierarchical Synthetic Evaluation – Heuristics miner result	142
6.5	Hierarchical Synthetic Evaluation – ILP miner result	142
6.6	Hierarchical Synthetic Evaluation – Genetic miner result	142
6.7	Hierarchical Synthetic Evaluation – Alpha miner result	143
6.8	Hierarchical Synthetic Evaluation – Fuzzy miner result	143
6.9	Hierarchical Synthetic Evaluation – ETMd miner result	144
6.10	Hierarchical Synthetic Evaluation – TS Regions result	144
6.11	Hierarchical Synthetic Evaluation – MINT ktails result	145
6.12	Hierarchical Synthetic Evaluation – Synoptic miner result	145
6.13	Hierarchical Synthetic Evaluation – MINT redblue result	146
6.14	Hierarchical Synthetic Evaluation – Inductive miner result	146
6.15	Hierarchical Synthetic Evaluation – NHD and RAD process tree results	147
6.16	Hierarchical Synthetic Evaluation – NHD and RAD statechart results	148
6.17	Hierarchical Synthetic Evaluation – NHD and RAD MSD results	149
6.18	The NHD result for the JUnit 4.12 software event log, visualized as statecharts.	158
7.1	Example cancelation process tree for the event log in Table 7.1	167
7.2	Cuts of the directly-follows graph for the cancelation discovery algorithm.	179
7.3	Example directly-follows graph with a loop cancelation cut	181
7.4	Cancelation Synthetic Evaluation – Alpha miner result	191
7.5	Cancelation Synthetic Evaluation – Fuzzy miner result	191
7.6	Cancelation Synthetic Evaluation – Heuristics miner result	192
7.7	Cancelation Synthetic Evaluation – ILP miner result	192
7.8	Cancelation Synthetic Evaluation – Genetic miner result	192
7.9	Cancelation Synthetic Evaluation – ETMd miner result	192
7.10	Cancelation Synthetic Evaluation – TS Regions result	193
7.11	Cancelation Synthetic Evaluation – Synoptic miner result	193
7.12	Cancelation Synthetic Evaluation – MINT ktails result	193
7.13	Cancelation Synthetic Evaluation – MINT redblue result	193
7.14	Cancelation Synthetic Evaluation – Inductive miner result	194
7.15	Cancelation Synthetic Evaluation – CD process tree result	194
7.16	Cancelation Synthetic Evaluation – CD statechart result	195
7.17	The Inductive Miner (IM) and Cancelation Discovery (CD) result for the BPIC12 event log, visualized as a statechart.	203
8.1	Our performance analysis approach positioned in the context of the level at which metrics are computed.	210
8.2	Outline of the hierarchical performance analysis approach.	210
8.3	Example model for the program in Listing 8.1.	214
8.4	The flat Petri net model corresponding to the model shown in Figure 8.3.	215
8.5	Example Petri net with unbounded behavior plus traces	220
8.6	Illustration of the concepts and terminology of Section 8.3.	225

8.7	Hierarchical Petri net model from Figure 8.3 annotated with performance metrics. . . . .	238
8.8	The Yourkit Profiler [136] results on the alignments algorithm. . . . .	245
8.9	The Yourkit Profiler [136] result on the JUnit software. . . . .	246
8.10	The Dotted Chart [172] results on the JUnit event log. . . . .	246
8.11	The Dotted Chart [172] results on the alignments event log. . . . .	247
8.12	The Dotted Chart [172] results on the BPIC 2012 event log. . . . .	248
8.13	The <i>Replay for Performance Alignment</i> [21] result on the alignments event log. . . . .	248
8.14	The <i>Replay for Performance Alignment</i> [136] result on the JUnit event log. . . . .	249
8.15	The <i>Replay for Performance Alignment</i> [136] result on the BPIC 2012 event log. . . . .	249
8.16	The <i>hierarchical performance analysis</i> results on the alignments event log. . . . .	250
8.17	The <i>hierarchical performance analysis</i> results on the JUnit event log. . . . .	251
8.18	The <i>hierarchical performance analysis</i> results on the BPIC 2012 event log. . . . .	252
8.19	Stacked bar chart showing a breakdown of the running times for the end-to-end performance metric computation. . . . .	255
9.1	The Translations and Traceability Framework . . . . .	260
10.1	QR code Tool Screencast – The Statechart Workbench . . . . .	277
10.2	The implemented tools, their context, and the interaction between tools. . . . .	278
10.3	The Statechart Workbench workflow. . . . .	280
10.4	Illustration of hierarchical depth filtering. . . . .	283
10.5	The initial Statechart Workbench for setting up the heuristics. . . . .	284
10.6	The Discovery & Analysis screen in the Statechart Workbench. . . . .	285
10.7	Info popup screen in the Statechart Workbench. . . . .	285
10.8	Various model visualizations and settings panels available in the Statechart Workbench. . . . .	288
10.9	The Instrumentation Agent process. . . . .	290
10.10	The SAW Eclipse plugin. . . . .	295
10.11	Background knowledge distribution of the user study participants. . . . .	296
10.12	User experience questionnaire results for the tool user study. . . . .	299
11.1	The Software Process Analysis Methodology . . . . .	305
12.1	The time-based Dotted Chart [172] view of the JUnit event log. . . . .	325
12.2	The index-based Dotted Chart view of the JUnit event log. . . . .	326
12.3	The initial statechart model for the JUnit log. . . . .	328
12.4	Part of the JUnit model, showing when annotated methods are scanned. . . . .	329
12.5	The discovered process tree model for the JUnit log, annotated with the duration metric. . . . .	331
12.6	Part of the projected JUnit model, focussing on the Observer design pattern. . . . .	332
12.7	Part of the JUnit model, focussing on recursive behavior. . . . .	334
12.8	Schematic view of the lithography machine in the ASML case study. . . . .	336
12.9	The analyzed interface in the ASML case study. . . . .	337



12.10	The time-based Dotted Chart view of the ASML event data. . . . .	338
12.11	The initial statechart model for the ASML log at 100%. . . . .	340
12.12	The initial statechart model for the ASML log at 80%. . . . .	341
12.13	The cancelation statechart model for the ASML log at 80%. . . . .	344
12.14	The hierarchical statechart model for the ASML log at 80%. . . . .	346
12.15	Zoomed in version of the hierarchical statechart model for the ASML log. .	347
13.1	xkcd: Machine Learning . . . . .	363

# List of Tables

1.1	Example snippet of an event log for a web service provider process. . . . .	6
2.1	Reduction rules for process trees. . . . .	42
2.2	Example event data related to the process model in Figure 2.7. . . . .	44
3.1	Comparison of related model discovery and learning techniques. . . . .	59
3.2	Comparison of active learning, passive learning, and process discovery on the <i>mainstream</i> assumptions and expressivity. . . . .	64
3.3	Comparison of techniques for hierarchical decomposition and submodel identification in various process mining techniques targeted at algorithmic and visual scalability. . . . .	66
3.4	Comparison of related performance analysis techniques in software engineering and process mining. . . . .	70
4.1	Example Inductive Miner discovery on an event log, illustrating how the discovery progresses step by step. . . . .	77
4.2	Example Inductive Miner fallback cases, illustrating how the different fallback solutions work. . . . .	82
5.1	Event logs used in comparing business and software event logs. . . . .	97
5.2	The caller-callee roles for an execution of Listing 5.1. . . . .	103
5.3	The states for the two methods in Listing 5.1. . . . .	104
5.4	Mapping from the <i>method execution transactional model</i> (Figure 5.2) to the <i>standard lifecycle transactional model</i> [8]. . . . .	104
5.5	Example event location information for some events for an execution of the program in Listing 5.1. . . . .	105
6.1	Example snippet of an event log for the program in Listing 6.1. . . . .	112
6.2	Example snippet of the hierarchical event log derived from Table 6.1. . . . .	113
6.3	Example snippet of the atomic hierarchical event log derived from Table 6.2. . . . .	116
6.4	The hierarchical structure for Case 1 as given in Table 6.3. . . . .	116
6.5	Example Naïve Hierarchical Discovery on an event log, illustrating how the discovery progresses step by step. . . . .	124
6.6	Example Recursion Aware Discovery on an event log, illustrating how the discovery progresses step by step. . . . .	133

6.7	One-letter acronyms for the activities used in the hierarchical synthetic evaluation. . . . .	139
6.8	Event logs used in the hierarchical performance and scalability evaluation. .	150
6.9	The depth of the discovered hierarchy for the event logs used in the performance and scalability evaluation. . . . .	152
6.10	Comparison of algorithm running times on software event logs with hierarchical behavior. . . . .	154
6.11	Comparison of algorithm running times on non-software event logs with hierarchical behavior. . . . .	155
6.12	Comparison of model quality scores on software event logs with hierarchical behavior. . . . .	156
6.13	Comparison of algorithm running times on non-software event logs with hierarchical behavior. . . . .	157
7.1	Example snippet of an event log for the program in Listing 7.1. . . . .	166
7.2	Example cancelation process tree with its language. . . . .	169
7.3	Reduction rules for cancelation process trees. . . . .	173
7.4	Example Cancelation Discovery on an event log, illustrating how the discovery progresses step by step. . . . .	177
7.5	Combined conditions base cases for hierarchical cancelation discovery. . . .	187
7.6	One-letter acronyms for the activities used in the cancelation synthetic evaluation. . . . .	188
7.7	Event logs used in the cancelation performance and scalability evaluation. .	196
7.8	The trigger oracles used for the event logs in the cancelation evaluation. . .	197
7.9	Comparison of algorithm running times on software event logs with cancelation behavior. . . . .	199
7.10	Comparison of algorithm running times on non-software event logs with cancelation behavior. . . . .	200
7.11	Comparison of model quality scores on software event logs with cancelation behavior. . . . .	201
7.12	Comparison of algorithm running times on non-software event logs with cancelation behavior. . . . .	202
8.1	Example snippet of an event log for the program in Listing 8.1. . . . .	212
8.2	Average running time for the end-to-end performance metric computation for different computational phases and model sizes. . . . .	255
9.1	From the basic process tree operators to a basic Petri net. . . . .	262
9.2	From the hierarchical process tree operators to a basic Petri net. . . . .	263
9.3	From the cancelation process tree operators to a basic Petri net. . . . .	265
9.4	From the extended process tree to a YAWL model. . . . .	269
9.5	From the extended process tree to a BPMN model. . . . .	270
9.6	From the extended process tree to a Statechart. . . . .	271
9.7	From the extended process tree to a Message Sequence Diagram. . . . .	272

# List of Symbols

- $\mathbb{A}$  Alphabet (finite set) of activities, 26, 47
- $LC$  Alphabet (finite set) of lifecycle transitions, 49
- $a+s$  Activity annotated with the *start* lifecycle transition, 49
- $a+c$  Activity annotated with the *complete* lifecycle transition, 49
- $\tau$  Silent activity or  $\tau$ -transition, 25
- $G(L)$  Directly-follows graph over  $L$ , 50
- $Start(G)$  Directly-follows graph: Set of start activities, 50
- $End(G)$  Directly-follows graph: Set of end activities, 50
- $\rightsquigarrow_L$  Directly-follows relation, 50
- $\triangleright$  Directly-follows relation: denotes the start of a trace, 50
- $\square$  Directly-follows relation: denotes the end of a trace, 50
- $\#_{act}(e)$  Event Attribute: the *activity* associated to event  $e$ , 45
- $\#_{time}(e)$  Event Attribute: the *timestamp* of event  $e$ , 45
- $\#_{res}(e)$  Event Attribute: the *resource* associated to event  $e$ , 45
- $\#_{life}(e)$  Event Attribute: the *lifecycle transaction type* associated to event  $e$ , 45
- $\#_{subtrace}(e)$  Event Attribute: the *hierarchical subtrace* associated to event  $e$ , 111
- $\lambda_{\#}(e)$  Event Classifier: a function that maps the attributes of an event onto a label, 45
- $L$  Event log, 46

- $L_A$  Event log, Atomic, 47
- $L_{LC}$  Event log, Non-Atomic, 49
- $\mathbb{A}(\sigma)$  Event log: Activity projection, set of activities in  $\sigma \in L_A$ , 48
- $\Sigma(\sigma)$  Event log: Activity projection, set of activities in the intersection of  $\Sigma \subseteq \mathbb{A}$  and  $\sigma \in L_A$ , 48
- $G = (V, E)$  Graph: a graph  $G$  with set of nodes  $V$  and set of edges  $E$ , 23
- $\Sigma_i$  Graph: a partition  $\Sigma_i \subseteq V$  in graph  $G = (V, E)$ , 24
- $(a, b) \in G$  Graph: edge  $(a, b) \in E$  exists in graph  $G$ , 23
- $a \rightsquigarrow b \in G$  Graph: there exists a (directed) path between nodes  $a$  and  $b$  in graph  $G$ , 24
- $\|\mathbb{A}(L)\|$  Hierarchical alphabet size: the maximum activity alphabet size across all hierarchy layers, 114
- $f.L$  Hierarchical concatenation: appends activity  $f \in \mathbb{A}$  to the start of each event's activity sequence, 114
- $\|L\|$  Hierarchical depth: the length of the longest event label in the event log, 114
- $L \downarrow_i^*$  Hierarchical projection: removes a prefix of length  $i$  from every event's activity sequence, 114
- $\min(I)$  Interval: minimum of set of intervals  $I$ , 226
- $\max(I)$  Interval: maximum of set of intervals  $I$ , 226
- $\text{sci}(I)$  Interval: smallest containing interval of set of intervals  $I$ , 226
- $\text{coi}(I)$  Interval: condensed overlapping intervals of set of intervals  $I$ , 226
- $\mathcal{I}$  Metrics: a set of execution subtraces, 230
- $I \in \mathcal{I}$  Metrics: an execution subtrace, 230
- $(s, c) \in I$  Metrics: an execution interval, 230
- $s, c$  Metrics: alignment moves, 230
- $T', T''$  Metrics: submodels in a system net  $SN$ , 230
- $T_F$  Metrics: A set of incoming transitions in a system net  $SN$  to filter on, 230

- $[t]$  Petri net: Enabled or Firing rule, 27
- $\mathcal{L}(SN)$  Petri net: Language, set of all traces in a labeled net, 28
- $x\bullet$  Petri net: Post set, set of output nodes, 26
- $\bullet x$  Petri net: Pre set, set of input nodes, 26
- $[PN, M]$  Petri net: Set of reachable markings in the marked net  $PN$  from marking  $M$ , 27
- $children(Q')$  Process tree function: *Node children*: the set of children for a given node  $Q'$ , 42
- $enum(Q')$  Process tree function: *Enumerate tree*: the set of nodes for a given node  $Q'$ , 42
- $parent(Q, Q')$  Process tree function: *parent node*: the parent of  $Q'$  in the tree  $Q$ , 42
- $path(Q, Q')$  Process tree function: *path sequence*: the sequence of nodes from root  $Q$  to node  $Q'$ , 42
- $\otimes_{\mathcal{L}}$  Process tree: Language-join function, 40
- $\rightarrow_{\mathcal{L}}$  Process tree language-join function: *sequence* or sequential composition, 40
- $\times_{\mathcal{L}}$  Process tree language-join function: *exclusive choice* or XOR choice, 40
- $\wedge_{\mathcal{L}}$  Process tree language-join function: *concurrency* or parallel composition, 40
- $\circlearrowleft_{\mathcal{L}}$  Process tree language-join function: *structured loop* or redo loop, 40
- $\mathcal{L}(Q)$  Process tree: Language, set of all traces, 40
- $\rightarrow$  Process tree operator: *sequence* or sequential composition, 40
- $\times$  Process tree operator: *exclusive choice* or XOR choice, 40
- $\wedge$  Process tree operator: *concurrency* or parallel composition, 40
- $\circlearrowleft$  Process tree operator: *structured loop* or redo loop, 40
- $\nabla_f(Q)$  Hierarchical process tree operator: *named subtree* with name  $f \in \mathbb{A}$  and subtree  $Q$ , 119

- $\Delta_f$  Hierarchical process tree operator: *recursive reference* to a named subtree ancestor with name  $f \in \mathbb{A}$ , 119
- $\xrightarrow{\star}$  Cancellation process tree operator: *sequence cancellation*, 167
- $\overset{\star}{\circ}$  Cancellation process tree operator: *loop cancellation*, 167
- $\star_a^C$  Cancellation process tree operator: *cancellation trigger* for activity  $a \in \mathbb{A}$  and the set of corresponding triggers  $C \subseteq \mathbb{A}$ , 167
- $\otimes$  Process tree: Set of process tree operators, 40
- $f|_X$  Projection, 23
- $X^*$  Sequence: Set of sequences containing elements of  $X$ , 22
- $\sigma \cdot \sigma'$  Sequence Concatenation, 22
- $\sigma \diamond \sigma'$  Sequence Shuffle, 22
- $\{x \mid \Phi(x)\}$  Set Comprehension: the set of all elements  $x$  such that  $\Phi(x)$  holds., 22

# Index

- 80/20 model, 58, 139, 188, 312
- Accept filter, 229
- Acceptance environment, 94
- Active learning, **56**
  - MAT, *see* Minimally Adequate Teacher
  - Minimally Adequate Teacher, **61**
- Activity (event log), 43
- Alignment, 213, 215, **216**
  - cost function, **218**
  - move (log, model, synchronous), 215, **216**
  - optimal alignment, **218**
  - standard cost function, **218**
- Alignments, 63
- AOP, *see* Aspect-Oriented Programming
- APM, *see* Application Performance Management
- Application Performance Management, 56, 92
- ASML, 335
- Aspect-Oriented Programming, 93, 290
- Atomic hierarchical event log, **114**
- Attribute (event log), 43
- Bag, *see* Multiset
- Block-structured workflow net, 29
- BPMN, 34
- Business Process Modeling Notation, *see* BPMN
- Business transactions, 95, 310
- Call graphs, 60
- Callee, 102
- Caller, 102
- Cancel algorithm, *see* Cancellation discovery
- Cancellation discovery, 175
  - `CDiscover()`, 175
  - base cases, 178
  - cancellation triggers, 176
  - cut detection, 178
  - directly-follows graph adaptation, 176
  - log split, 181
  - trigger completeness, 184
  - trigger edge, 176
  - trigger oracle, 173, 281
- Cancellation process tree, **167**
  - cancellation trigger, 167
  - loop cancellation, 167
  - petri net interpretations, 261
  - semantics, 169
  - sequence cancellation, 167
  - soundness, 172
- Cancellation region, 31
- Case (event log), 43
- Case identification, 94
- Case notion, 94
  - business transactions, 95
  - instance identifiers, 95
  - protocol runs, 95



- regeneration points, 95
- windowing, 95
- CD algorithm, *see* Cancellation discovery
- Classifier (event log), 43
- coi, *see* Interval, condensed overlapping intervals
- Completeness of software observations, 318
  - hidden influences, 318
- Composite (design pattern), 333
- Concatenation (sequence), 22
- Conformance analysis, 5, 9
- Conformance checking, 9
- Connected component (graph), 24
- CRISP DM, 304
- Cut (graph), 24
  
- Data extraction scoping, 307
- Data projection, 310
- Data retrieval strategy, 307
- Development environment, 93
- Directly-follows completeness, 85
- Directly-follows graph, **50**, 50
- Directly-follows relation, **50**, 50
- Dry-test analysis, 320
- Duplicate label problem, 64
- Dynamic analysis, 11, **56**
  
- Enabler move, 219
- Endpoint pointcut, 292
- Enhancement, 5, 10
- Event (event log), 43
- Event log, 4, 5, 43, **46**
  - activity, 43
  - activity alphabet size, 48
  - activity log-projection, 48
  - atomic event log, **47**, 47
  - atomic hierarchical event log, **114**
  - attribute, 43, **45**
  - case, 43
  - classifier, 43, **45**
  - event, 43, **45**
  - hierarchical event log, **111**, 111
  - log size, 48
  - non-atomic event log, 48, **49**
  - process instance, 43
  - structure, 43
  - trace, 43
  - XES, 46
- Exception (software), 107
- Execution correlation, 227
- Execution interval, 219, **224**
- Execution intervals, 227
- Execution policy, 219
  - continuously, **221**
  - race with age memory, **221**
  - race with enabling memory, **221**
- Execution subtrace, 219, **227**
- Extensible Event Stream, *see* XES46
  
- Fitness, 7
- Flat activities/models, 99, 109
- Flat model, 213
- Flower model, 42
- Function, 23
  
- Graph, 23
  - connected component, 24
  - cut, 24
  - path, 24
  - strongly connected component, 24
  
- Handled exception (software), 107
- Hierarchical concatenation, 114
- Hierarchical event log, **111**, 111
  - heuristics, 116, 279
  - multiple attributes, 117
  - nested intervals, 116
  - structured names, 117
- hierarchical alphabet size, 114
- hierarchical concatenation, 114
- hierarchical depth, 114
- hierarchical projection, 114

- Hierarchical performance analysis framework, 219
- Hierarchical process tree, **119**
  - named subtree, 119
  - petri net interpretations, 261
  - recursive reference, 119
  - semantics, 119
  - soundness, 122
- Hierarchical projection, 114
- Hierarchy unfolding, 213
- IM (framework), *see* Inductive miner
- IMc, *see* Inductive miner, incompleteness
- IMf, *see* Inductive miner, infrequent
- Inductive miner, 74
  - base case, 74, **75**, 123, 132, 178
  - cut detection, 74, **76**, 178
  - directly-follows abstraction variant, 87
  - directly-follows completeness, 85
  - fallback, 74, **81**
  - incompleteness (IMc), 87
  - infrequent (IMf), 87
  - lifecycle aware extension, 87
  - log split, 74, **79**, 181
  - operator extension, 87
- Instance identifiers, 95, 310
- Instrumentation, 93
- Instrumentation Agent (tool), 277, 289
  - endpoint pointcut, 292
  - interface pointcut, 292
  - method pointcut, 290
  - method-call pointcut, 291
- Interface pointcut, 292
- Interval, **223**
  - condensed overlapping intervals (coi), 226
  - maximum, 226
  - minimum, 226
  - smallest containing interval (sci), 226
  - Interval correlation, **225**
    - execution correlation, 227
    - overlapping, 226
  - Interval partitioning, **225**
- Java agent, 93, 289
- JUnit, 324
- L\* life-cycle model, 304
- Label splitting, 64
- Labeled Transition System, 61
- Language, 28, 40, 49
- Learner (active learning), **61**
- Legacy software, 11
- Lifecycle transition (event log), 44
- Log files, 91
- LTS, *see* Labeled Transition System
- Mealy machine, **61**
- Message Sequence Chart, *see* MSD
- Message Sequence Diagram, *see* MSD
- Method pointcut, 290
- Method-call pointcut, 291
- Model learning, 11, 57
- Monitoring, 92
- Move enabler, **219**, 219
  - continuously enabled, **221**
  - observable, **223**
- Move enabler filter, 229
- MSD, 36
- Multiple attributes, 117
- Multiset, 21
- Naïve hierarchical discovery, 122
  - `NHDiscover()`, 123
  - base cases, 123
- Nested intervals, 116
- NHD algorithm, *see* Naïve hierarchical discovery
- Non-trivial cut (graph), 24
- Observer (design pattern), 330
- Operational support, 317

- Overlapping interval correlation, 226
- Passive learning, **56**
- Path (graph), 24
- Performance analysis, 5, 9, 10
- Performance metric
  - absolute frequency, 231
  - case frequency, 231
  - computational phases, 253
  - cumulative duration, 235
  - duration, 233
  - duration efficiency, 236
  - followed-by frequency, 232
  - model-move frequency, 231
  - own duration, 235
  - resource frequency, 232
  - service time, 233
  - sojourn time, 234
  - waiting time, 234
- Petri net, 25, **26**
  - cancelation region, 31
  - enabled, 25, **27**
  - firing, **27**
  - firing rule, 25, **27**
  - firing sequence, **27**
  - full firing sequence, **28**
  - inhibitor arc, 31
  - language, **28**
  - marking, 25
  - pre set and post set, 26
  - reachable, **27**
  - reset arc, 31
  - reset/inhibitor firing rule, **31**
  - reset/inhibitor net, **31**
  - soundness, **28**
  - system net, **27**
- PM<sup>2</sup>, 304
- PMD, *see* Process Diagnostics Method
- Power set, 21
- Precision, 8
- Process Diagnostics Method, 304
- Process discovery, 5, 6, 12, 57
  - duplicate label problem, 64
- Process instance (event log), 43
- Process mining, 4
- Process tree, 38, **40**
  - cancelation process tree, **167**
  - children, **42**
  - enumerate, **42**
  - flower model, 42
  - hierarchical process tree, **119**
  - language, **40**
  - language-join function, **40**
  - parent, **42**
  - path, **42**
- Production environment, 94
- Program monitoring, **56**
- Projection, 23
- ProM-Eclipse link, 294
- Protocol runs, 95, 310
- RAD algorithm, *see* Recursion aware discovery
- Recursion aware discovery, 128
  - RADiscover(), 130
  - RADstep(), 130
  - base cases, 132
  - context path, 129
  - delayed and repeated discovery, 129, 131
- Refined Process Structure Tree, 67
- Regeneration points, 95
- Register automata, 61
- Relaxed soundness, **29**
- Representational bias, 8, 58
- Resource (event log), 44
- RPST, *see* Refined Process Structure Tree
- SAW (tool), 278, 294
- Scalability, 9
  - algorithmic scalability, **9**, 16, 66
  - visual scalability, **9**, 16, 68

- sci, *see* Interval, smallest containing interval
- Semantic-aware filters, 229
  - accept filter, 229
  - move enabler filter, 229
- Sequence, 22
  - concatenation, 22
  - shuffle, 22
- Sequence Diagram, *see* MSD
- Set, 21
- Set Comprehension, 22
- Shuffle (sequence), 22
- Slice and dice, 310
- Software Analysis Workbench (tool), *see* SAW
- Software environment, 93
  - acceptance environment, 94
  - development environment, 93
  - production environment, 94
  - test environment, 93
- Software event location, 105
- Software event type, 103
- Software process analysis methodology, 304
  - completeness of software observations, 318
  - data extraction scoping, 307
  - data projection, 310
  - data retrieval strategy, 307
  - dry-test analysis, 320
  - operational support, 317
  - system scoping, 307
  - timings and clock, 319
  - view selection, 310
  - visual analysis workflow, 320
- Soundness, 7, **28**
  - relaxed soundness, **29**
  - weak soundness, **29**
- Source code link, 261, 294
- Statechart Workbench (tool), 277, **278**
  - heuristic for hierarchy, 279
  - trigger oracle heuristics, 281
  - workbench workflow, 279
- Statecharts, 35
- Static analysis, 11, **56**
- Strongly connected component (graph), 24
- Structured names, 117
- Supported behavior, 8
- System net, **27**
- System scoping, 307
- Test environment, 93
- Thrown exception (software), 107
- Timestamp (event log), 44
- Timings and clock (software), 319
- Trace (event log), 43
- Traceability, 261
- Tracing, 92
- Translation and traceability framework, 259
- Trigger completeness, 184
- UML SD, 61
- UML Sequence Diagram, *see* UML SD
- View selection, 310
- Visual analysis workflow, 320
- Weak soundness, **29**
- WF-net, *see* Workflow net
- Windowing, 95, 310
- Workflow net, **28**, 28
  - block-structured workflow net, 29
- Workflow patterns, 33
- XES, 46
  - extensions, 46
- YAWL, 33
- Yet Another Workflow Language, *see* YAWL